Filter Object Framework for MICO

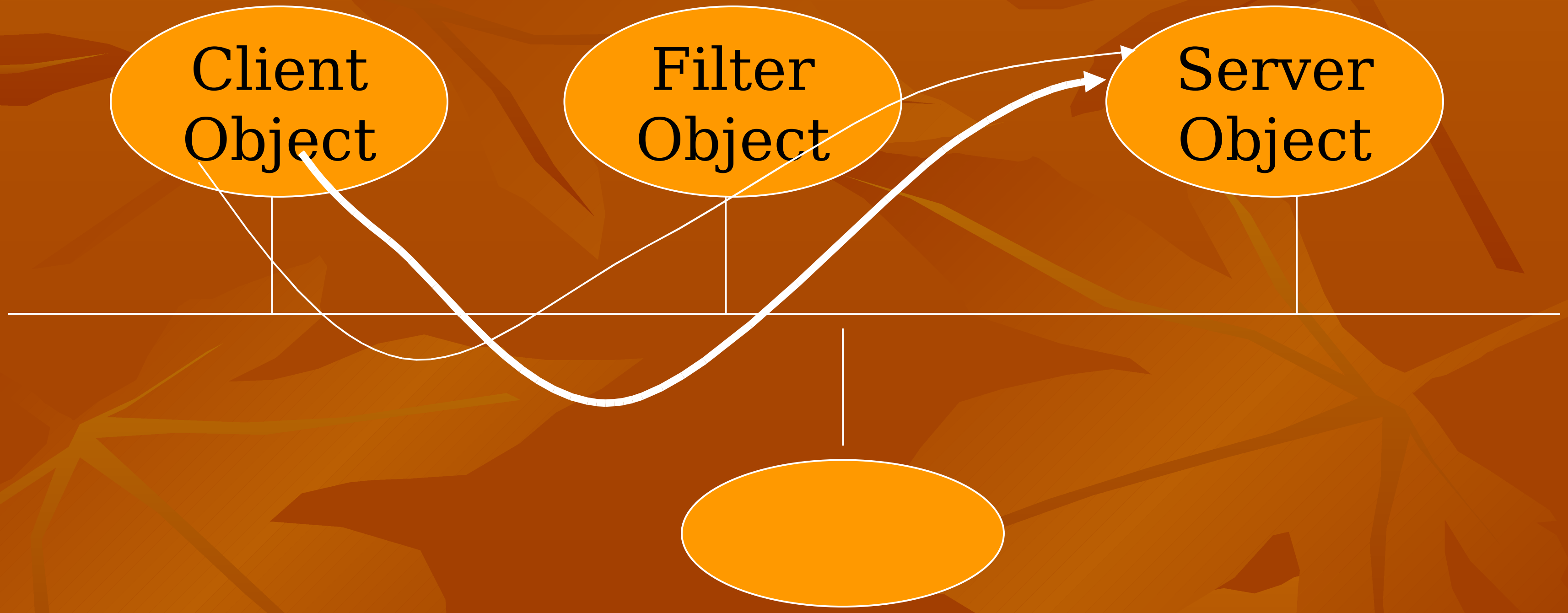
Rushikesh K Joshi
Department of Computer Science and Engineering

Indian Institute of Technology
Bombay, India

Filtered Delivery Model

- Separation of message control from Message Processing
- Filter Objects Model
  - C++/JAVA/MICO user level
  - *Filter Object Aware Environment*
- *Modularity and First class Filters*
- *Dynamic Pluggability*

A Filtering Scenario



Why Filter Objects as First Class Objects?

- All benefits of full-fledged objects

+ Special abilities to filter method invocations

- Separation of Concerns
- Parallel Development
- Runtime capabilities
- Aspect Modeling / Way of Composing Aspects
- Towards Transparent Evolution

Previous work in Filter Objects

- Filters for:
  - C++
  - Java Programming Language (TJF)
- A Distributed Filter Object Implementation on Aspect-J
- User-level Filter Objects for MICO

- Related Work: Aspect Modeling, Composition Filters, Context Relations, CORBA Interceptors

Properties of Filters

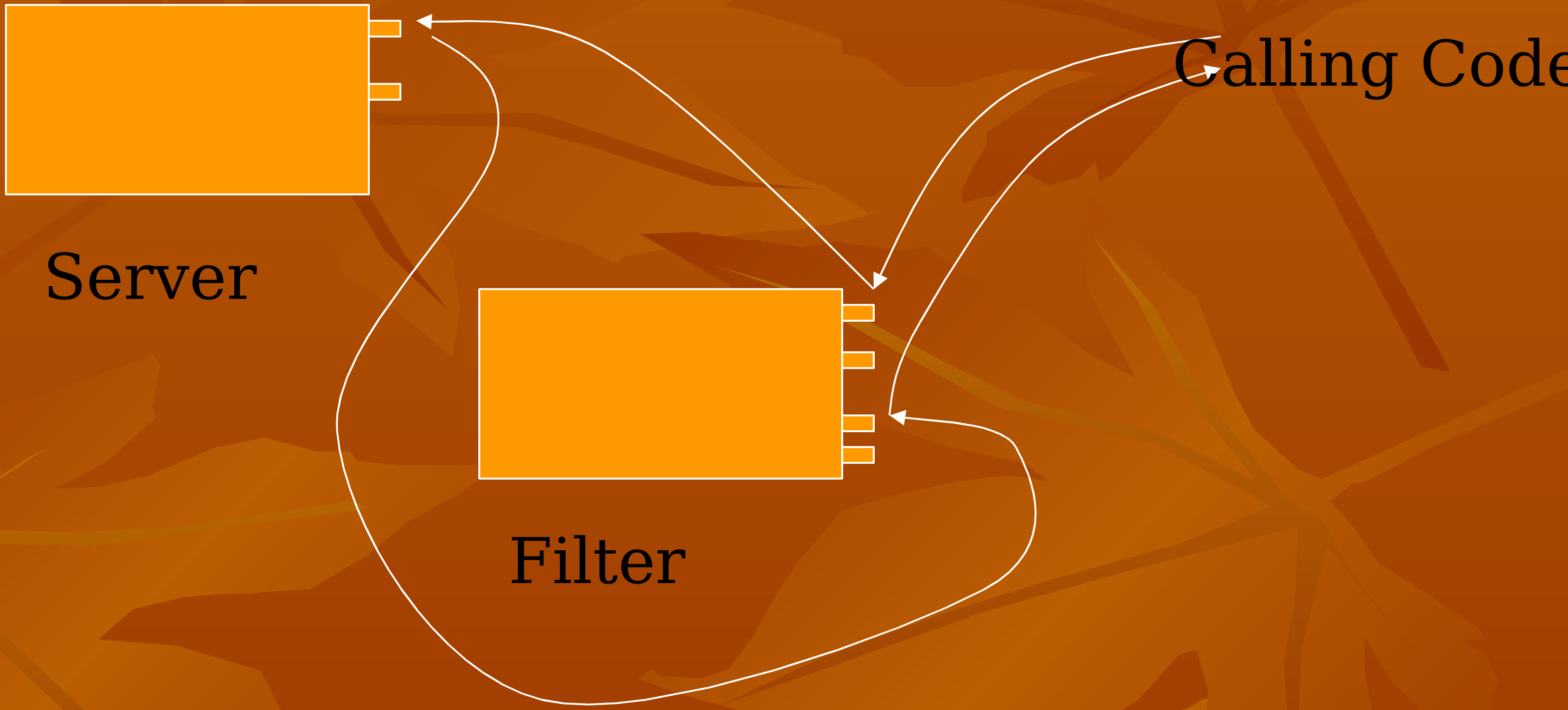
- Basic filtering actions: up/down filtering at method level
- Modularity: Filter specification & implementation separate from the server's
- First-class-ness: Filter objects are first class, full-fledged CORBA Objects
- Transparency: w.r.t. both client and server ends
- Selective Filtering: enable/disable filter member functions at runtime
- Group Filtering: one-to-many
- Dynamic binding: plug/unplug filter objects
- Layered Filtering: Multiple levels of filters

The Development Process: *Specifying Filter Objects*

- Build a Filter IDL
  - Manually
  - By fidlgen utility

- Filter IDL specification ..
  - For every server method:
    - At least one upfiler method
    - At least one downfilter method if server method returns a non-void value
  - Arguments to an upfilter method are *inout*
  - Names of Filter Methods can be different from their corresponding server methods

A Pictorial View



The Development Process: *Implementing Filter Objects*

- Compile the Filter IDL
  - using MICO IDL compiler
- Run a filtergen utility
  - Modifies inheritance for the generated Filter class
  - Filter object inherits from CORBA::Filter instead of CORBA::Object
  - CORBA::Filter is CORBA::Object
- Implement Filter Object as a CORBA object

An Example Filter Implementation

Dictionary.idl

```
Interface Dictionary
    Wpair lookup (in string word);
};
```

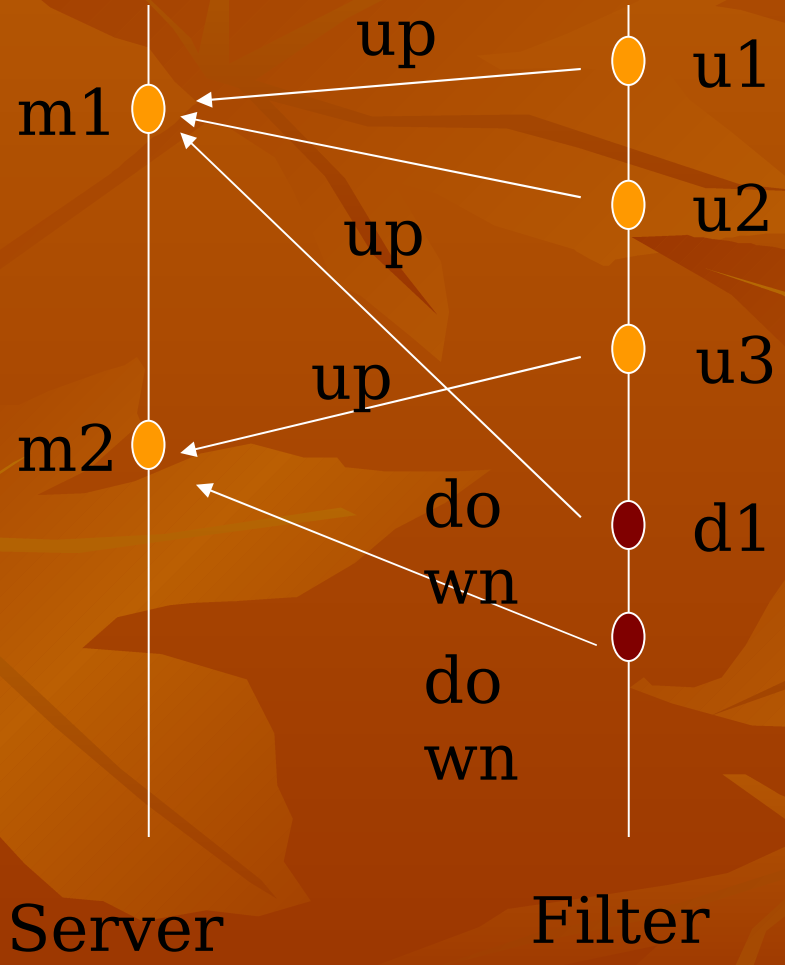
Cache Implementation

```
class DictionaryFilter_impl : virtual public
    DictionaryFilter_skel {

    lookup_up(){..};
    lookup_down(){..};

};
```

The Development Process: *Working with Catalysts*

- Mapping Method Names
  - map filter IDL implementations → up/down filtering members for corresponding server methods (names may be different)
  - Through upfilter() and downfilter() mapping methods in class CORBA::Filter
  - In absence of these mappings, invocations are directly delivered
  - Performed after creation of filter instances

A Pictorial View of a Mapping

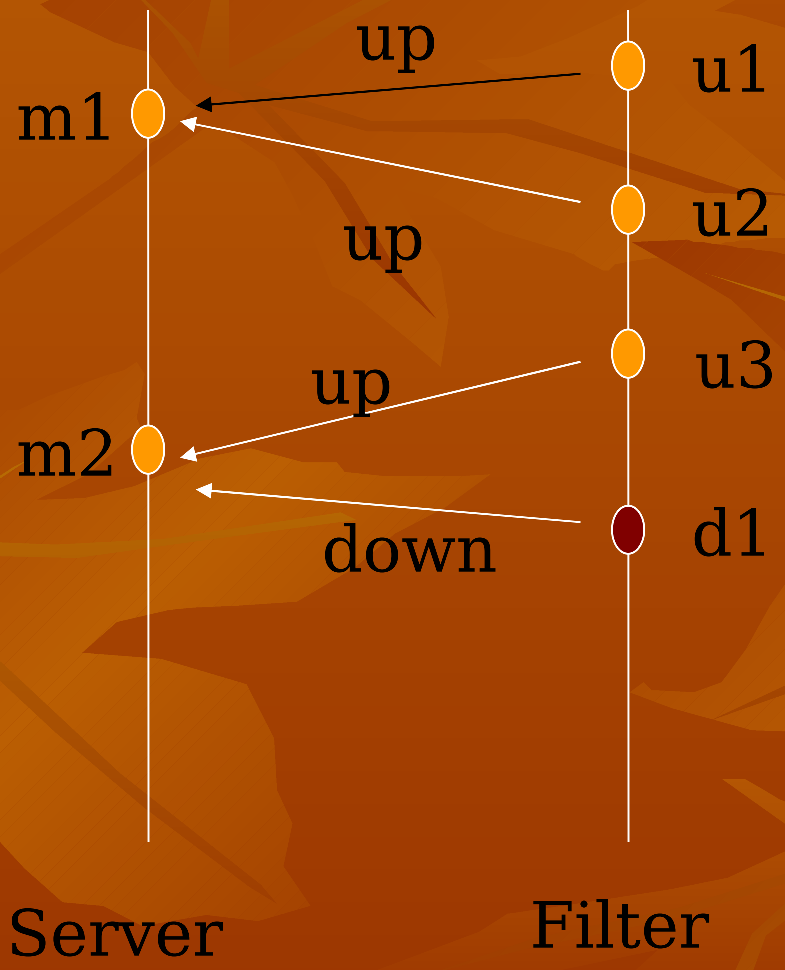


The Development Process: *Working with Catalysts*

- Dynamic enabling of filter methods
  - Through enable() /disable () on class CORBA::Filter
- Plug and Unplug
  - Obtain a local ORB reference
  - Through plug(), unplug() of CORBA::ORB class
    - Server and filter references passed as arguments

filterconf utility provided to assist catalyst development

A Runtime View



Design Requirements

- Support all filter properties
- Transparency
- System evolution with ideally NO change in existing code
- Keeping overheads low
- Filter Objects as CORBA Objects
- Control over filter methods

Design Alternatives

- Design Considerations
  - Location of mappings between the server and filter objects
  - Location of intercepting the call
- Design Choices
  - Mappings as CORBA service
  - Mappings in the micod
  - Mappings managed at the server-side

Interfaces for Filter developer

- Class `CORBA::ORB`
  - Two new methods

- Class `CORBA::Filter`
  - Superclass for all filter objects

CORBA::ORB class

- Plugging Filter Objects onto Server Objects
  - `Plug`

- Unplugging Filter Objects
  - `unplug`

CORBA::Filter Class

- Mapping methods
  - `upfilter` and `downfilter`
- Enabling methods
  - `enable` and `disable`
- Setting “pass” and “bounce” actions
  - `setPass` and `setBounce`

Managing Server→Filter Mappings on the Server-side

- Class CORBA::Object
  - superclass of every CORBA Object
- private `plug` and `unplug` interfaces
  - for adding and deleting server→filter mappings

Carrying Filter Requests

- Intercepted invocations are routed to filter objects
- Two specialist classes
  - Class `FilterRequest inherits StaticRequest`
  - Class `FilterServerRequest inherits StaticServerRequest`
    - Method name translation: opname() is overriden
    - Upward filtering: readargs() is overriden
      - iterate through plugged up-filters
      - agrs are changed to inout
      - pass bounce status is checked
    - Downward filtering: writeresults() is overrriden
      - Iterate through plugged down-filters
      - Arg is changed to inout

Deactivation and Reactivation

- Objects may shutdown and reactivate
- BOA’s save_object() method is modified
  - Save filter framework related information
- Upon reactivation, filter framework is restored when a request is made

Assessment of the Filter Object Framework

- Enhancements to 3 classes in MICO static model and addition of 7 classes
- Advantages
  - First class dynamically pluggable Filter Objects
  - Separate Development of Filter Objects
  - In most cases, NO change in existing code required for system evolution
  - All filter properties supported
  - Multiple methods can filter single server method
  - Utilities for working with catalysts
- Limitations
  - Only intercepts static invocations on servers following the shared activation policy through the BOA.
  - Some mappings maintained at the server side
  - Exceptions are not handled

Summary of Enhancements

- Class ORB
- Class Object
- Class StaticServerRequest

- Additional public methods – plug and unplug
- Maintains mappings
- Additional private methods – plug and unplug
- Modified methods op_name, read_args, and write_results as virtual

Summary of Additions

- `Class Filter`
- `Class FilterRequest`
- `Class FilterServerRequest`
- `Class BetaMessage`

- Basic Filter Interface
- Specializes class ServerRequest for filtering at the client-side
- Specializes class StaticServerRequest at the server-side
- Abstract class for handling special messages

Summary of Additions

- Class `PlugUnplugMessage`
- Class `EnableDisableMessage`
- Class `UpDownFilterMessage`

- Concrete implementation of plug and unplug beta messages
- Concrete implementation of enable and disable messages
- Concrete implementation of upfilter and downfilter beta messages