

M.Tech. Dissertation

Filter Object Framework for MICO - Dynamic Model

Submitted in partial fulfillment of requirements
for the degree of
Master of Technology

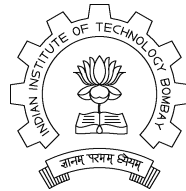
By

Amit L. Padalkar

Roll No. : 00329004

Under the guidance of

Prof. R. K. Joshi



Kanwal Rekhi School of Information Technology

Indian Institute of Technology, Bombay

Mumbai, 400 076

January 29, 2002

Dissertation Approval Sheet

This is to certify that the dissertation titled
Filter Object Framework for MICO - Dynamic Model

By

Amit L. Padalkar

(00329004)

is approved for the degree of **Master of Technology**.

Prof. R. K. Joshi

(Guide)

Internal Examiner

External Examiner

Chairperson

Date: _____

Acknowledgments

I take this opportunity to express my sincere gratitude to my guide **Prof. R. K. Joshi** for his support, encouragement and advice. His constant guidance has been invaluable throughout the span of the project.

I have worked with my classmate at the School of Information Technology, Pranav Nabar. I thank him for his support and encouragement. Our camaraderie helped us in realization of this work. Finally, I thank school of Information Technology for providing excellent resources and support.

Amit L. Padalkar

IIT Bombay

January 29, 2002

Abstract

Filter objects is a paradigm for separation of concerns. Filter objects are transparent objects which act on message when they are in transit. In this report, we present a design and implementation of the filtered delivery model for an open-source CORBA implementation - MICO. The model is realized by extending MICO implementation and by providing tools for easy creation of filter objects. We have analyzed the implementation model of MICO from behavioral point of view in order to provide filter object support in the MICO CORBA kernel. The analysis of the implementation model of MICO is presented initially. Various design choices were available for integrating filter objects into MICO kernel. A comparison of these design alternatives is presented next. We then discuss in detail the evolution of the MICO kernel for implementing the selected design strategy. Performance implications of the filtered delivery model are also listed in this report.

Contents

- 1 Introduction** **6**
 - 1.1 Overview of filtered delivery model 7
 - 1.2 Overview of CORBA 9
 - 1.2.1 Object Request Broker 9
 - 1.2.2 Architecture 9
 - 1.2.3 Object activation policies 11
 - 1.3 Organization of the report 12

- 2 MICO Implementation - Dynamic Model** **13**
 - 2.1 Client server interaction in MICO 14
 - 2.2 MICO daemon 14
 - 2.3 Server registration 15
 - 2.3.1 Registration with the Implementation Repository . . . 15
 - 2.3.2 Registration with the Basic Object Adapter 17
 - 2.4 Object binding 20
 - 2.4.1 Binding at client side 20
 - 2.4.2 Binding in MICO daemon 22
 - 2.5 Method invocation 25
 - 2.5.1 Invocation at client side 25
 - 2.5.2 Invocation at server side 28

- 3 MICO Filter Object Framework Design** **31**
 - 3.1 Comparison 31

4	Filter Object Framework - Dynamic Model	34
4.1	Beta message handling	34
4.1.1	Upfilter/Downfilter beta messages	35
4.1.2	Enable/Disable beta messages	36
4.1.3	Plug/Unplug beta messages	37
4.2	Filtered method invocation	37
4.2.1	Method name translation	38
4.2.2	Up filtering	40
4.2.3	Down filtering	41
4.3	Persistence of framework related information	41
5	Filter Object Support in MICO	43
5.1	Filter relationship	43
5.2	Essential properties of filter objects	44
5.3	Extended properties of filter objects	45
5.4	Performance	46
6	Concluding Remarks and Future Work	49

List of Figures

1.1	Direct delivery model	7
1.2	Filtered delivery model	8
1.3	Common Object Request Broker Architecture	10
2.1	Account interface	14
2.2	MICO daemon	16
2.3	Server registration with IMR	18
2.4	Server registration with BOA	19
2.5	Client side object binding process	21
2.6	Object binding in daemon - 1	23
2.7	Object binding in daemon - 2	24
2.8	Object binding in daemon - 3	26
2.9	Client side invocation	27
2.10	Server side invocation - 1	29
2.11	Server side invocation - 2	30
4.1	Filtered method invocation	39
5.1	AccountFilter interface	44

List of Tables

3.1	Feature matrix	32
5.1	Direct call	47
5.2	Client and filter beta messages	47
5.3	Filtered call	47

Notations

OMG	: Object Management Group
CORBA	: Common Object Request Broker Architecture
MICO	: MICO is CORBA
ORB	: Object Request Broker
BOA	: Basic Object Adapter
POA	: Portable Object Adapter
IDL	: Interface Definition Language
GIOP	: General Inter Operable Protocol
IIOP	: Internet Inter Operable Protocol
IOR	: Inter Operable Reference
DII	: Dynamic Invocation Interface
DSI	: Dynamic Skeleton Interface
IMR	: Implementation Repository
micod	: BOA Daemon
TCP	: Transmission Control Protocol
IP	: Internet Protocol
DBMS	: Database Management System
RAM	: Random Access Memory
LAN	: Local Area Network

Chapter 1

Introduction

An object-oriented system is composed of many objects, which form its basic building blocks. Object communication takes place by sending messages, which are mapped to method invocations. In the direct delivery model for message passing between client and server in CORBA, the message control is integrated with the message processing code. As a result, message control cannot be separated out without breaking the transparency.

An application may require its message control policy to be changed dynamically. For example, an application might need to decrypt message contents before using them for security concerns. In cases like these, the *filtered delivery* [11] model achieves separation between message control and message processing in a transparent way. In this model, messages sent to the destination objects can be intercepted by special objects called *filter objects*. While filter objects intercept messages, the calling semantics of the source object do not change.

The filtered delivery model for CORBA based distributed systems has already been implemented. The implementation acts as a user level facility requiring the filter client to be *filter-aware* and is available for MICO. For overview of user level filters for MICO, refer [8]. For an in-depth discussion

of this facility, refer [10].

Filtering is mostly implemented as an add-on facility over existing application framework. This filtering mechanism can be designed in different ways depending on the design goals and the architecture of the existing framework, thus leading to different filter models. For overview of various filtering models, refer [8]. For a more detailed discussion of filtering models, refer [7]. For specific information on message and composition filtering models, refer [11] and [6] respectively.

1.1 Overview of filtered delivery model

The following example illustrates motivation for using filter objects in evolution of systems.

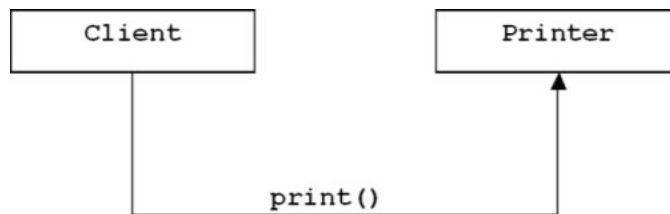


Figure 1.1: Direct delivery model

Figure 1.1 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 the **Printer**. As a result, the corresponding operation is invoked at the destination object. This implies any intermediate message control cannot be decoupled from the operation without sacrificing transparency.

For example, if after a period of time, it is found that one **Printer** is insufficient to fulfil the needs of increasing number of clients. Then one solution

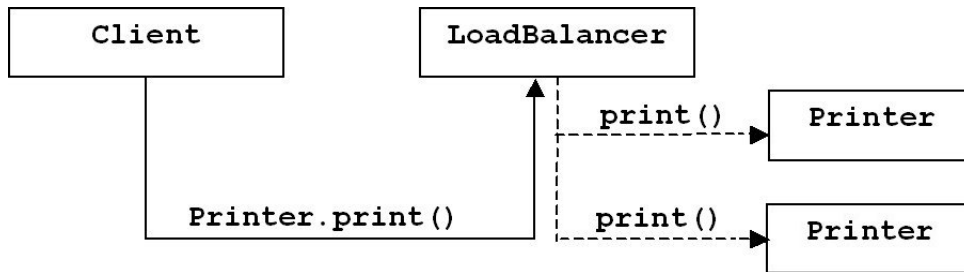


Figure 1.2: Filtered delivery model

is to add multiple printers with load balancing. But it is not possible to dynamically introduce this solution without making considerable changes to the `Printer` object code.

The filtered delivery model [11] provides an elegant, modular solution to the above problem. Filter objects act as message filters for *their* client objects. Messages to these objects are intercepted by the filter objects transparently. Removal, addition or replacement of filter objects does not require any modification of code, either at the source object or at the destination object. Figure 1.2 shows how load balancing can be effected using 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.

For details of implementing this solution using the filter object framework, refer [8]. For more general discussion on applications of the filtered delivery model, refer [5].

1.2 Overview of CORBA

In this section, we briefly introduce some important concepts related to Common Object Request Broker Architecture (CORBA). CORBA, proposed by Object Management Group (OMG), is an industry standard that defines a higher level facility for distributed computing. CORBA allows applications to communicate with one another without being aware of the hardware or software systems or the location of the application. We start by introducing Object Request Broker, followed by the CORBA architecture and object activation policies.

1.2.1 Object Request Broker

An Object Request Broker (ORB) is the central component of CORBA. ORB provides inter-operability between applications on different machines in heterogeneous distributed environments and brings together multiple object systems. It is the middleware that establishes client-server relationships between objects. Using an ORB the client can transparently invoke a method on a server object. The server object can be on the same machine or on a remote machine in a network. The ORB intercepts the method call and is responsible for finding an appropriate object, pass the parameters, invoke the method and finally return the results to the server. Objects on ORB can act as either client or server depending on the context.

1.2.2 Architecture

Figure 1.3 shows the architecture described by the CORBA standard. The client-side objects communicate with the server-side components using the underlying ORB middleware. We briefly touch upon the components on the client and the server side in the following sections.

1. *Client-side Components*

- Client IDL stubs provide static interfaces to object services. From

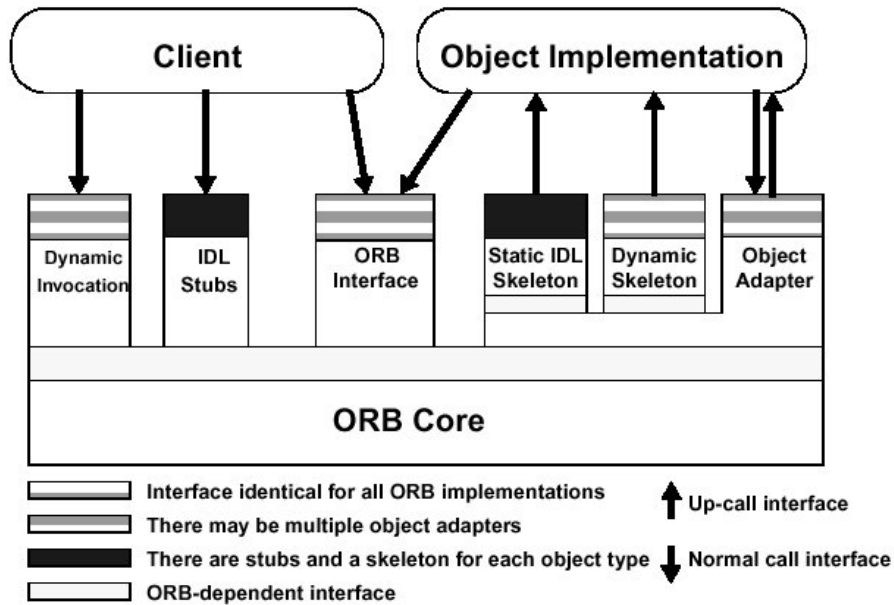


Figure 1.3: Common Object Request Broker Architecture

a clients perspective these stubs are local proxies for remote server objects.

- Dynamic Invocation Interface (DII) is used for dynamic method invocation.
- Interface Repository is a dynamic meta-data repository of the ORB that contains machine-readable versions of the IDL-defined interfaces.
- ORB Interface consists of API's for local services in client and server.

2. Server-side Components

- Server IDL skeletons, created by IDL compiler, provide static interfaces to each object exported by the server.
- Dynamic Skeleton Interface (DSI) provides run time binding for

servers that need to handle incoming method calls for components that do not have IDL stubs.

- Object adapter sits on top of the ORB core communication services and accepts requests on behalf of the server objects. It provides run-time environment for creating object references and passing parameters. It also registers the server classes with the implementation repository.
- Implementation repository provides a run-time repository of information about the classes a server supports, the objects that are instantiated and the object references.

1.2.3 Object activation policies

To get the widest application coverage, CORBA defines four object activation policies —*shared server*, *unshared server*, *server-per-method*, and *persistent server*— that specify the rules a given implementation follows for activating objects.

- In a shared server activation policy, multiple objects may reside in the same program. The BOA activates the server the first time a request is invoked on any object implemented by that server. All subsequent requests are then delivered to this server process, and BOA does not activate another server process for that implementation. Most CORBA servers use this activation policy.
- In an unshared server activation policy, each object resides in a different server process. A new server is activated the first time a request is invoked on the object. A new server is started whenever an object is requested that is not yet active, even if a server for another object with the same implementation is active. This activation policy is used whenever a dedicated object is required.
- In a server-per-method activation policy, a new server is always started each time a request is made. The server runs only for the duration of

the particular method. Several server processes for the same object—or even the same method of the same object— may be concurrently active. The best use of this policy is for running scripts or utility programs that execute once and then terminate.

- In a persistent server activation policy, servers are activated by means outside BOA. BOA treats all subsequent requests as shared server calls. The persistent server is a special case of the shared server and typically used along with DBMS, Web server etc.

For detailed information on CORBA standard, refer [9].

1.3 Organization of the report

This report discusses dynamic model of the MICO Filter Object Framework and complements with [8], which details the corresponding static model. We start in Chapter 2 by documenting the behavioral aspects of MICO implementation. In Chapter 3, we compare various design choices that were available to us and discuss the reasons behind selecting a particular design choice over others. The details of the framework design (dynamic model) appear in Chapter 4. Chapter 5 discusses framework facilities that allow it to satisfy all filter object properties. It also presents performance implications of the filtered delivery model. We conclude the report by stating our inferences and identifying the areas of future work - Chapter 6.

Chapter 2

MICO Implementation - Dynamic Model

Our goal is to incorporate the filtered delivery model into the ORB of MICO, a C++ CORBA implementation [1]. As a first step in doing this, we closely studied the implementation model of MICO, both architectural and behavioral. Since no documentation was available on distributed system behavior in MICO, we had to analyze the MICO implementation and construct the dynamic view presented in this chapter. In the following sections, we present behavioral view of the implementation discussing various stages through which the system goes, different system components, and detailed discussion of interactions among them. The architectural view of MICO implementation i.e. static model is discussed in [8].

For understanding the system behavior, we take an example of a bank which maintains accounts of its customers. Figure 2.1 shows an interface of a bank account in CORBA IDL. For more information on this example, refer [8].

```
interface Account {
    void deposit(in unsigned long amount);
    void withdraw(in unsigned long amount);
    long balance();
};
```

Figure 2.1: Account interface

2.1 Client server interaction in MICO

To allow for client server interaction, the MICO daemon must be started first. The interaction between client and server occurs in three stages viz. *server object registration*, *object binding* and *server method invocation*.

In the server object registration stage, a server implementation is registered with the *implementation repository*. The implementation repository maintains information that allows the ORB to locate and activate implementations of objects. This stage is explained in detail in Section 2.3.

In the object binding stage, a client obtains a reference of the required server. This allows a client to invoke methods on the server. The details of object binding interactions are given in Section 2.4

In method invocation stage, a client calls server methods through reference obtained in object binding stage. Section 2.5 explains the details of work that goes behind invoking a method on server object.

2.2 MICO daemon

The MICO daemon is a BOA daemon. It is part of basic object adapter that activates object implementations when their service is requested. It also contains the implementation repository. For client server interaction to

take place, MICO daemon must be started first.

The interaction diagram for MICO daemon's startup is shown in Figure 2.2. The daemon starts by initializing ORB. The initialization involves creation of ORB, IIOProxy and IIOServer. The IIOProxy facilitates interoperability among different CORBA implementations and uses TCP/IP. The IIOServer listens on specific address by binding newly created TCP transport server to it. The ORB initialization is followed by BOA initialization. It involves creation of object adapters like BOA, POA and mediators. Finally, the daemon waits in an infinite loop for any communication or events e.g. timer.

2.3 Server registration

A server offers certain services, which its clients are interested in. But before any client can use these services, the server must register itself with the implementation repository (IMR). With the help of implementation repository, ORB locates and activates server object implementations as and when requests from clients come. In this section, we provide in-depth view of server registration process with the IMR and the BOA.

2.3.1 Registration with the Implementation Repository

The interaction diagram for server registration with the implementation repository is shown in Figure 2.3. The registration process starts with ORB and BOA initialization. The ORB initialization involves binding to implementation repository and linking the repository reference with stringified name. This linkage helps in obtaining repository reference using stringified name. Actual server registration is done by calling `create` on implementation repository reference. This call stores implementation specific information like activation mode, repository id, object tag, server and program name in the IMR as implementation definition.

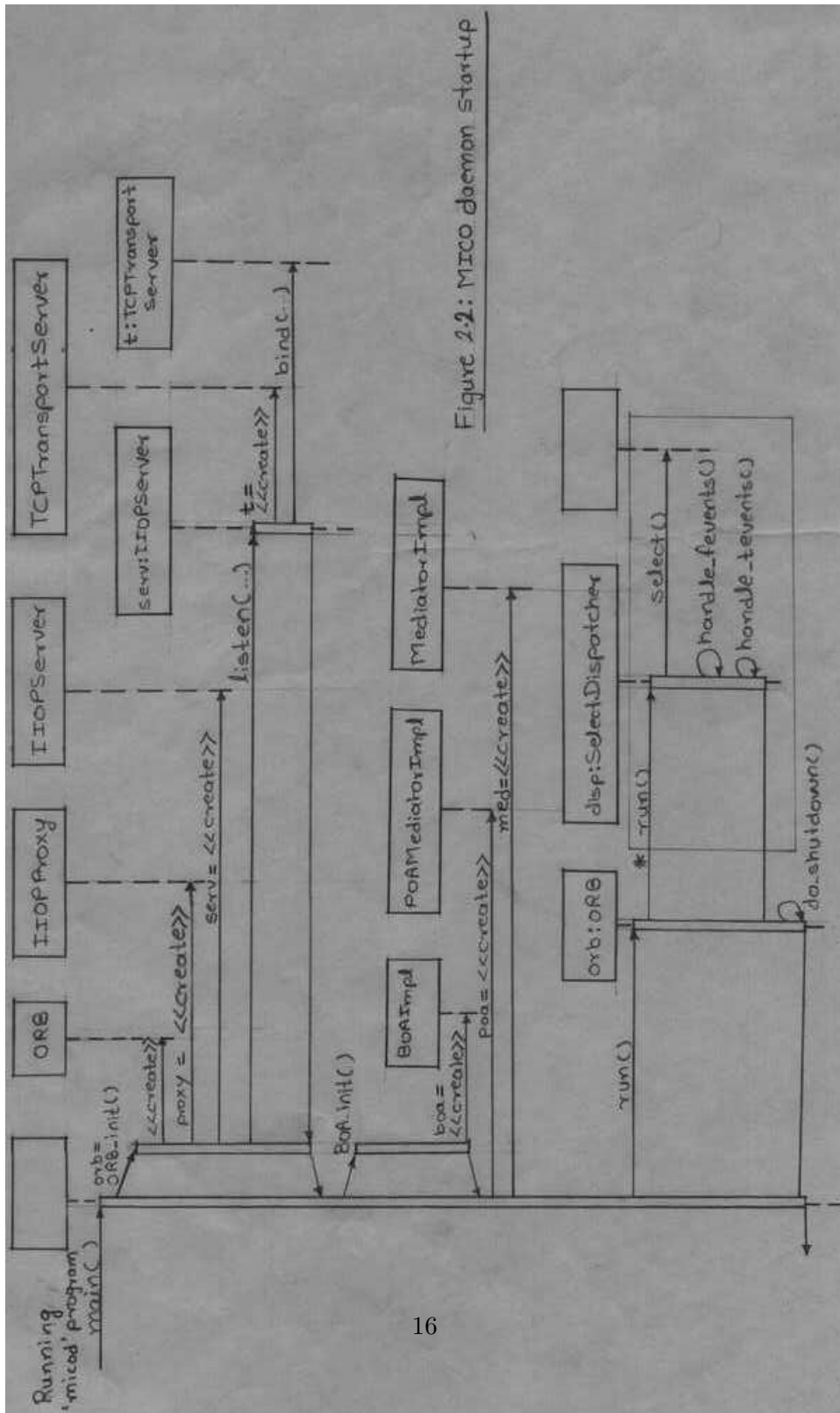


Figure 2.2: MICO daemon startup

Figure 2.2: MICO daemon

The implementation repository is a CORBA object having stubs and skeletons. Hence binding to implementation repository involves interactions with MICO daemon, which are explained thoroughly in Section 2.4. Also the `create` call is actually made on implementation repository's stub reference and like any server call in CORBA world, it travels through ORB to server side and executes on the server. The Section 2.5 explains this interaction in detail.

2.3.2 Registration with the Basic Object Adapter

The registration of server with BOA ensures that the BOA can invoke the server methods and perform other related tasks. The interaction diagram for server registration with the BOA is shown in Figure 2.4.

The registration process starts by finding an implementation definition corresponding to server's repository id and name. The process of finding implementation definition involves querying the implementation repository for sequence of implementation definitions matching the server name and selecting one of them based on repository id. Note that to obtain an implementation definition corresponding to the server, it must have been registered earlier with the implementation repository, as described in Section 2.3.1. After finding appropriate implementation definition, BOA creates and stores an object record. This record maintains details like implementation definition, skeleton reference, reference data etc. and is used while invoking server methods through BOA. The registration process completes with registration of the dispatcher, which is used for dispatching method invocations.

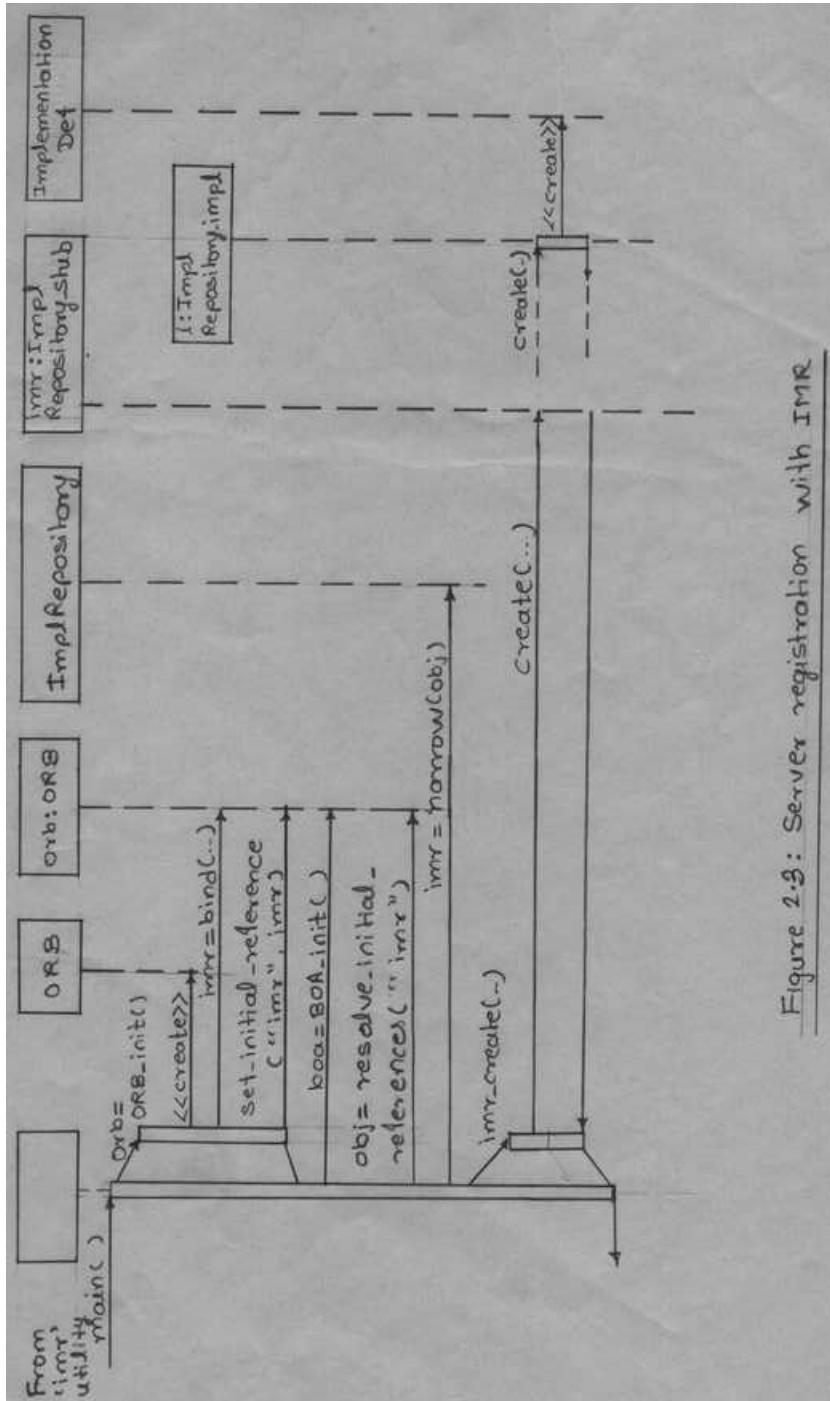


Figure 2.3: Server registration with IMR

Figure 2.3: Server registration with IMR

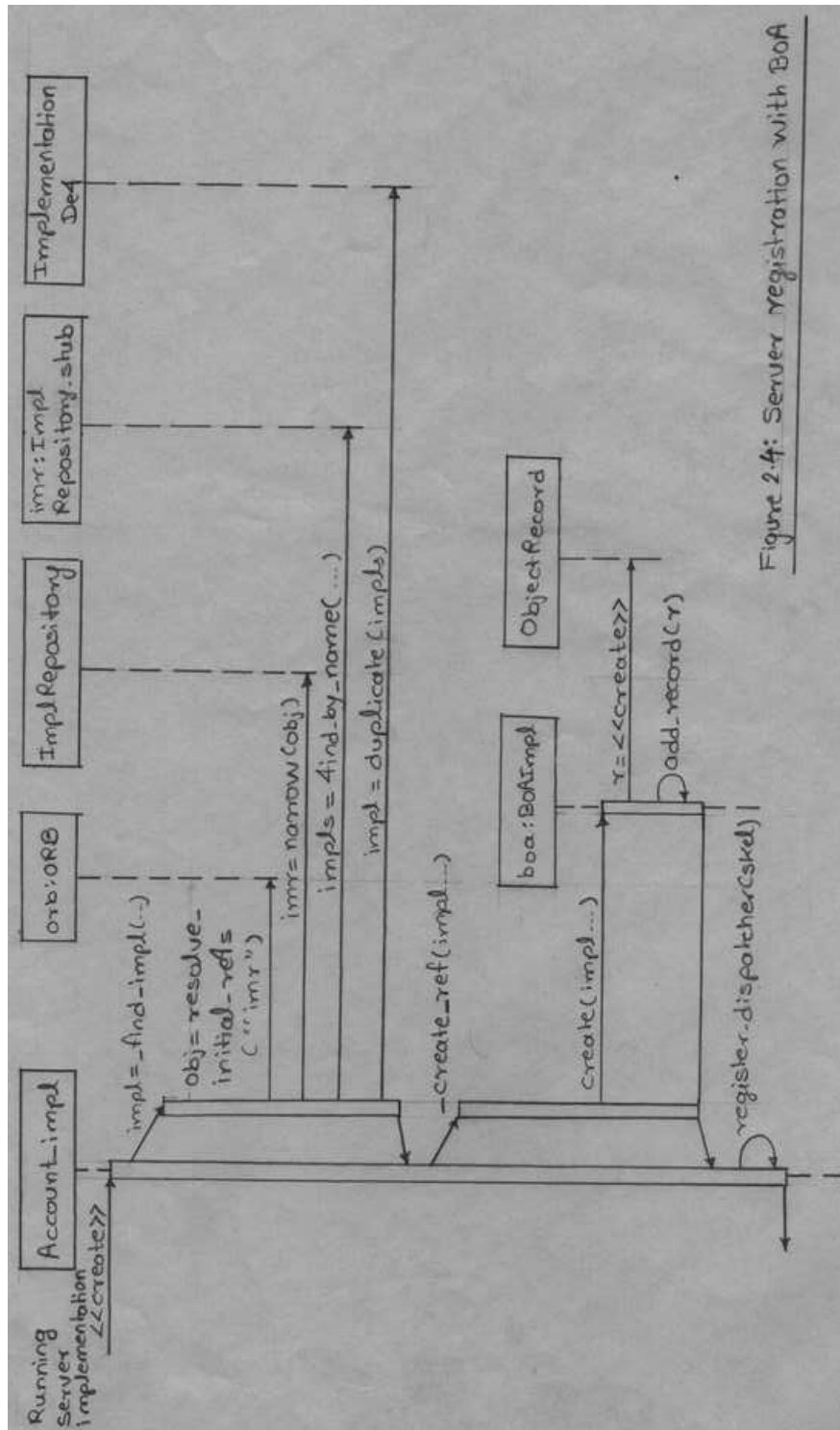


Figure 2.4: Server registration with BOA

Figure 2.4: Server registration with BOA

2.4 Object binding

For client server interaction to take place, a client must obtain a server reference. In CORBA, the object binding process returns server reference to the client. The binding process in MICO involves interaction between client and the MICO daemon.

2.4.1 Binding at client side

As shown in Figure 2.5, object binding process at client side starts when a client invokes `bind` on ORB. This call initializes an `Address` object with daemon's address and creates an object pointer to store a server reference. Asynchronous version of `bind` is then called, which creates a new message id and an ORB invocation record. This record is initialized with information like ORB reference, message id, repository id, object tag etc and is used for keeping track of invocations. It is then added to ORB invocation record list for future reference.

The binding process continues by calling `bind` on IIOP proxy. The proxy establishes a GIOP connection with the MICO daemon using TCP transport. Then, a GIOP codec encodes this bind request followed by creation and addition of the IIOP proxy invocation record representing this request to the invocation list. The `bind` call on IIOP proxy ends by outputting encoded bind request on GIOP connection.

The asynchronous bind call returns a message id which is used for waiting till the ORB invocation record indicates that the message has completed. While waiting, the dispatcher in its `run` method continuously watches for any communication from the MICO daemon. When the daemon replies, appropriate callback methods are called for handling the reply. The reply handling involves first decoding the bind reply information using GIOP codec and then setting the appropriate ORB invocation record with that information.

Once the message completes, value of server object reference is retrieved from the ORB invocation record. The record is then deleted and the server object reference is returned. Before a client can call server methods using that reference, it must be narrowed properly. Narrowing operation returns the stub reference of remote server object. This completes the client side object binding process.

2.4.2 Binding in MICO daemon

While the MICO daemon is running, it continuously monitors for any communication from clients as described in Section 2.2. When a bind request from a client comes, callback methods are invoked in daemon. These callbacks cause TCP transport server to accept the request and create TCP transport and GIOP connection for communicating with the client. This interaction is shown in Figure 2.6.

After initial acceptance by TCP transport server, the bind request handling in MICO daemon involves decoding the request using GIOP codec, creating IIOB server invocation record for it and executing it. Since this is a bind request, its execution involves calling `bind` in the mediator. The mediator tries to find any server with given repository id in implementation repository and creates an implementation record based on activation mode. Because the server is not running at time of this request, new implementation record is used to create it. The server creation involves starting a server program, specified in the implementation definition, as separate process. Since the server is not yet active, a request queue record is created for the bind request and is added to the implementation record for later retrieval. Though the server has been started, the bind request from client is yet to be fulfilled. This interaction is shown in Figure 2.7.

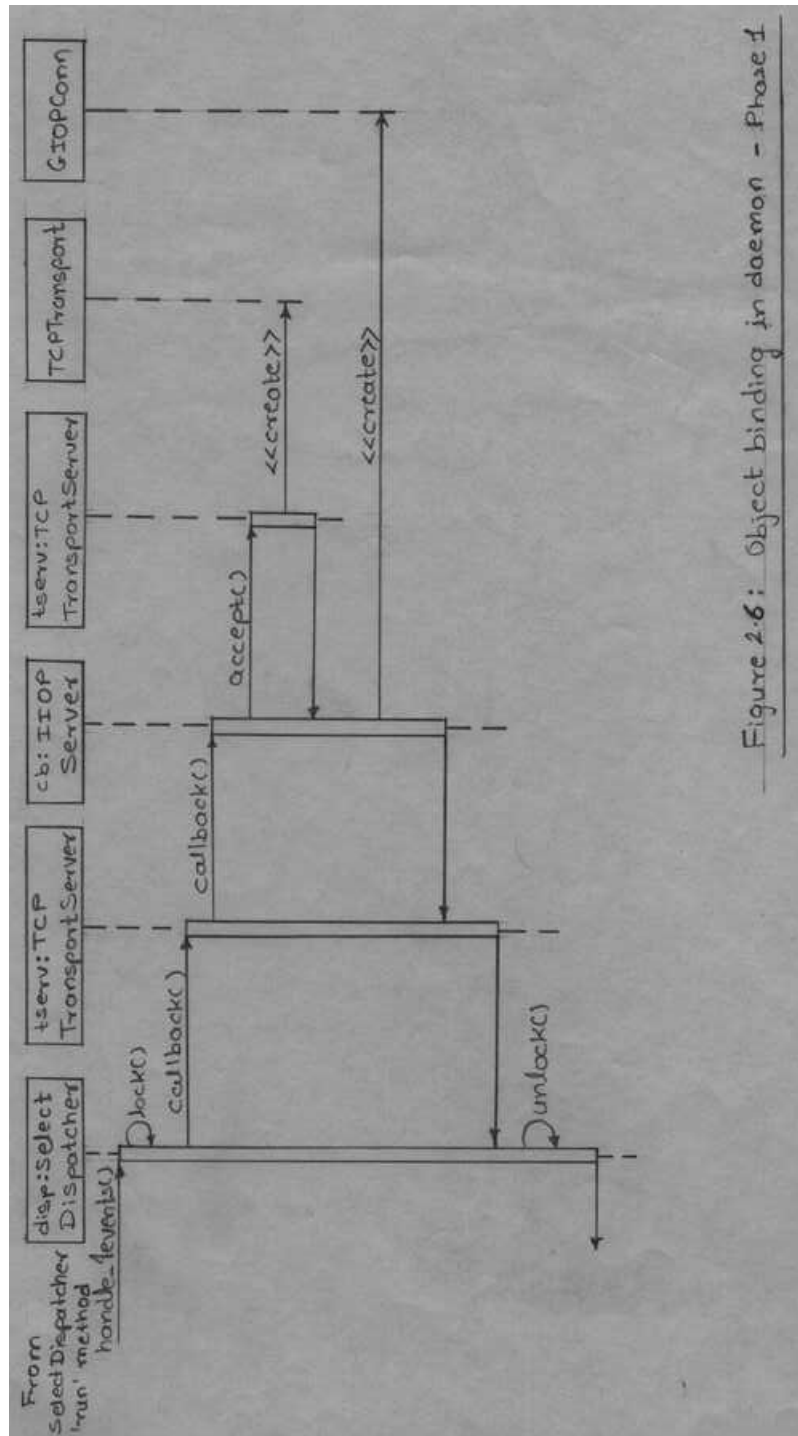


Figure 2.6: Object binding in daemon - Phase 1

Figure 2.6: Object binding in daemon - 1

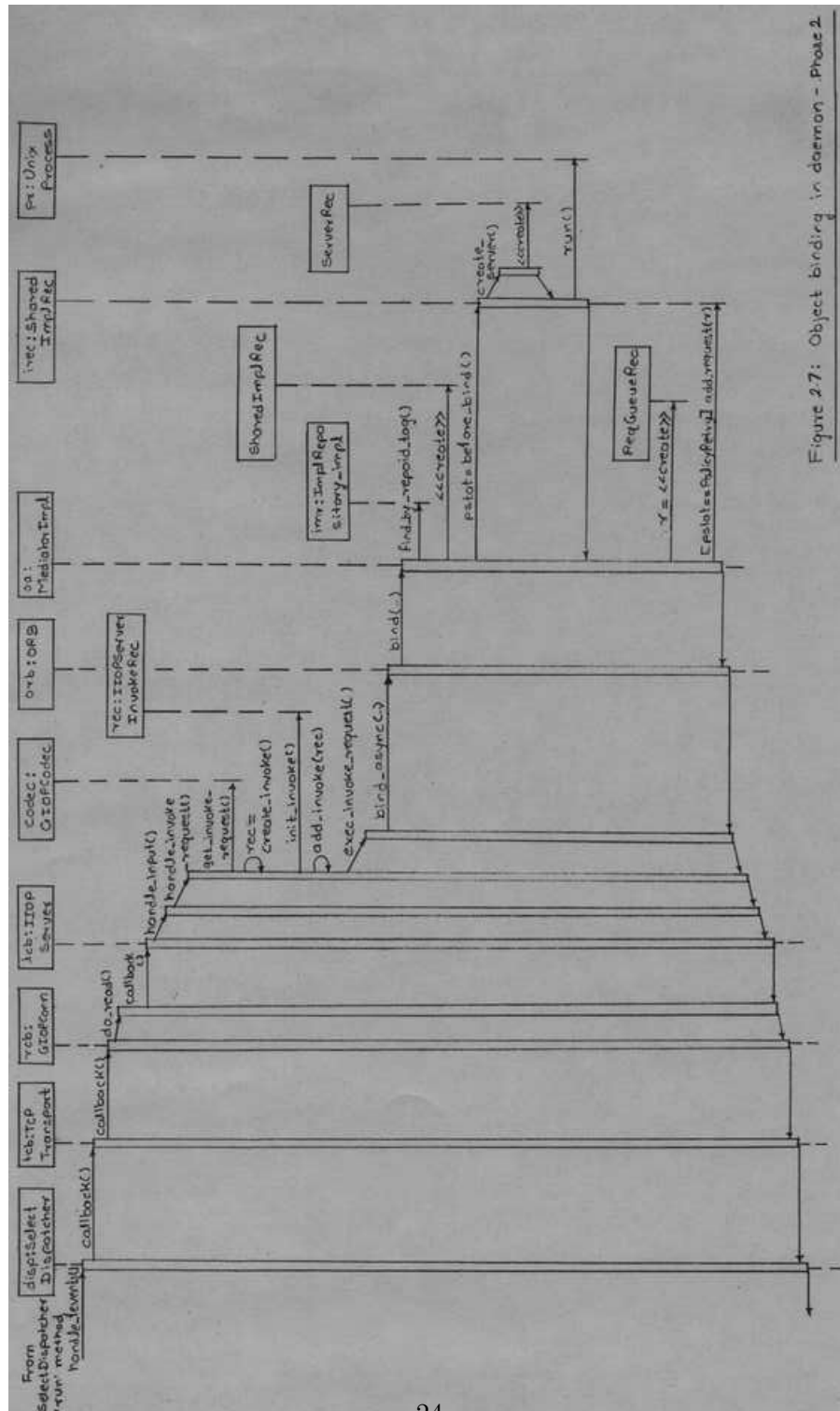


Figure 2.7: Object binding in daemon - Probe 2

Figure 2.7: Object binding in daemon - 2

When the server starts running, it registers itself with the BOA as explained in Section 2.3.2. The server then requests its activation to MICO daemon. The activation request by the server results in retrieval of request queue record added earlier by the bind request. This record is indirectly used to encode bind reply using GIOP codec. The encoded reply is then output on GIOP connection completing the object binding process in MICO daemon. This interaction is shown in Figure 2.8.

2.5 Method invocation

This section discusses the handling of server method invocation by a client in MICO. Before a client can call server methods, the server object must have registered itself with the implementation repository and the BOA as explained in Section 2.3 and the client should have called `bind` on ORB to get a server object reference as explained in Section 2.4. In the following sections, we explain the method invocation process for `balance` call.

2.5.1 Invocation at client side

The interaction diagram for client side invocation is shown in Figure 2.9. It starts with client making a `balance` call on the stub reference. The stub version of this method creates `StaticAny` objects which encapsulate method arguments and its return value. A static request, representing method invocation, is then created. It maintains information like server method name, stub reference, method arguments, return result etc. Asynchronous invocation on ORB through static request's `invoke` creates a new message id and an ORB invocation record for the current request. The invocation record is initialized with information like ORB reference, message id, request, stub reference etc. and is added to ORB invocation record list for future reference. The invocation process continues in IIOP proxy, which causes a GIOP connection to be established with the server using TCP transport. A GIOP codec encodes invocation request and outputs it on GIOP connection.

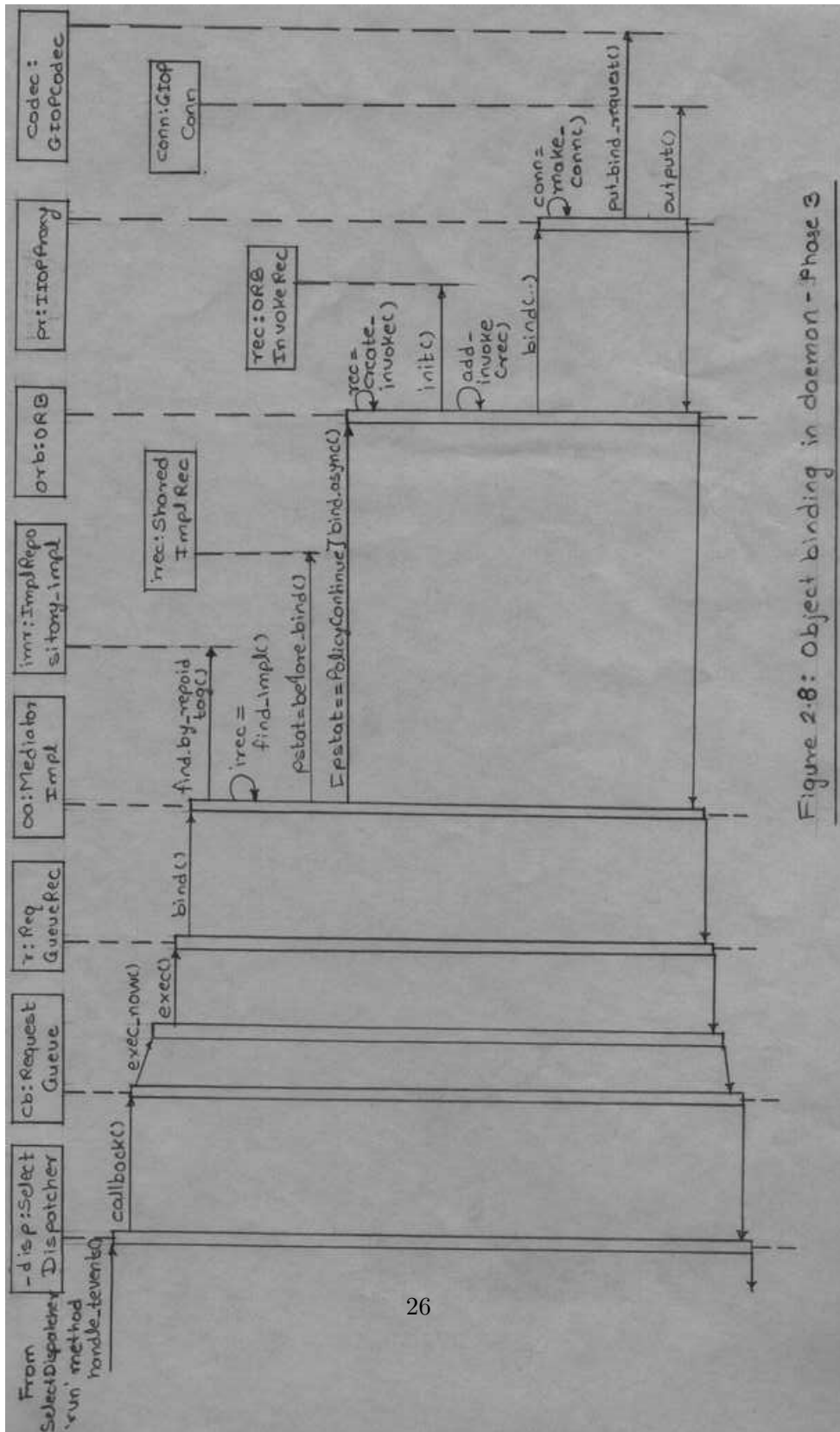


Figure 2.8: Object binding in daemon - Phase 3

Figure 2.8: Object binding in daemon - 3

The asynchronous invoke returns a message id which is used for waiting till the ORB invocation record indicates that the message has completed. While waiting, the dispatcher in its `run` method continuously watches for any communication from the server. When the server replies, appropriate callback methods are called for handling the reply. The reply handling involves first decoding the invoke reply information using GIOP codec and then setting the appropriate ORB invocation record with that information.

Once the message completes, invocation information is retrieved from the ORB invocation record. The record is then deleted and the results are returned, which completes the client side method invocation.

2.5.2 Invocation at server side

While the server is active, it continuously monitors for any communication from clients. When a invoke request from a client comes, callback methods are invoked for handling it. The request handling involves creation, initialization and addition of IIOP server and ORB invocation records representing the current request. The invocation request is then forwarded to BOA for further handling. This interaction is shown in Figure 2.10.

The invocation handling in BOA involves retrieval of corresponding object record. This record is put in BOA as explained in Section 2.3.2. The object is then loaded if the skeleton is present in object record. If the skeleton is not present, object is restored using BOA interceptors. A static server request, representing server side request, is then formed. Actual method invocation on the server then follows. The invocation status is set to normal and results are written in appropriate ORB invocation record. Finally, a reply is sent to the client by first encoding it using GIOP codec and then outputting it on the GIOP connection. This interaction is shown in Figure 2.11.

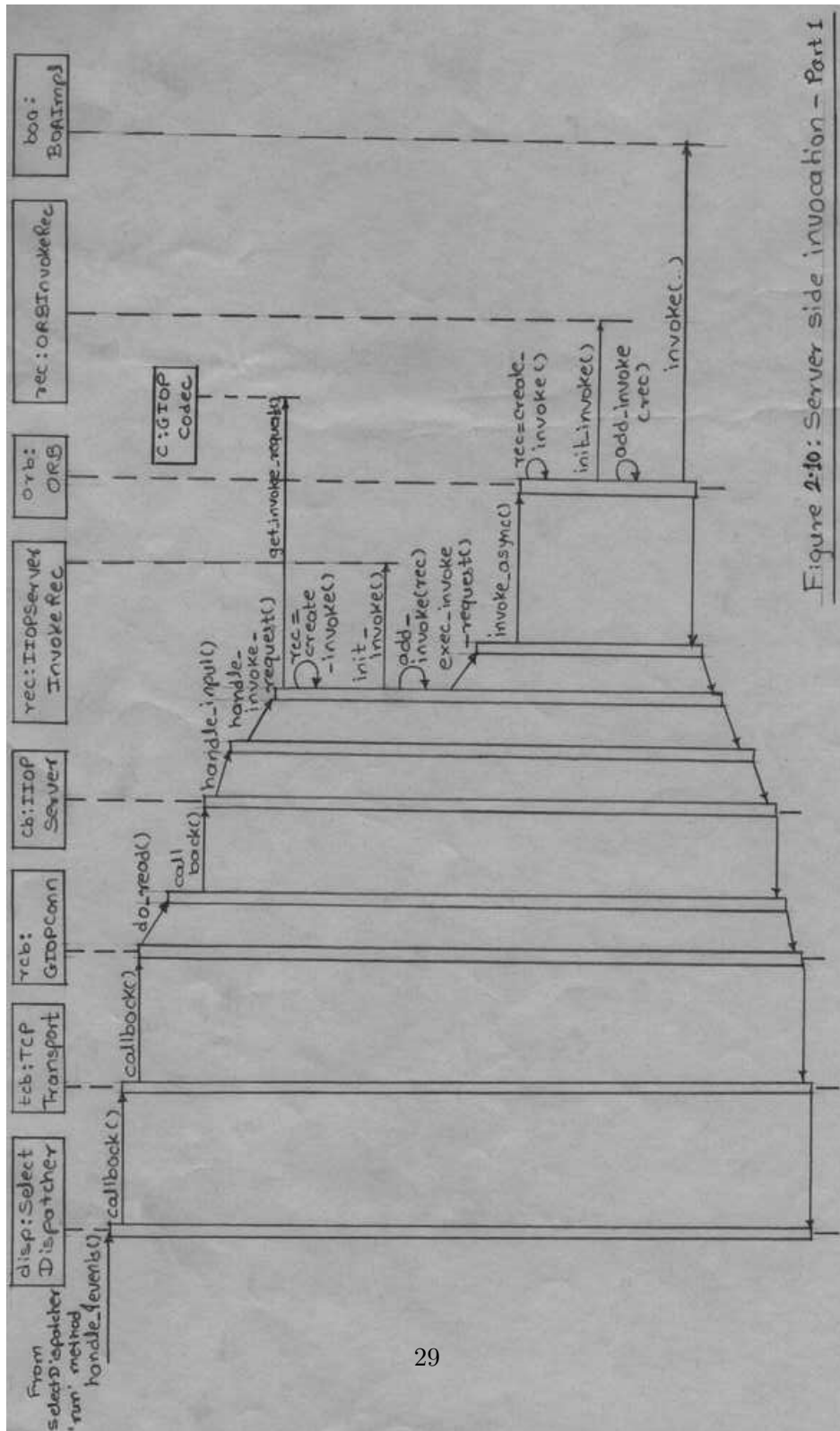


Figure 2.10: Server side invocation - Part 1

Figure 2.10: Server side invocation - 1

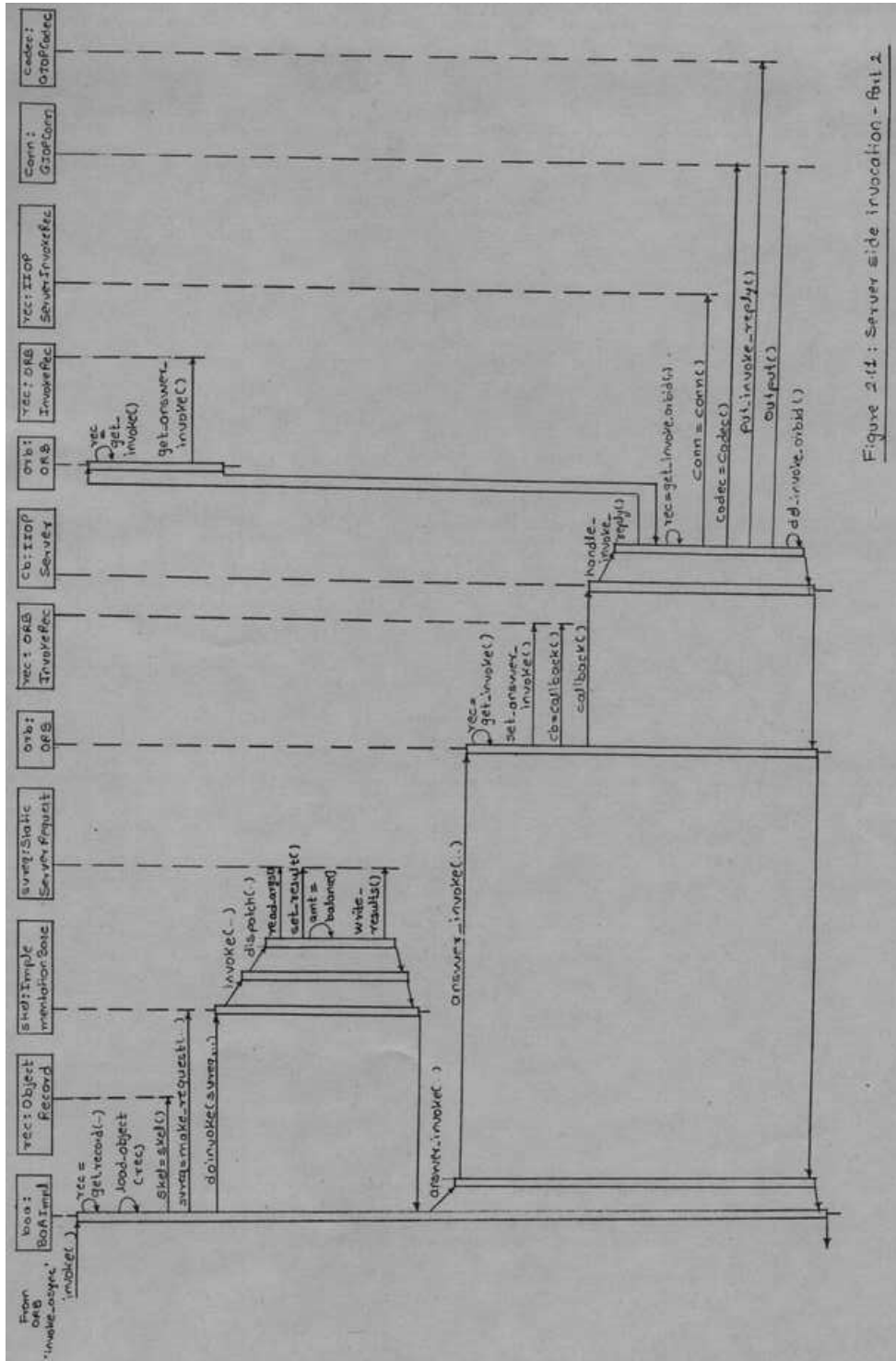


Figure 2.11: Server side invocation - Part 2

Chapter 3

MICO Filter Object Framework Design

After understanding the static and dynamic aspects of MICO architecture, we move on to designing the framework for filtered delivery model called *filter object framework*. Detailed discussion of the design issues involved in embedding the filter object framework in MICO and the design alternatives considered by us based on MICO's implementation model appears in [8]. In this chapter, we compare different design alternatives, evaluate their pros and cons and select one of the alternatives for the framework implementation. In the next chapter, we describe behavioral aspect of the selected design in detail.

3.1 Comparison

The comparison between the design alternatives can be abstracted in a feature matrix shown in Table 3.1. Choice 1, 2 and 3 refer to position of *client-filter* mappings in separate CORBA object providing mapping service, in BOA daemon (micod) and in filter client object respectively.

From the feature matrix, we observe that all three choices are capable of

Design Features		Choice 1	Choice 2	Choice 3
Basic actions		Yes	Yes	Yes
Modularity		Yes	Yes	Yes
Transparency	Client-unaware	Yes	Yes	Yes
	Server-unaware	Yes	Yes	Yes
Selective filtering		Yes	Yes	Yes
Extended properties	Dynamic	Yes	Yes	Yes
	Grouping	Yes	Yes	Yes
	Layering	Yes	Yes	Yes
Overheads				
Plug/Unplug		High	Low	High
Method invocation	No filters	High	High	Low
Method invocation	Plugged filters	High	High	High

Table 3.1: Feature matrix

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.

For 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.

For choice 3, where mappings are stored in filter client, plug/unplug overhead is higher than that in choice 2. This overhead increases since sending these beta messages requires obtaining a filter client 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 filter client objects with plugged filters results in higher overheads because of method routing to filters.

From the above discussion, it is clear that choice 1 is unacceptable. In case of choice 2, though its plug/unplug overheads are low, we rejected it since method invocations are more frequent than plug/unplug beta messages. This leaves us with choice 3, which has higher plug/unplug and invocation (plugged filters) overheads. But this choice does not penalize normal method invocations (no plugged filters). And since method invocation frequency is higher than plug/unplug frequency, choice 3 better suits the design considerations and was selected for implementation of the filter object framework.

Chapter 4

Filter Object Framework - Dynamic Model

After finalizing on the filter object framework design, we discuss the selected design in detail. In this chapter, we present behavioral view (dynamic model) of our design. The architectural view (static model) is described in [8]. 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

Beta messages are special messages, which must be handled by the filter object framework. Although these are special messages, they follow normal method invocation sequence as specified in Section 2.5. All beta messages are handled by BOA in its `invoke` method. Currently there are three different categories of beta messages.

4.1.1 Upfilter/Downfilter beta messages

Each filter object interface consists of *Up* and *Down* filter methods. Up methods filter corresponding client methods whereas Down methods filter return results. Multiple Up and Down filter 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 Up and Down filter methods. These mappings can be manipulated by sending *upfilter* or *downfilter* beta messages. Each beta message takes client method name and corresponding Up or Down filter 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.

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.

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.

4.1.2 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.

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.

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.

4.1.3 Plug/Unplug beta messages

These beta messages allow plugging and unplugging of filters from their clients. ORB provides a public interface viz. *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.

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.

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

Normal static method invocation sequence in MICO is discussed in Section 2.5. Here, we discuss modifications made to method invocation sequence in

order to incorporate the filter object framework. The interaction diagram in Figure 4.1 shows only the modified invocation sequence.

First important modification to invocation sequence is creation of a `FilterServerRequest` instead of `StaticServerRequest` in `make_request()` method. The responsibilities of server-side filter request object include method name translation and performing *Up* and *Down* filtering. To satisfy these responsibilities, it overrides three methods viz. `op_name()`, `read_args()` and `write_results()`.

4.2.1 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.

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,
 - 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.
 - If intercepted method call is *Up* call, search the Up filter methods corresponding to intercepted client method.
 - If intercepted method call is *Down* call, search the Down filter methods corresponding to intercepted client method.
 - If one of the Up/Down filter methods is enabled, set the invocation status as normal filter client invocation and return enabled method name.

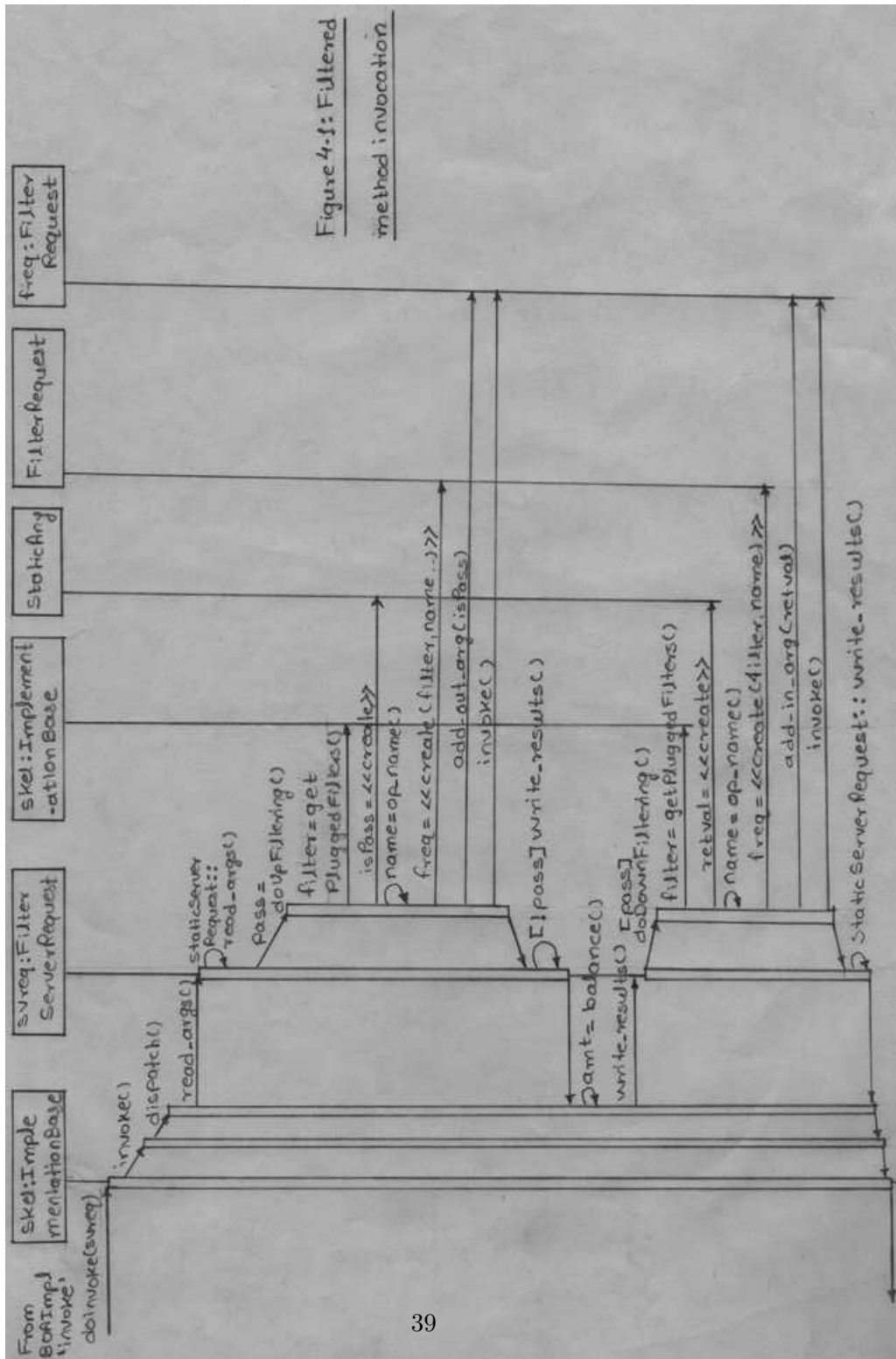


Figure 4.1: Filtered method invocation

Figure 4.1: Filtered method invocation

- 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.

4.2.2 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.

1. Read the argument data. If unsuccessful, set read status as false and return.
2. Perform *Up* filtering setting method *pass* status.
 - If the invocation status is *bad filter client invocation*, set the pass status as false.
 - If there are no filters plugged to current object, set the pass status as true.
 - 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.
 - 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()`.

4.2.3 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.

1. Perform Down filtering if method *pass* status is true.
 - If current method has *void* return type, do nothing.
 - If the invocation status is *bad filter client invocation*, do nothing.
 - If there are no filters plugged to current object, do nothing.
 - 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 Up/Down filter 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`.

Chapter 5

Filter Object Support in MICO

After presenting behavioral view of the filter object framework, we look at the framework facilities that allow it to satisfy both essential and extended properties of the filter objects. First, we discuss specification of the filter relationship using our framework. In the following sections, we present details of facilities that allow our framework to satisfy filter object properties. We also discuss performance implications of the filter object framework.

The filter object framework includes three utilities that help in development using the framework. The `fidlgen` utility generates a sample filter IDL from filter client IDL. The post-processing of compiled filter IDL is done by `filtergen`. The `filterconf` utility allows sending beta messages using configuration files called *beta* files. For more information on these utilities and their use in the development process, refer [8].

5.1 Filter relationship

The filtered delivery model is based on the *filter relationship* between classes. Filter relationship is the ability given by filter client object to filter object

to intercept, manipulate, pass or bounce the message sent to it. In our framework, filter relationship is specified by deriving a filter interface from filter client interface using *fdlgen* utility. For example, Figure 5.1 shows the filter interface derived from the corresponding client interface given in Figure 2.1. Moreover any object, which wants to act as a filter must inherit from `Filter` class. This class provides methods that help in satisfying essential as well as extended filter properties.

```
interface AccountFilter {
    void depositUp(inout unsigned long amount);
    void withdrawUp(inout unsigned long amount);
    long balanceUp();
    long balanceDown(in long result);
};
```

Figure 5.1: AccountFilter interface

5.2 Essential properties of filter objects

1. *Basic filtering actions* -

- **Interception:** Filter objects are specified to intercept messages arriving at an ordinary object called a *filter-client*. Filter object framework allows interception of filter client message by forwarding it first to plugged filters before local handling, as explained in Section 4.2.2.
- **Manipulation:** A filter may manipulate the client messages. The client message manipulation is made possible by specifying all arguments of *Up* methods as *inout*, as shown in Figure 5.1.
- **Pass/Bounce:** A filter may *pass* or *bounce* the client message. Pass specifies forwarding the message to the client, while bounce

is a return performed by the filter. A filter object can specify message *pass* or *bounce* by using `setPass()` or `setBounce()` methods of the `Filter` class.

2. *Modularity*: Specification of the filter object is separate from the client specification. In our framework, a filter object is a full fledged CORBA object and doesn't interfere with client specification. Hence it doesn't break the encapsulation of its client. Similarly a filter client doesn't break the encapsulation of its filter.
3. *Transparency*: Message sender as well as filter client itself are unaware of existence of filter objects. This preserves direct call semantics and requires no source code change when filters are added, removed or replaced. Transparency is achieved by providing beta message handling (Section 4.1) and modified method invocation sequence (Section 4.2).
4. *Selective filtering*: A filter can intercept messages selectively. Some methods may be filtered, while some may remain unfiltered. The `Filter` class provides `enable()` and `disable()` methods, which allow enabling or disabling particular filter object method. These methods along with Up/Down filter beta messages facilitate selective filtering.

5.3 Extended properties of filter objects

1. *Dynamic Filtering*:
 - Plug/Unplug: Filters can be changed for a client over its lifetime. The ORB interface specifies `plug()` and `unplug` methods. A filter client object maintains a list of plugged filters and these methods allow changing filters for a client by manipulating this list.
 - Filtering method change: Every filter object maintains mapping of client method and corresponding *Up* and *Down* filter methods. The `upfilter` and `downfilter` methods of the `Filter` class manipulates this mapping. These methods along with `enable()` and

`disable()` allow changing individual member functions within a filter object that filter corresponding counterparts in the client object at runtime.

2. *Group filtering*: This allows multiple filter clients to be served by a single filter. A filter object may intercept messages sent to number of its client objects that are instances of the same client class. In the filter object framework, group filtering is done by simply plugging same filter object to multiple clients.
3. *Layered Filtering*: With layered filtering, multilevel filters can be designed by specifying filters to filters and are used for multilevel message processing. The filter object framework allows layered filtering either by plugging filters to filters or by plugging multiple filters to same client.

5.4 Performance

The performance implications of using the filtered delivery model are discussed in this section. The test setup included machines (P4, 256MB RAM), which are part of 100Mbps LAN. The machines participating in the test setup were running MICO (version 2.3.4) with integrated filter object framework on Linux. Note that observations made here are applicable to test setup described above and are likely to vary with changing setup.

We start with presenting in Table 5.1, time required to make direct calls to local and remote servers without any plugged filters. These timings are used to judge the overheads of the filter object framework. Here the term *local* means that the server is running on the same machine as test program and the term *remote* means that it is running on different machine.

Next, we present time required to send beta messages. Table 5.2 shows time to send beta messages to local and remote filter clients and filters. Fi-

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

Table 5.1: Direct call

nally, we show timings for *filtered* method invocation with one filter plugged to filter client. Table 5.3 presents time required to make a filtered call with *Up/Down* filtering enabled and disabled. Time measurements are shown for four combinations of clients and filters viz. Local Client/Local Filter, Local Client/Remote Filter, Remote Client/Local Filter and Remote Client/Remote Filter.

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

Table 5.2: 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 5.3: Filtered call

By comparing direct calls (Table 5.1) and *Passed* filtered call with *Up/Down* filtering enabled (Table 5.3), it can be observed that later incurs approximately 2.5 times overheads over direct call. Similarly overheads of *Bounced* filtered call with *Up/Down* filtering enabled over direct call are approximately 1.6 times. With *Up/Down* filtering disabled, overheads are approximately twice that of 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.

Chapter 6

Concluding Remarks and Future Work

Evolution of object-oriented systems to meet the changing requirements is possible using transparent filter objects. Filtering frameworks based on filtered delivery model (Section 1.1) help in system evolution by enhancing reusability. Such a filtering framework was designed and implemented for MICO, an open-source CORBA implementation. The filter object framework supports all essential as well as extended filter properties (Section 5). Filtering framework is implemented by modifying the existing implementation of MICO version 2.3.4, which is compliant with OMG-CORBA 2.2 standard. The framework also provides utilities to support the development process. These include `fidlgen`, to generate sample filter IDL from filter client IDL, `filtergen`, for post-processing compiled filter IDL, and `filterconf` for sending *beta* messages using configuration files called *beta* files.

In this report, we gave overview of filtered delivery model of message delivery in object-oriented systems and CORBA architecture. We looked into the behavioral aspect of the MICO implementation - dynamic model. The static model is explained in [8]. After understanding the implementation,

we could bring out various design options, which are discussed in [8]. We rigorously compared these choices, discussing their pros and cons and ultimately a robust design was evolved. The dynamic implementation model of the filter object framework was documented in detail. The framework facilities supporting filter object properties and performance implications of the framework were also discussed.

Possible extensions to this work:

- The limitations expressed in [8] can be removed. The current implementation can be extended to cover all possible activation policies and object adapters like POA. Interception of dynamic method invocations should be possible. Exception handling needs to be introduced to check for errors like incorrect filter interface.
- The MICO filter object framework provides server-side filtering. The implementation can be extended to include client-side filtering as well.
- Filter objects have several applications in distributed systems. More work needs to be done to systematize the design process of filter objects. Applications of filter objects for evolving object-oriented systems needs to be studied more closely and over a large set of examples. This might lead to interesting design patterns that use filter objects.

Bibliography

- [1] *MICO Is CORBA: An Open Source CORBA 2.3 Implementation (Documentation)*, 2.3.4 edition, 2000.
- [2] Ralph Johnson E. Gamma, Richard Helm and John Vlissides. *Design Patterns*. Addison Wesley, August 1994.
- [3] Tony Mason J. Levin and Doug Brown. *lex and yacc*. O'Reilly and Associates, 1990.
- [4] Rushikesh K. Joshi. Modeling with filter objects in distributed systems. *Proceedings of 2nd Workshop on Engineering Distributed Objects (EDO 2000)*, 1999:176–181, November 2000.
- [5] Rushikesh K. Joshi. Filter configurations for transparent interactions in distributed object systems. *Journal of Object Oriented Programming*, pages 12–17, June 2001.
- [6] L. Bergmans M. Aksit, J. Bosch, editor. *Abstracting Object Interactions using Composition Filters*, volume 791 of *Proceedings of the ECOOP'93 Workshop on Object-based Distributed Programming*, Springer-Verlag, 1994.
- [7] Maureen Rita Mascarenhas and Rushikesh K. Joshi. Filter objects for java. Technical report, Indian Institute of Technology, Bombay, January 2001.

- [8] Pranav S. Nabar. Filter object framework for micro - static model. Master's thesis, KReSIT, Indian Institute of Technology, Bombay, January 2002.
- [9] Object Management Group (OMG). *The Common Object Request Broker: Architecture and Specification*, 2.4.1 edition, November 2000.
- [10] G. Srirami Reddy and Rushikesh K. Joshi. Filter objects for distributed object systems. *Journal of Object Oriented Programming*, pages 12–17, January 2001.
- [11] N. Vivekananda Rushikesh K. Joshi and D. Janaki Ram. Message filters for object-oriented systems. *Software - Practice And Experience*, 27(6):677–699, June 1997.