THE FilterManager : MAINTAINING CONSTRAINTS AMONG PROCESSES

Juliana Khing Hua Lim
B.Sc., National University of Singapore, 1983

A thesis submitted to the faculty
of the Oregon Graduate Center
in partial fulfillment of the
requirements for the degree
Master of Science
in
Computer Science

May 1989

The thesis "The FilterManager: Maintaining Constraints Among Processes" by Juliana K.H. Lim has been examined and approved by the following Examination Committee:

David Maier
Professor, Oregon Graduate Center
Thesis advisor

Robert G. Babb II
Associate Professor, Oregon Graduate Center

Alan Borning
Associate Professor, University of Washington

ACKNOWLEDGEMENTS

It has been my privilege to work on the FilterManager under the direction of Dr. David Maier and benefit from his helpful suggestions. Allan Schiffman provided insight on the socket interface for Smalltalk so that the two implementations could work together. John Maloney and his group provided ThingLabII for testing. I would like to thank Dr. Richard Kieburtz and Dr. Ronald Cole for accomodating my use of goedel and groucho. I appreciate the patience of Boris, Dwight, Sherri and Yeshwant while I used Smalltalk on goedel and groucho.

I have also enjoyed strong support and encouragement from my family, although half the world away, and many friends both near and far.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

The FilterManager provides application processes with a way to establish and maintain some invariant relationships among their states.

A relationship in the states of two application processes is identified by a constraint that must be satisfied between an object in each process. This constraint is represented by a *filter* (hence the name "FilterManager") that connects the two objects and is the mechanism by which they are kept consistent. A filter works in both directions: when either of the objects changes, the other is updated correspondingly.

The FilterManager manages the collection of filters as a network, allowing a dynamic set of applications and relationships among them without requiring the applications to be constantly aware of their environment and relationships.

This thesis reports on the implementation of the FilterManager in ParcPlace Smalltalk Versions 2.3 and 2.4. The implementation in Version 2.3 is on a Sun-3 Workstation with the SunOS 3.5 operating system, and that in Version 2.4 on a Sun-4 Workstation with the SunOS 4.0 operating system. Each implementation maintains filters among applications written in the same version of Smalltalk and on the same machine. In addition, the implementation in Version 2.3 also manages filters among applications in both versions and spanning both machine types.

CHAPTER 1

INTRODUCTION

1.1. Objective

The objective of the FilterManager project is to investigate how a network of constraints can be used to maintain some consistency in the states of a set of application processes. Since the state of a process is represented by the objects in its object space, a constraint established between two objects, each existing in the context of a different process, represents the way the two processes are related. These two processes are otherwise independent, and may even be executing in different environments.

1.2. Constraints Between Processes

In an interactive user interface, when multiple views of a program object are simultaneously presented to a user, there needs to be some consistency among the object and all views of it. When a user changes some part of the object in either the program, or one of its views, the change should be reflected in every representation of the object.

Along the same lines, it would be useful to allow requirement of consistency between two objects existing in the context of different

application processes. For example, an object in a GemStone object-oriented database [Pur87] could be related to an object in an interface built with the FilterBrowser [Ege87, EMB87], a tool for the designer of a user-interface. Although the GemStone database and the FilterBrowser execute independently, a constraint established between an object in each of their object spaces identifies how their states are related. When either of the two objects is updated, the other object should have a corresponding update, to maintain a consistency in the relationship between the two processes.

This notion can then be extended to a network of such constraints involving a set of objects in various application processes, with each process being constrained in some part of its object space.

1.3. Terminology

A *filter* is the representation of a bidirectional constraint that connects two objects and describes how they are related. In the model of the FilterBrowser, a filter relates a *source* object in an application to a *view* object that is the graphical representation of the application object (Fig. 1.1). The source and view objects that the FilterBrowser relates must belong to the same invocation of a single application process.

The filter serves to maintain the consistency of the two objects it connects when either one of them changes, and to perform various transformations between the objects. It can be a composite filter, built up

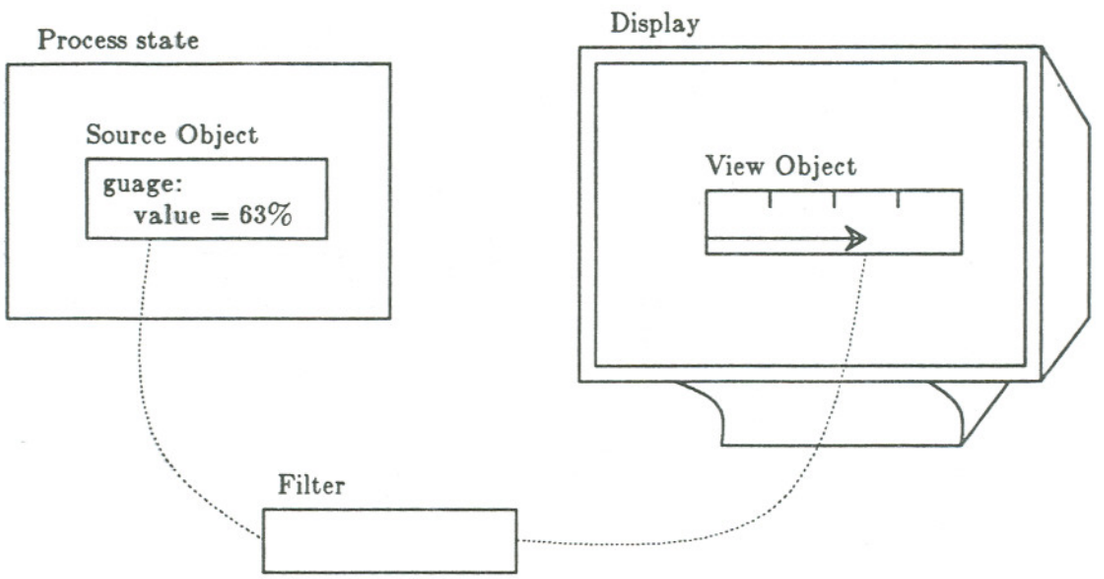


Fig. 1.1 A filter: representing constraints between source and view objects of an application.

from simpler filters, each of which represents a constraint to be maintained between parts of the two objects.

In contrast, a *filter* in the model of the FilterManager represents a constraint between two objects in different application processes (Fig. 1.2). The filter is no longer in the closed environment of a single application but in an open environment where the application processes are diverse and

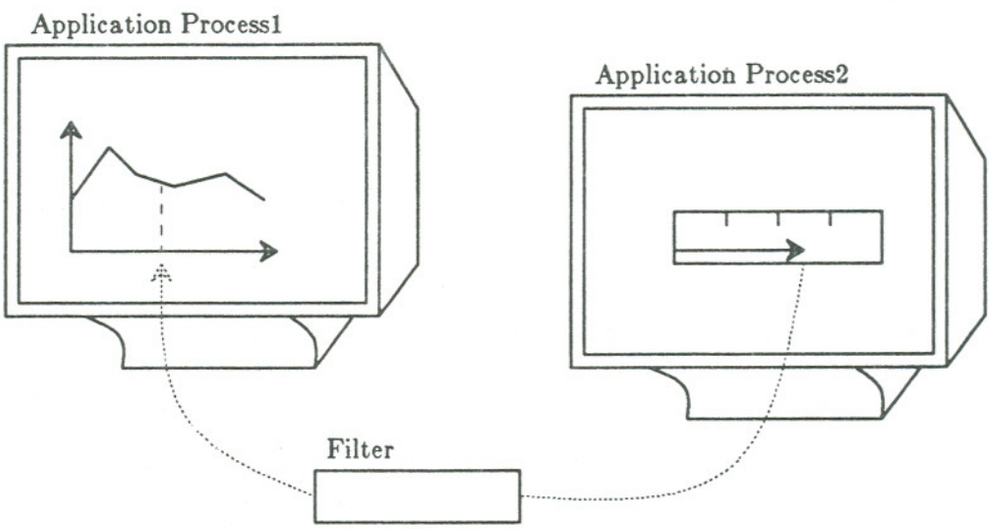


Fig. 1.2 A filter in an open environment: representing constraints between two objects in different processes.

dynamically changing. The *source* object of a filter refers to the object that initiates a change and triggers the filter mechanism to maintain the constraint; the *target* object of a filter is the receiver of the change. Since filters work in both directions, the source object of a filter is not limited to be at a specific end of the filter, and the object at either end becomes the source object when it changes. The ends of a filter are typed, and a filter performs a specific transformation between the two objects it relates.

A *process* is an application executing in its own environment and having its own object space. The term process will mean here an application process, or a job in an operating system, rather than the Smalltalk class Process, unless otherwise identified.

1.4. The FilterManager Approach

The moment an object that is related to objects in other application processes changes is a good time to check on, and to enforce, the consistency of its relationships with those other objects. Maintaining the consistency of object relationships at the time of change ensures that each object in the network is at most one step away from consistency in its relationships.

An application process having constraints on its objects must contain a number of routines for the FilterManager to support the filters representing those constraints. An application process, however, does not contain the entire support for filters because each process would then need to be aware of the states of the processes it is related to.

In order for filters to be established between objects in different processes, the FilterManager must first be executing and each process needs to identify itself to the FilterManager and the objects in its context that may be constrained. Filters can then be established between these objects at the initiative of any of the processes known to the FilterManager.

The FilterManager has been implemented to manage filters spanning Smalltalk applications written in Versions 2.3 and 2.4 of ParcPlace Smalltalk on Sun-3 and Sun-4 Workstations with the SunOS 3.5 and SunOS 4.0 operating systems, respectively.

CHAPTER 2

PROBLEM DESCRIPTION

2.1. The Ultimate FilterManager

The initial phase of exploring filters as a mechanism to build interfaces began with the FilterBrowser [Ege87, EMB87], which is constructed based on ThingLab [Bor81]. In it, filters relate an application object, the source of a filter, to its graphical representation, the view of the filter. Since ThingLab needs a global knowledge of the filters and the objects they connect, however, this is a closed system where filters are limited to connecting objects within the same Smalltalk process (in fact, within the same ThingLab invocation).

When interfaces must span processes and machines, filters need to be established outside of a closed environment. In an open and dynamic system where each application exists as a process in its own environment and with its own object space, a filter is used to relate two application processes rather than just two objects in one application. Maintaining the consistency of the filter constraints in an open environment becomes more complicated because little or no global knowledge can be assumed about the application processes and their objects.

Consider the following scenario: three different applications are in progress, each having an environment of its own (Fig. 2.1). A Smalltalk

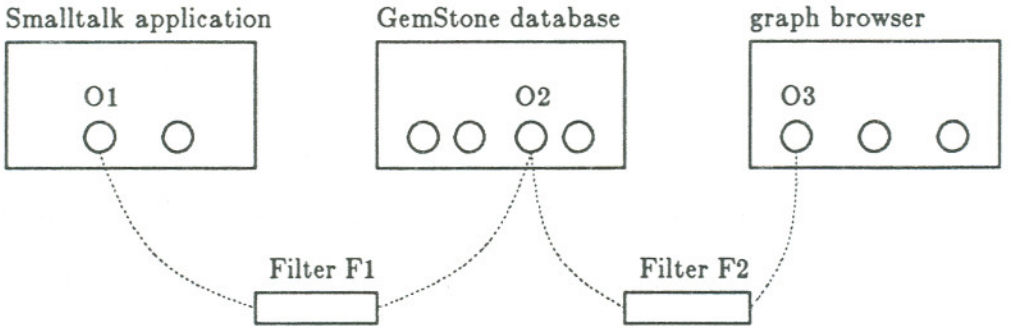


Fig. 2.1 An example scenario.

application with an interactive display created with the FilterBrowser, a GemStone database, and a graph browser are executing as processes on different machines. Some filters are established among some of the objects managed by these different applications. Suppose, for example, that the user at the interactive display of the Smalltalk application is experimenting with an object O1, which is related by a filter F1 to an object O2 in the GemStone database. The user at the graph browser is displaying a graph object O3 which happens to be a part of some statistics generated from GemStone database objects, O2 being among them. The objects O2 and O3 are thus connected by a filter F2. If the user at the display of the Smalltalk application updates object O1, then both the objects O2 in the GemStone

database and O3 in the graph browser now potentially require updates for consistency, as imposed by filters F1 and F2.

2.2. An Incremental Approach

The problem can be viewed as a progression from the FilterBrowser, which establishes and maintains a network of constraints between objects within a single application process, to the FilterManager of the future, which will manage a network of filters spanning processes, environments and machines.

In the current version of the FilterManager, a model is developed and implemented to extend the FilterBrowser to handle filters on objects in different processes having similar environments: multiple Smalltalk images. The current implementation is for Smalltalk processes, possibly residing on different machines, and is restricted to equality filters, where the objects of a filter are constrained to be identical in structure and equal in value. Equality filters are sufficient for processes executing in similar environments since any filter F can be redefined as the equality filter E followed (or preceded) by F, where the equality filter E spans environments, but F is entirely within one environment.

To extend this model to one in which the processes execute in dissimilar environments, such as the GemStone database and the application with a display created with the FilterBrowser as described earlier, other types of

filters would have to be included. Equality filters suffice for two processes with similar environments because the type system is the same. When a filter relates two objects in environments with dissimilar type systems, however, the filter must perform type conversion. Such filters will include a component that describes the transformation of an object from one environment to another.

2.3. Related and Background Work

The FilterBrowser [Ege87, EMB87] provides the designer of a user-interface with a tool that uses constraint-satisfaction to build interfaces. It allows constraints among objects in an interface to be defined graphically. It models the description and maintenance of these constraints using filters, which may be composites of simpler filters. ThingLab [Bor81], on which the FilterBrowser implementation is based, uses constraint-satisfaction techniques to plan and solve constraints defined among objects. Animus [Dui86], an animation toolkit, introduces the dimension of time into ThingLab.

Other graphic tools handle relationships in different ways and for different purposes. One facility provided by the PV [Bro85] system is the dynamic display of the contents of specified data structures of a program as it executes. Incense [Mye83] is a system that displays the data structures of an executing program and allows a programmer to interactively investigate

those data structures. BALSA [Bro88, Bro84] and its descendant systems provide multiple views of a data structure that are simultaneously and dynamically updated with the execution of the animated algorithm, which is a specially modified version of the original program.

The ANTICS [Dio78] system is used as a visual aid to the operation of an animated LISP program. The GARDEN [Rei86] system, a successor to the PECAN [Rei84, Rei83] system, provides a means to simulate the message-passing mechanism of Smalltalk, and the active-values mechanism of access-oriented programming [Ste86, Ste83]. In access-oriented programming, when an active-value is accessed, it triggers a corresponding action.

These models all exhibit a common trait — that of objects having to maintain some relationships among themselves. They serve as toolkits for the design and testing of user-interfaces, algorithms, programs and other systems, within a single environment. Some form of global or shared information is maintained among all the objects and/or processes in the system, and the simultaneous and dynamic updates on related objects are handled through this common information.

The Database Access Manager used by the Humanizer [Gro85], on the other hand, maintains a separate database of information on which processes periodically check to maintain the consistency of the relationships among their objects.

2.4. Limitations of Existing Models

The Smalltalk application, the GemStone database and the graph browser described in the scenario are independent applications in the sense that each one is complete and can exist on its own. Yet, because all three applications happen to be executing simultaneously on some set of objects that are semantically related, some form of consistency among these processes is desirable. Constraints can be established to identify and maintain the required consistency.

Maintaining the constraints by having each process broadcast all changes in any object it manipulates is inefficient because not every change is interesting to the other processes. Process interactions in this context pose a different problem from process interactions in the context of operating systems. An application process needs only to interact with other processes if the object being changed is at one end of any filters; this interaction is limited to those processes whose objects are the targets of those filters.

The models of the various user-interface toolkits mentioned above could perhaps be extended to handle related objects in a predefined set of applications. A database could be used to maintain some global state with which to enforce consistency among the set of applications, and the Model-View-Controller paradigm of Smalltalk could be adopted to handle relationships. Most databases, however, do not have a mechanism to notify clients of changes, and the clients would have to poll the database for changes. An application should not have to track all events in the

environment in order to maintain a state of consistency, as this would result in either a high overhead, or low consistency. Furthermore, concurrency control in a database prevents the changes made by a client to data from being visible until the transaction is completed. Long-running transactions by a client on the database impede rather than enhance consistency.

Another limitation on this approach is that when the environment is to be an open one, with a dynamic and diverse set of application processes, the addition of each new application is complex. The database that maintains the global state must be able to handle any new types of objects used in the new application. It must be provided with information on how to perform object transformations between these new object types and existing types. In addition, if the application has an environment that is different from the environments of all other applications, the database would need to know how to perform type-conversion between the new environment and the existing ones.

2.5. Design Goals

The FilterManager was designed with the objective of managing filters representing the constraints existing among a dynamic set of application processes. Each process participating in a filter network requires only that the FilterManager be accessible from its environment. Since the object relationships among processes have a semantic nature and are thus dynamic

over time and with the environment, a process should not have to be dependent on a static environment to allow its relationships to be maintained.

Since heterogeneity is to be a characteristic of the eventual FilterManager, it is desirable for an application process to require as little modification as possible to participate in a filter network. In particular, the application itself should be able to operate in any of the environments it was originally set up for, without requiring a special modification for each environment or system. The FilterManager should take care of the system-specific, or environment-specific, details of managing filters on the objects of an application process, so that a process can participate in a filter network with little more than the establishment of a filter.

In the ultimate filter system, there would also need to be support for creating new kinds of filters so that the services of the FilterManager can be extended to manage new environments and new kinds of constraints.

CHAPTER 3

THE FilterManager

3.1. Conceptual Model

The conceptual model of the FilterManager describes what an application programmer sees when constraints are established among some objects from a set of application processes, while the implementation model describes the software components of the FilterManager and their roles.

Consider generalizing the functionality of the FilterBrowser [Ege87] so the two objects of a filter need not belong to the same process (Fig. 3.1). In particular, they belong to the object spaces of two different processes that reside in possibly dissimilar environments.

Each process manages and manipulates some objects in its own private object space. A process can choose to make some of its objects known to other processes. A filter can then be established between any two of these objects, each from the object space of a different process, to represent and to maintain a constraint between them. When one of the two connected objects changes, it becomes the source object of the filter, and the other object, the target object, is changed correspondingly to maintain the constraint.

In the conceptual model of the FilterManager, a process does not need to check on the relationships that its objects have with objects outside its

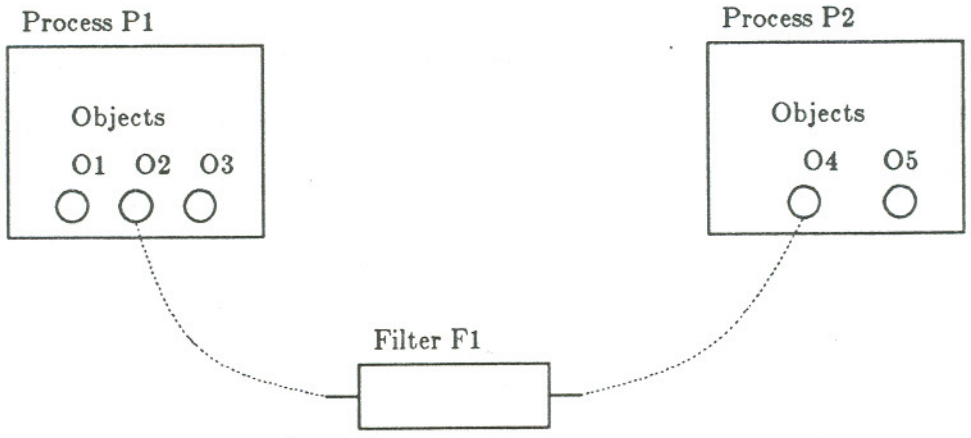


Fig. 3.1 The conceptual model of the FilterManager.

private object space. Instead, a filter describes each of these relationships, and the FilterManager performs constraint-satisfaction on a network of these filters.

This model can be realized as a system of three types of components, the processes whose objects may be related in some way, the filters that describe those relationships, and the FilterManager, which maintains the network of filters.

3.1.1. Processes

Each application process has its own object space representing its state; these objects may have two kinds of relationships (Fig. 3.2). An object may be related to other objects within the process, but such relationships exist only within the context of the process and have no direct bearing on other

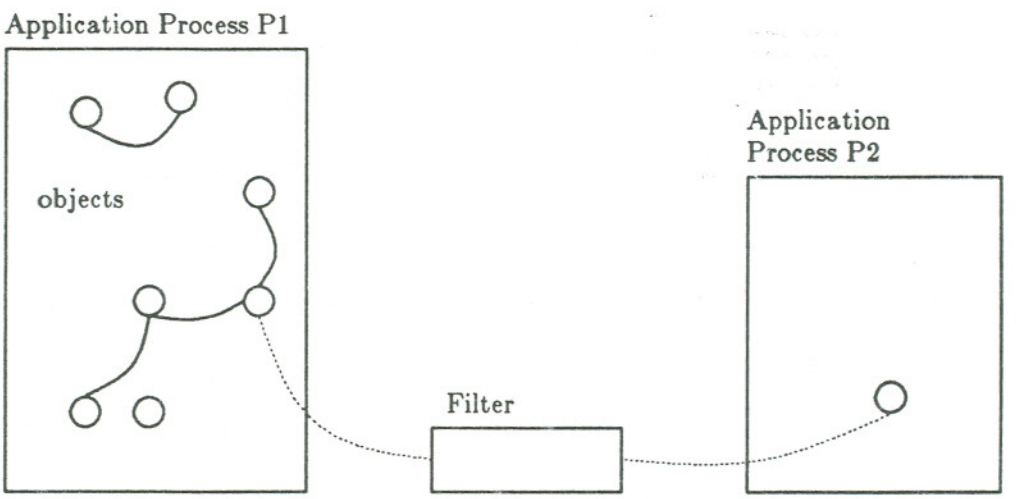


Fig. 3.2 Object relationships within and outside a process.

processes. An object may also be related to objects in other application processes, and the process must identify them as objects that may have filters established on them.

3.1.2. Filters

A filter identifies how two objects are related, and holds information about each object and the process it belongs to. The two objects may be from the same process, or from different processes, and the filter connecting them contains a well-defined transformation that allows the constraint to hold in both directions without loss of information.

Since the relationship between two objects may be rather complex, a filter could be a composite of other simpler filters. For example, a filter between a Fahrenheit thermometer and a Celsius thermometer could consist of one filter to scale the ratio of the units, followed by another filter to translate the values to the same base, as seen in Fig. 3.3.

3.1.3. The FilterManager

The FilterManager, as its name suggests, manages the network of filters representing constraints between processes. It ensures that when one object in the filter network changes, all the objects related to it are updated correspondingly.

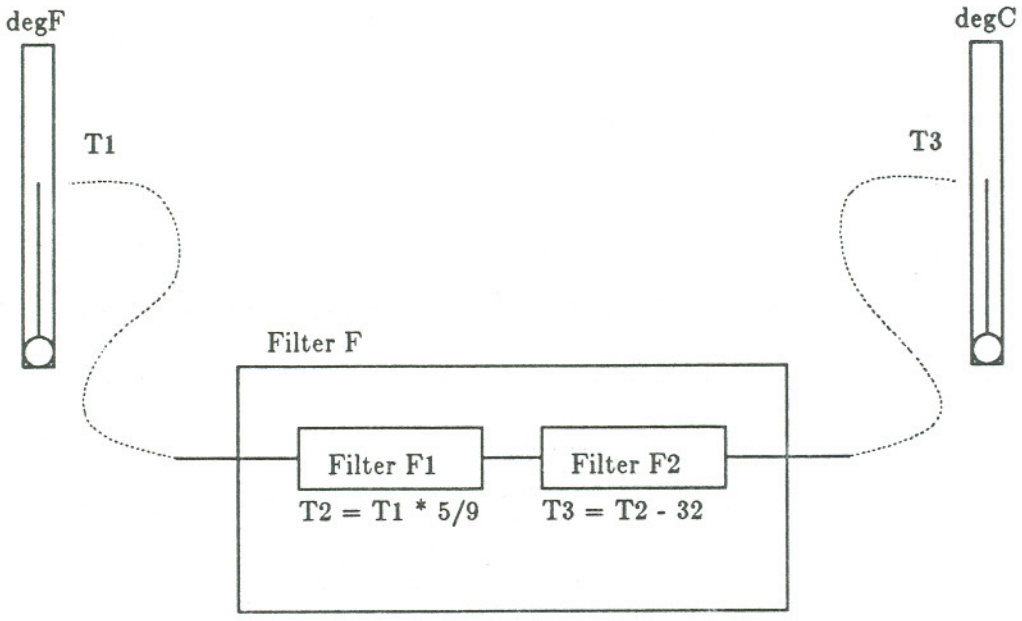


Fig. 3.3 A composite filter.

The services of the FilterManager are available to an application process after it is registered; a process may manipulate constraints on its objects by adding or removing filters. It may also find out about other processes and their objects.

3.2. Implementation Model

In the implementation model, the FilterManager consists of a server process, the *FilterManagerServer*, that is installed as an independent application process, and the *FilterManagerClient*, a set of routines compiled in each application process (Fig. 3.4). Communication between the FilterManagerServer and each FilterManagerClient takes the form of a well-defined communication protocol over *sockets*. For each object that the application identifies as a potential source or target object of a filter, the FilterManagerClient associates an *object handler* to perform additional functions on behalf of the object. A description of each component follows an illustration of how they relate.

To see how the components of the system relate, consider the scenario described earlier. The Smalltalk application having a display created with the FilterBrowser, the GemStone database, and the graph browser are applications executing on different machines (Fig. 3.5). Constraints exist between an object O1 in the Smalltalk application and object O2 in the GemStone database, as well as between object O2 and an object O3 in the graph browser.

To maintain the constraints between objects O1 and O2, and between objects O2 and O3, the FilterManager must be operational. Suppose the FilterManagerServer is executing, and each of the three application processes has recompiled the FilterManagerClient routines as a part of its context. The Smalltalk application, GemStone database and graph browser register

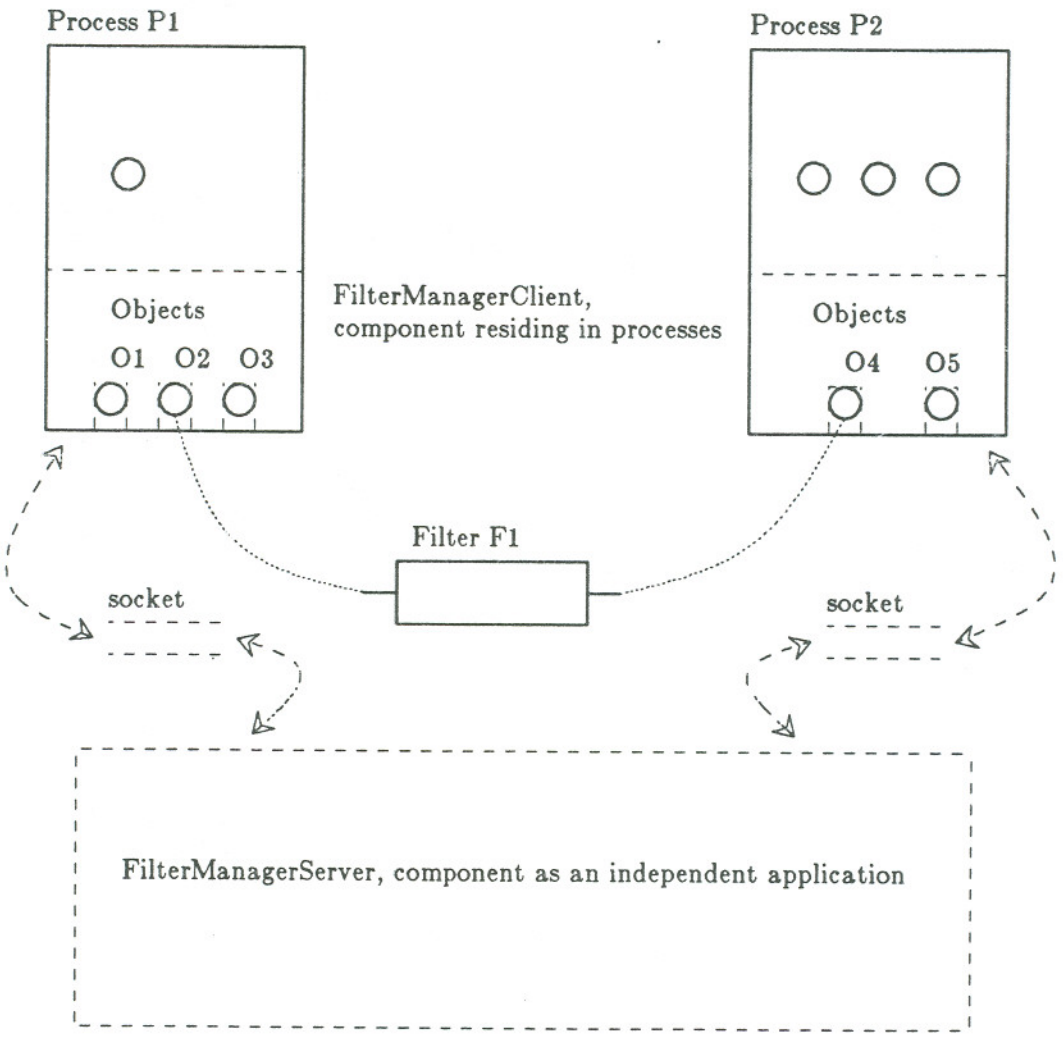


Fig. 3.4 The implementation model of the FilterManager.

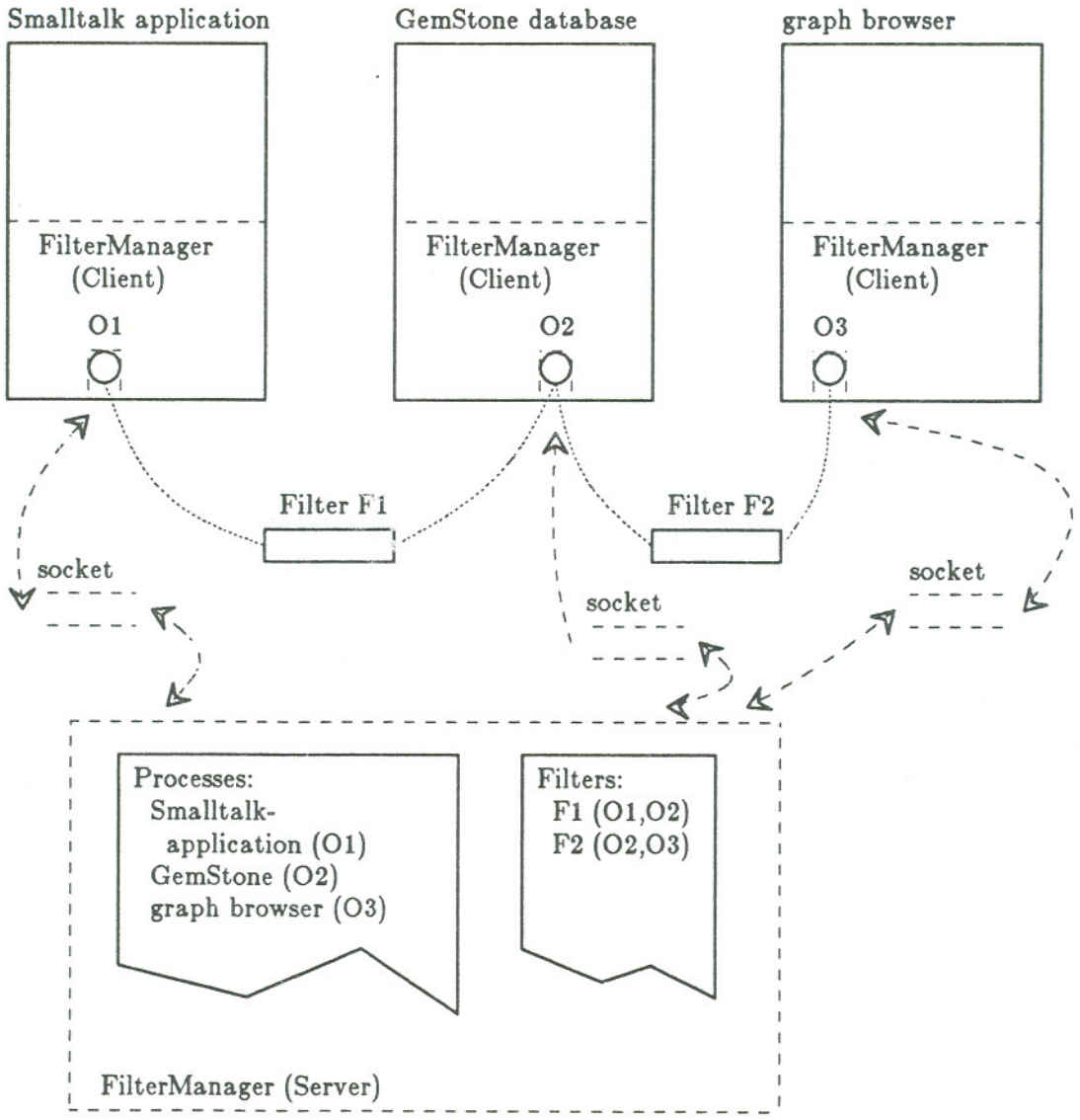


Fig. 3.5 The FilterManager in the example scenario.

objects O1, O2 and O3 respectively with the FilterManager. An object handler is created for each of these objects by the FilterManagerClient in the respective processes. Pointers to the object and its object handler are swapped, and the object handler keeps the new pointer to its object. The constraints are requested, and filters F1 and F2 are established by the FilterManager.

When object O1 in the Smalltalk application is updated, the update reaches the object via its object handler. Since object O1 is at one end of a filter, its object handler informs the FilterManagerClient in the Smalltalk application of the update. The FilterManagerClient communicates the update to the FilterManagerServer through a socket. The FilterManagerServer, which knows about filter F1, determines the target object of F1 given that object O1 changed, which in this example is object O2 in the GemStone database. The FilterManagerClient in the GemStone database is thus informed of the update required on object O2, and the object handler of O2 is delegated the task of sending the update to its object.

However, the FilterManagerServer also finds that object O2 is related to yet another object O3 through filter F2. Hence the FilterManagerClient in the graph browser is also informed of the relevant update. In this way, a change in object O1 is propagated to all other objects that have some relationship to it, whether directly, as for object O2, or indirectly, as for object O3.

The sections following describe in greater detail each component of the implementation model and its role.

3.2.1. Sockets

A socket [Joy, Lef, Sec] provides a process with a channel for communication with other processes. The sockets used by the FilterManager use the server-client role model; sockets, rather than pipes, must be used because the processes that need to communicate are not related in a parent-child process hierarchy.

A server socket is associated with an application process that is willing to provide some service to other processes; the socket provides a means for other processes to communicate their requests to the process providing the service. The server socket is created with a call to the operating system and given an identification by being bound to a communication port on the host specified by the service process, usually the host on which it is executing. It needs to have a queue initialized for outstanding service requests, which take the form of requests for connections from client sockets. To process a request, the service process accepts the first connection in the queue of connections on the server socket. An accepted connection is given a new descriptor and forms an open communication path between the service process and the process requesting the service. Messages can be exchanged in both directions over a connected socket between the two processes, and the

processes have to co-ordinate their exchanges. The service process can terminate its services by closing its open connections and shutting down its server socket.

A client socket is created by an application process which needs a service that another process is willing to provide. The client process initiates a request for the service by connecting the socket it has created to the server socket of the service process. The server socket is identified by the communication port it is bound to. When the connection has been accepted by the service process, communication between the two processes can proceed. The communication path is broken when either the server socket is shut down, or the connection is closed.

Sockets can be created with specific properties; the sockets used for the FilterManager must not block for read or write operations, so that an incomplete operation results in a return instead of a wait. For each server socket created, the FilterManagerServer must request the operating system to notify it via the operating system signal *SIGIO* when any operations are completed on that socket. Each accepted connection takes on the properties of the server socket and is treated as an open file descriptor by both processes using it. The limiting number of open connections that a process may have at any time is the number of available file descriptors. There may thus be multiple open connections on a socket between two processes, and a process may have multiple sockets.

A closed connection cannot be reopened. To restore a communication path between two processes after a connection has been closed, the processes

have to repeat the connection procedure. A communication path cannot be initiated over a server socket; only processes with client sockets may initiate communication.

3.2.2. Processes

When an application process is to have constraints between itself and another process maintained by the FilterManager, it has to identify itself to the FilterManager. During registration with the FilterManager, an application process identifies the objects in its context that may be known to, and constrained by, other processes. The registering process is given a unique name by which it is known to other processes and to the FilterManager. The identified objects are the only objects in the context of the process that may be related to objects of other application processes registered with the FilterManager. A process may register or deregister objects after the initial registration.

As long as a process remains registered, the FilterManager maintains any constraints on its registered objects, and ensures that when any of these objects become target objects of filters, they are updated. The FilterManager also handles the propagation of updates when any of these objects become the source object of a filter.

A terminating process is required to deregister from the FilterManager; a process can also deregister when it is no longer interested in participating

in the filter network. At process deregistration, all constraints between objects of the process and those of other processes are removed.

3.2.3. Object Handlers

For each object in an application process that may be the source or target object of a filter, the FilterManagerClient in that process creates and maintains an object handler. The object handler for an object assumes the identity of the object in the process and retains that identity, keeping a count of the filters that have been connected on it. All messages generated within the application process and sent to the object arrive at the object handler. The object handler examines each message for a resulting change in state of the object before sending the message to the object. If the state of the object will change and the object is constrained, the object handler notifies the FilterManagerClient once the object has been updated.

In addition, an object handler receives messages from the FilterManagerClient requesting it to increment or decrement the number of filters on the object it represents, and messages informing it of an update in the object when it is the target of a filter. The FilterManagerClient does not need to be notified of any changes in the object as a result of these messages.

When an object is deregistered, the pointers to the object handler and its object are swapped again, so that the identity of the object is restored. This is done after first ensuring that any existing constraints on the object

can be removed. Each type of object has its own type of object handler, and the protocol to an object handler of a type contains the protocol for the type, and other messages.

3.2.4. Filters

Each type of constraint has a separate filter definition, and a process identifying a constraint between two objects specifies the type of filter to be used, by name. The FilterManagerServer creates an instance of a filter of the specified definition to connect the two objects. A filter contains information about the type of constraint, the two objects being constrained, and the application process to which each object belongs. The only restriction that the FilterManager places on the filters that can be established is that an object cannot be directly related to itself. Filters that relate two objects of the same process are managed by the FilterManager in the same manner as filters spanning processes.

When an object at one end of a filter is updated, the FilterManagerServer tells the filter the identity of the changed object, so that it knows which participant of the filter is the source object. The filter also receives sufficient information about the update, so that together with the type of constraint that the filter represents, it can determine the kind of update that must be made to its target object.

3.2.5. The FilterManager

3.2.5.1. The FilterManager's State and Initialization

The FilterManager consists of two separate sets of routines. The FilterManagerServer executes as a separate application process, maintaining information about the dynamic environment of application processes and the constraints existing among them. The FilterManagerClient is a set of routines compiled into each application process, maintaining information for, and performing functions of, the FilterManager specific to the process.

The FilterManagerServer keeps a directory of the processes that are registered; for each process, it maintains a server socket, called the *request* socket, and an open connection, the *target* connection, using that socket. A set of constrained objects is associated with each process. The FilterManagerServer has a directory of filter definitions and a table of all the filters that are established. In addition to a request socket for each registered process, it maintains a special server socket, the *registration* socket.

The FilterManagerClient keeps a directory of objects (actually, their object handlers) for all objects in the application process that may have filters established on them. It keeps the identity of the request socket, which is the server socket set up by the FilterManagerServer for communication with the client, and maintains the same target connection that the

FilterManagerServer has for that process, using that socket.

Registered processes and their objects are known to the FilterManagerServer by unique symbolic names. The FilterManagerClient in each application process maps the symbolic name of a registered object to the object in the process. Filter types are identified by name in a process and mapped to their corresponding definitions by the FilterManagerServer. The FilterManagerClient keeps the identity of a server socket by the communication port number and the host it resides on; all other sockets, including open connections, are identified with socket descriptors similar to open file descriptors.

When the FilterManagerServer is invoked, it creates the registration socket and writes the socket address in a standard file so that any registering process can determine where a communication path to the FilterManager is to be found. To cater for application processes residing on different machines, the registration socket should really be maintained by a name server. The FilterManagerServer polls the registration socket for connection requests from registering and deregistering processes.

3.2.5.2. Process Registration

The FilterManagerClient, having been recompiled in the application process, is first active when the process registers (Fig. 3.6). It requests a connection on the registration socket to register the process and to determine

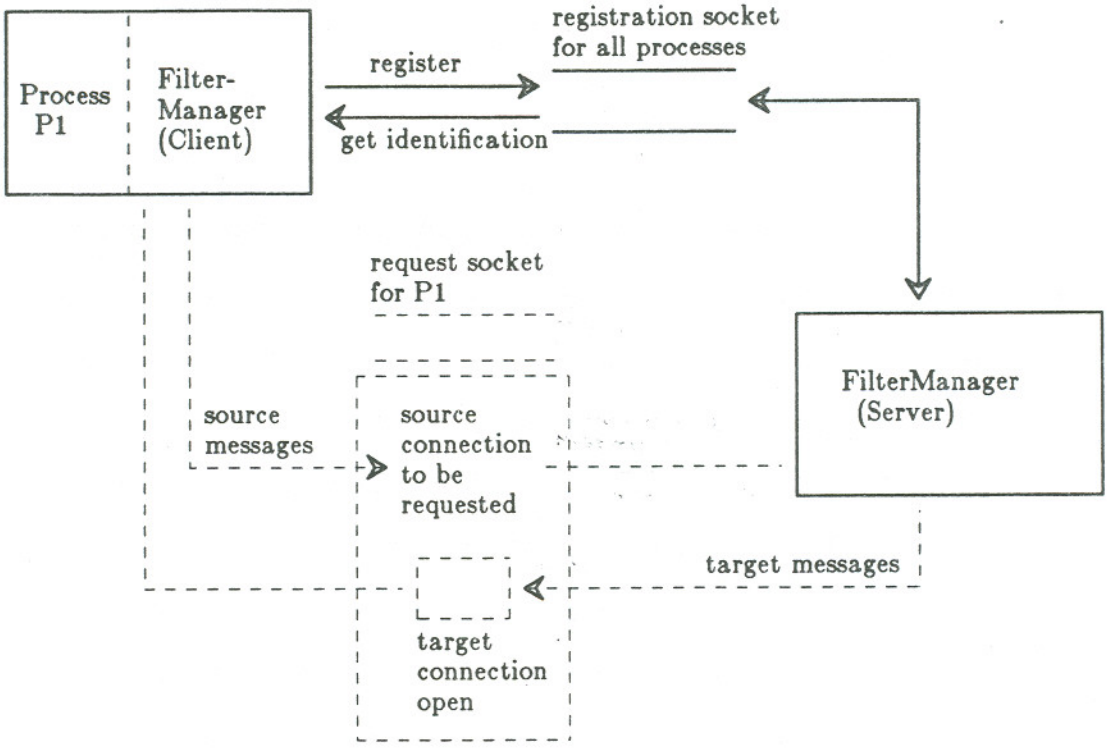


Fig. 3.6 Process registration with the FilterManager.

the unique identification that the FilterManagerServer will use for the process.

Once an application process registers with the FilterManager, the FilterManagerServer establishes a separate request socket for that process to

initiate any subsequent communication. Since connections cannot be initiated on a server socket, an open connection, the target connection, is established using the request socket to allow the FilterManagerServer to initiate message exchanges with that process. Both the request socket and the target connection are maintained until process deregistration, and the FilterManagerServer polls the socket for connection requests.

The FilterManagerClient handles *source* messages, messages initiated by itself to the FilterManagerServer, and *target* messages, messages initiated by the FilterManagerServer to itself, with two separate processes in a parent-child hierarchy. Managing source and target messages separately allows an application process to be both making a request to the FilterManager and servicing a request from the FilterManager simultaneously without deadlocks. (Deadlock issues are discussed further in Section 4.4.) The FilterManagerClient thus spawns a child process that waits at the target connection of the application process for target messages. It is then dormant in the application process until one of three events occurs: a message arrives from the FilterManagerServer, a constrained object is updated, or the programmer of the application requests some services of the FilterManager.

3.2.5.3. Target Messages

When the FilterManagerServer needs to initiate a message exchange with the FilterManagerClient in an application process, it sends a target

message via the target connection (Fig. 3.7). This situation arises when the process has a target object of a filter that has to be updated for consistency, as well as when the FilterManagerServer needs to know about the registered objects of that process.

The application process receives a notification from the operating system via the SIGIO signal that wakes the child process waiting at the target connection. The child process interrupts the application process to process

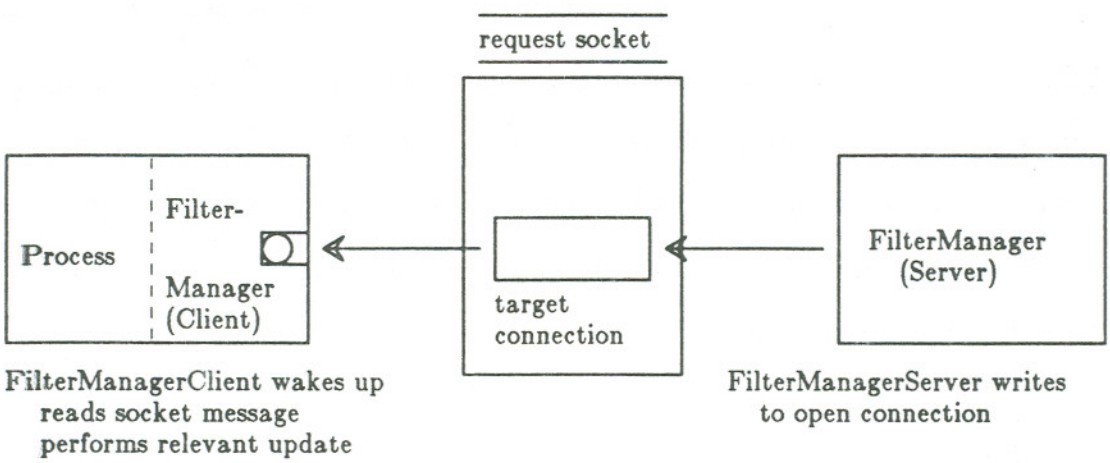


Fig. 3.7 Process has the target object of a filter.

the message, before becoming dormant again, awaiting the next target message.

3.2.5.4. Source Messages

The processing of a source message requires the FilterManagerClient in an application process to request a connection, the *source* connection, via the request socket of that process (Fig. 3.8). Source connections are requested on demand by the FilterManagerClient and need not be kept open, as in the case of target connections. Once the FilterManagerServer has accepted the connection and serviced the request, the connection is closed.

Source messages arise when a constrained object in an application process changes and the FilterManagerClient in that process needs to provide the FilterManagerServer with the identity of the changed object and the nature of the change. The FilterManagerServer uses the filter information it keeps to propagate the update of the object to all processes that have objects related to it.

Some routines are provided by the FilterManagerClient for the programmer of an application to change the set of objects that may be constrained, and to add and remove filters. There are also queries that can be made on the identities of other registered application processes, and their objects. Except for the registration of objects and the deregistration of an unconstrained object, these utilities also generate source messages. In

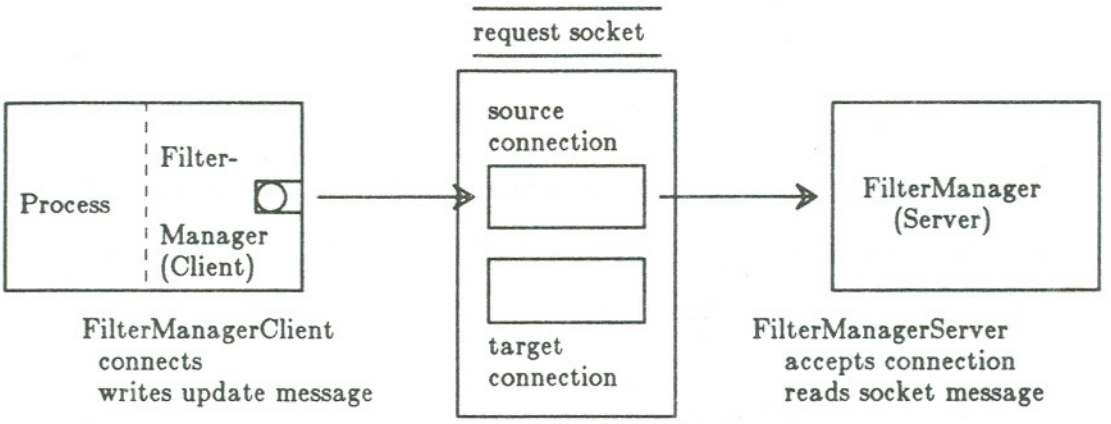


Fig. 3.8 Process has the source object of a filter.

addition, when a filter is added, initial constraint-satisfaction has to be performed so that the newly-constrained objects are consistent.

3.2.5.5. Process Deregistration

At process deregistration, the FilterManagerClient is responsible for informing the FilterManagerServer that the application process no longer wants constraints to be maintained on its objects. The FilterManagerClient sends a deregistration message via the registration socket, and waits for the

FilterManagerServer to inform it that all constraints are satisfied and deregistration can proceed. In the meantime, the client processes any further socket messages that the FilterManagerServer sends via the target connection. The client must process these to ensure that its state is consistent with the other processes at the time of deregistration. Once the FilterManagerClient has completed the consistency checks and is allowed to continue with deregistration, it takes the relevant actions to close its socket and discards the object handlers, reinstating the objects they represent. The FilterManagerClient terminates its processing and removes its state from the state of the application process so that none of the information it maintained remains with the process.

3.3. Assumptions

The FilterManager assumes the equivalence of all filters connecting a group of related objects. In a network where every constraint on a group of related objects is equivalent, the objects can be grouped into equivalence classes. Each object in an equivalence class is related to every other object in the class. A change in one object needs to be propagated to each object in its class exactly once in order for all objects to achieve a consistent state. Even filters in the network that were not applied would have their constraints satisfied once every related object has had one update. The FilterManager uses this assumption when propagating updates to all objects

that are related to a changed object. As long as each new filter established is equivalent to the filters that are connected to the two objects being constrained, the resulting filter network will be consistent.

CHAPTER 4

SYNCHRONIZATION IN THE FilterManager

4.1. Communication Protocol

As with any processes that communicate, the FilterManagerServer and the FilterManagerClient component in each process need to have a well-defined protocol for communication once a connection has been accepted. The sending and receiving of messages must be co-ordinated to ensure that messages are not lost or in error, and that deadlocks do not occur. Most of the details of message transmission are already handled through the implementation of sockets by the operating system and the supporting communications software.

However, there are a number of different implementations of sockets, depending on both the domain and type of sockets, as well as the protocol used. Rules vary with implementation for reliable delivery of each message sent, for sequencing of message delivery to tally with the sequence of message sending, for message boundaries, and even for error recovery. To avoid a dependency on the type of sockets used, the FilterManager employs a simple protocol for synchronous communication between the FilterManagerServer and the FilterManagerClient in each application process. This communication protocol is then mapped into the specific socket implementations as read-write sequences.

4.1.1. Sequencing of Message Exchanges

Once a connection from the FilterManagerClient in an application process has been accepted by the FilterManagerServer, their message exchanges are synchronized and take the following sequence: type of request, response, text of request, reply. This sequence lends simplicity to synchronization and deadlock avoidance. The specific exchange of messages depends on the context for which communication is taking place.

When the FilterManagerClient in an application process is requesting information from the FilterManagerServer, it initiates a source connection on the request socket and sends a message over the socket indicating that a query is to follow (Fig. 4.1). It then waits until the FilterManagerServer responds before sending the text of the query and awaiting a reply. Similarly, if a filter is to be established or removed, or a constrained object has changed, the FilterManagerClient indicates that a request for service is to follow, and then waits for the FilterManagerServer to respond before sending the text of the request over the socket. To standardize the message exchanges, the FilterManagerClient also waits for the FilterManagerServer to indicate that it has completed the request.

The FilterManagerServer initiates messages to the FilterManagerClient in an application process in a similar way (Fig. 4.2). In this case, the messages to be sent are either instructions to the FilterManagerClient about a change of its state, or a request for information, and they are sent via the target connection between the two processes that is kept open for this

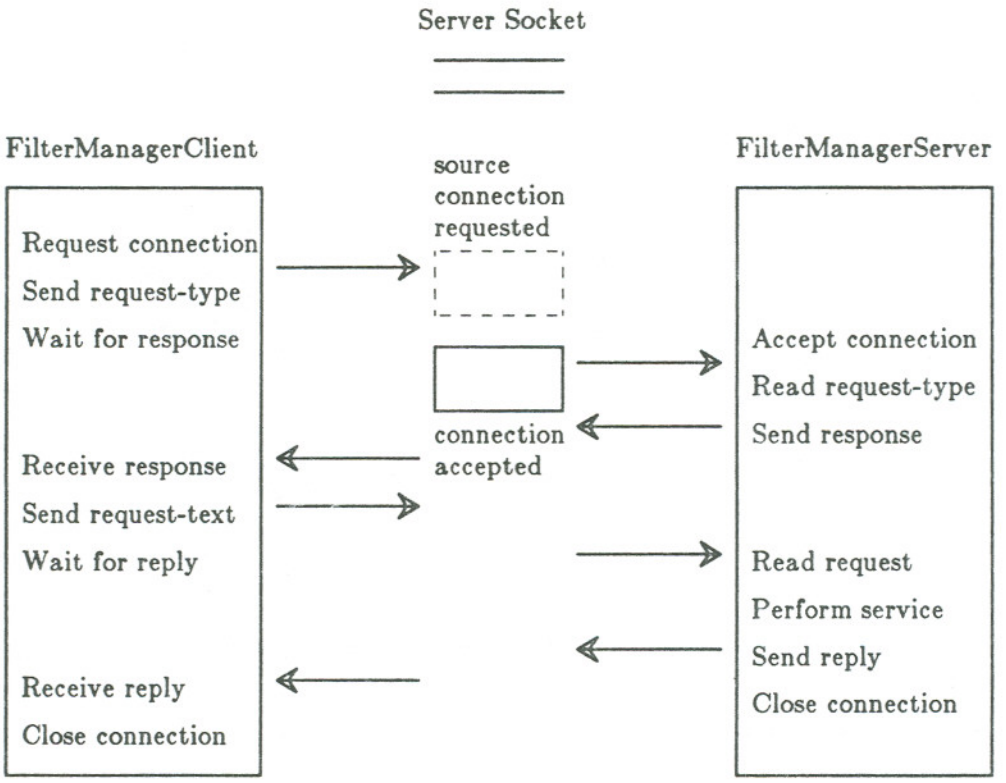


Fig. 4.1 Message exchange initiated by FilterManagerClient.

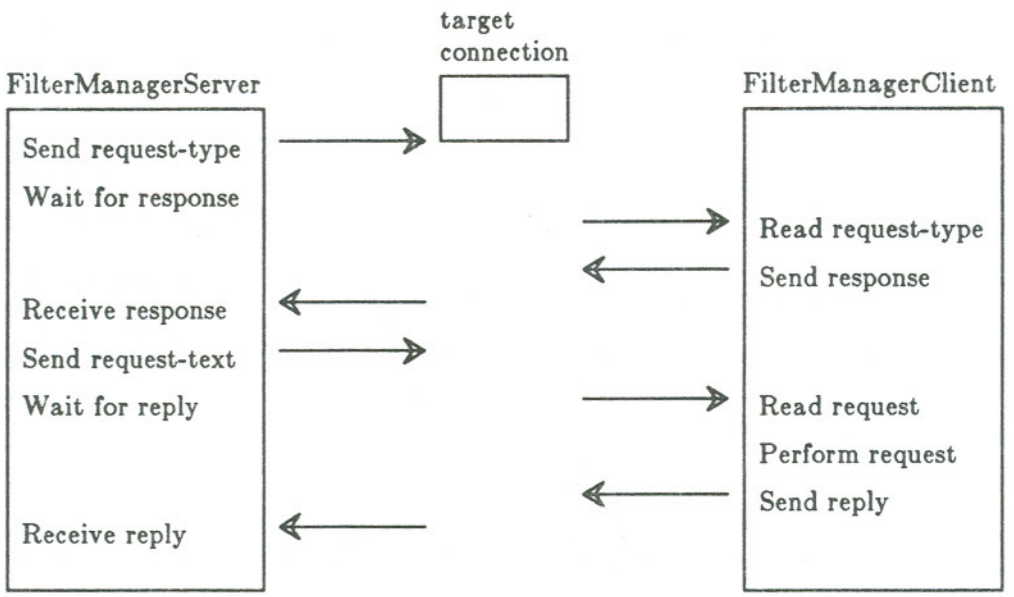


Fig. 4.2 Message exchange initiated by FilterManagerServer.

purpose. The FilterManagerServer sends a message indicating the type of request to follow, waits for the FilterManagerClient to respond, and proceeds to send the text of the request. Again, for uniformity, the FilterManagerServer waits for a reply from the FilterManagerClient in both cases.

4.1.2. Assumptions

In employing the protocol described, it is assumed that the sockets used provide a guaranteed delivery of all messages sent. Neither the FilterManagerServer nor the FilterManagerClient concerns itself with error recovery from lost messages and message resends. Stream sockets, the specific implementation of sockets used, provide error recovery and message resends, and an interface can be built to provide the FilterManager with sockets having this functionality if the sockets used for implementation do not do so.

Another assumption made is that outstanding connections are queued in the sequence in which they are requested. Although the FilterManagerServer applies its own policy of selecting a socket to check for an outstanding connection request, assuming that the connections on each socket are queued according to request ensures that no request for a connection is ignored. Again, stream sockets possess this facility, and other socket implementations that require the communicating processes to do their own sequencing will need an interface to provide this feature.

4.2. Update Phases

In a filter network, an object may be directly or indirectly related to other objects. When an object O1 is directly related to some other objects, a

filter is established between object O1 and each of the other objects (Fig. 4.3a). Objects O3 and O4 are indirectly related to object O1; there is no filter connecting objects O1 and O3, or O1 and O4, but there are filters connecting object O1 through some intermediate objects and filters, to object O3, and similarly, object O4 (Fig. 4.3b).

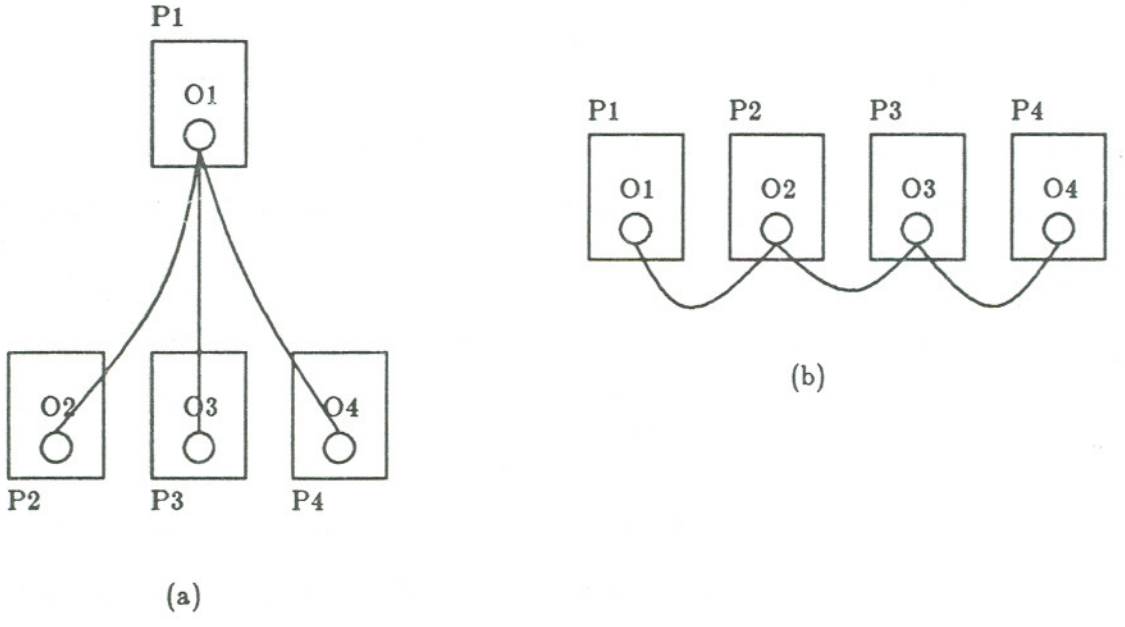


Fig. 4.3 Phasing updates.

In either case, when object O1 changes, a consistent state for the processes in the filter network is one where objects O2, O3 and O4 reflect the change corresponding to that in object O1. The FilterManagerServer therefore should treat the propagation of an update in source object O1 to target objects O2, O3 and O4 as a single task rather than as three separate tasks. Since the FilterManagerServer has information on all the filters established, it can propagate a change in an object to all objects that are both directly and indirectly related, as a single update phase. At the end of the update phase, consistency has been re-established.

To propagate updates in phases, the FilterManagerServer accepts a request from the FilterManagerClient in one application process at a time, and completes the processing of that request in a single update phase before proceeding to process the next request. For example, suppose the FilterManagerServer receives a notification from the FilterManagerClient in process P1 that object O1 has been updated (Fig. 4.3a, b). An update phase would then constitute the FilterManagerServer interacting with the FilterManagerClient in each of the processes P2, P3 and P4 to effect the relevant updates in objects O2, O3 and O4. Only after these interactions does the FilterManagerServer accept another request from the FilterManagerClient in any process. Each FilterManagerClient, on the other hand, has a child process that handles update requests even while it has a connect request outstanding. The child process interrupts the processing of the application and the FilterManagerClient, whenever an update request arrives, to perform the update. This scheme ensures that an object is

inconsistent in its relationships by at most one update at any time.

When a target object of a process is first updated by the FilterManager in an update phase U1, and then changes again as a result of constraint-satisfaction or computation within the private object space of the process, the second update is not part of update phase U1. The second update thus makes the object become the source object of a filter in a separate and subsequent update phase, U2.

4.3. Polling Requests for the FilterManager's Services

When many application processes are registered with the FilterManager, more than one FilterManagerClient may request service from the FilterManagerServer at the same time. To ensure that only one of the requests is processed at a time and hence enforce update phases, the FilterManagerServer determines when it is ready to accept a request, and which request it will accept.

For simplicity, the FilterManagerClient in each registered application process is polled for a request to the FilterManagerServer in a round-robin fashion. Polling takes the form of determining if a connection has been requested on the request socket of a process. The registration socket of the FilterManagerServer is also polled together with the other sockets. This policy implies an equal priority for each request. Once a connection is found, polling stops, the connection is accepted and the request serviced before

polling proceeds.

Different schemes of polling could implement priorities to include some knowledge of the states of the processes. For example, the FilterManagerClient in a process may be polled with a relative frequency proportional to the number of constrained objects it has. As long as polling is not continued during an update phase, the FilterManagerServer and the FilterManagerClient in each application process are synchronized.

4.4. Deadlocks

In the context of the FilterManager and its communication protocol, deadlocks can arise in the following situation: The FilterManagerServer is waiting for the FilterManagerClient in an application process to respond, and the same FilterManagerClient is waiting for a reply from the FilterManagerServer.

An example is when the FilterManagerClient in a process P1 has requested a connection and is awaiting the response of the FilterManagerServer. However, the FilterManagerServer is busy processing the request of the FilterManagerClient in another process P2, and finds that some object in process P1 is the target object of a filter. The FilterManagerServer sends a message to the FilterManagerClient in process P1 and awaits a response. Both are now waiting, and neither can proceed. When an object in a process is indirectly related to another object in the

same process via external filters, a similar situation results.

To avoid this situation, an implementation of the FilterManager must ensure that the FilterManagerClient in an application process is installed to handle source and target messages simultaneously. Each client process thus has two additional operating system processes in a parent-child hierarchy that implement the FilterManagerClient. At process registration, the FilterManagerClient is initiated as the parent process managing source messages, and spawns a child process to manage target messages. While the parent process of the FilterManagerClient in an application process is waiting for its connection to be accepted, or for the FilterManagerServer to reply, the child process responds to, and takes action on, any target messages.

A parent-child hierarchy of processes provides simplicity in sharing of sockets and a single control over the processes. Besides avoiding deadlocks, this approach allows updates on target objects in an application process to be effected as soon as a message containing the required update is delivered.

CHAPTER 5

ISSUES IN THE FilterManager SOLUTION

This chapter discusses the policies that the FilterManager uses, and the alternatives.

5.1. Cycles in the Filter Network

The filters established on the objects of application processes registered with the FilterManager, and their objects, may be treated as a graph; the objects being the nodes of a graph, and the filters the edges between them (Fig. 5.1).

The propagation of a change in a constrained object to all related objects is analogous to visiting every node in the connected subgraph that contains a specified node. If each object in the filter network is represented as a node in a graph, then all filters on it can be represented as edges incident on that node. When an object becomes the source object of a filter, it forms the root of a connected subgraph to be traversed. Each filter activated is an edge traversed, and each target object updated is a node visited. The objective, then, is to visit each node in the subgraph starting from the root.

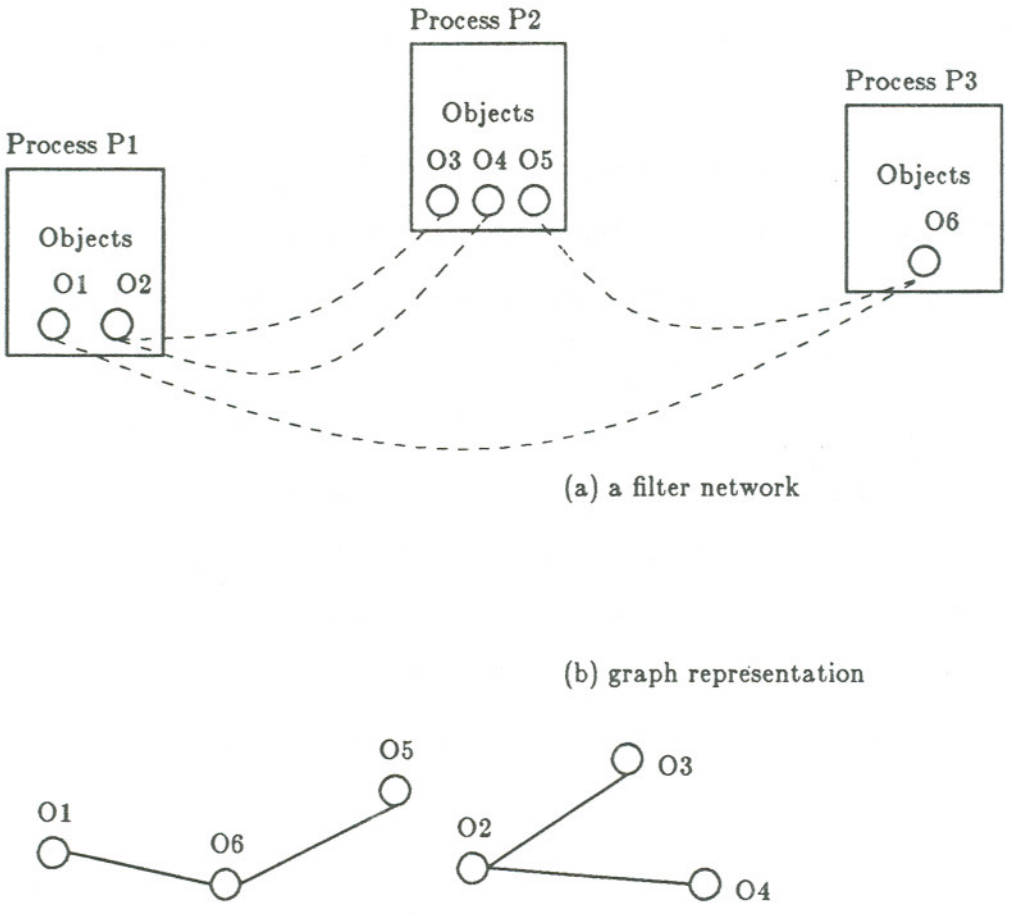


Fig. 5.1 Graph representation of a filter network.

Since filters are established to identify constraints representing the way objects are related in a set of application processes, it is not unusual for

cycles to occur in the graph representing a filter network (Fig. 5.2a).

When a cycle occurs in the connected subgraph of a filter network to be traversed, an edge should never be traversed more than once, i.e., a filter should never be applied more than once in an update phase. Besides avoiding redundant work, this rule defines a bound on the number of updates needed to achieve consistency when an object changes. However, it does not prevent an object from being sent multiple update messages if it has many filters on it. Assuming the equivalence of all filters in a connected subgraph, it is necessary to send an object only one update message in a single update phase. To ensure that an object is a target object only once in an update phase, a traversal of the connected subgraph must never visit a previously visited node. This implies that during an update phase, some filters in the connected subgraph of objects are not applied. Since all the filters connecting those objects are assumed equivalent, however, any of those filters that are not applied would still satisfy their constraints as a result of the application of the other filters.

In each update phase, a breadth-first traversal is made on a subgraph of the filter network (Fig. 5.2b). The only objects that are examined are in the connected subgraph whose root is the object that changed. Each such object is visited exactly once in a breadth-first traversal order, and any filters on it will only be applied if the filter has not been applied yet and the object at the other end of the filter has not been visited yet. Breadth-first traversal defines a partial ordering on the updating of the objects in a filter network. However, under the assumption that all chains of filters between two objects

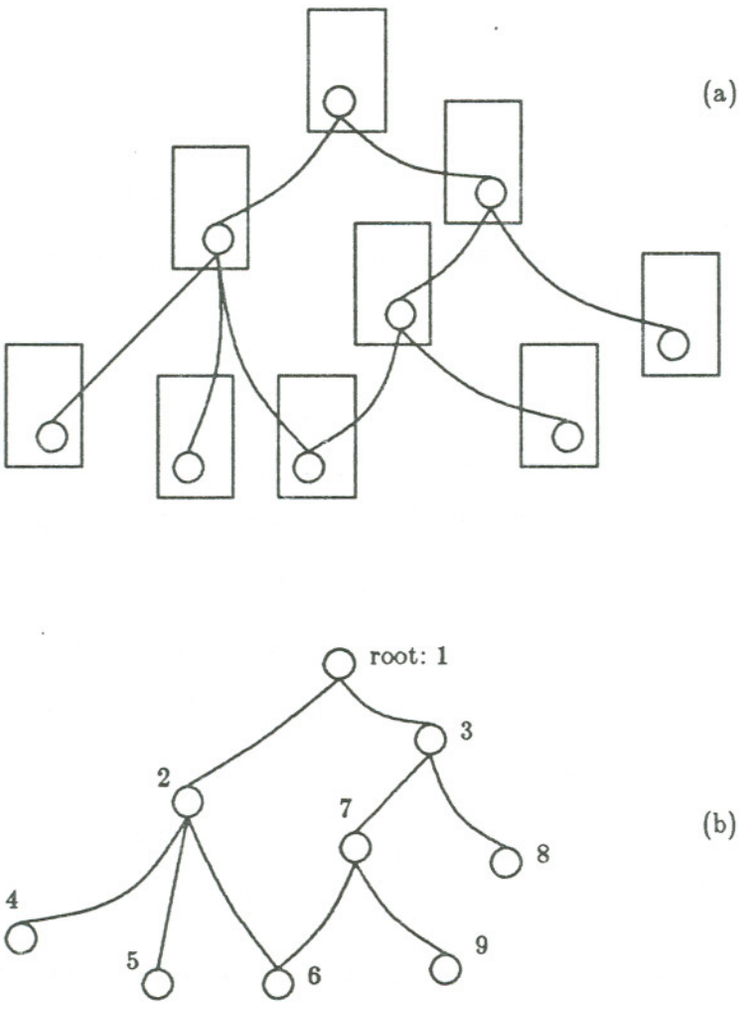


Fig. 5.2 Traversing a filter network with cycles.

are equivalent, the order of updates does not affect the result of the update phase.

5.2. Initial Constraint-Satisfaction

The two endpoints of a filter are typed; the role played by the FilterManager in the establishment of a consistent filter involves ensuring that the type of the object attached to each endpoint agrees with the type of the endpoint.

Immediately after a filter has been established, however, the two objects may not yet satisfy the constraint required between them, since they have been previously unconstrained. Thus it is necessary that an initial propagation of updates be done just after a filter is established. The choice of which of the two objects to designate as the source object is obvious only in some instances.

There are a few different approaches that the FilterManager could take to decide how initial constraint-satisfaction should be done. The simplest way is to always propagate changes from the process requesting the establishment of a filter. The target object that was previously unconstrained is then made to conform to the object in the process identifying the constraint. Conversely, the process requesting the establishment of a filter could have its object be made the target object of the filter, and assume a state corresponding to the object that is being

constrained in the other process. In the case where neither of the two objects being constrained belongs to the process requesting a filter, this approach is equivalent to making an arbitrary choice between the two objects for a source object.

An alternative would be to take the source object to be the object with a larger network of filters already established on it. This results in having fewer updates to propagate, but the network of filters on both objects have to be examined before a choice can be made.

The FilterManager could also identify the target object as the one whose state has not changed since its initialization, or the one belonging to the application process that was registered with the FilterManager later. In principle, however, the invocation of an application process and its registration with the FilterManager do not necessarily occur together, and this approach might not always be the right choice either.

Finally, a combination of these strategies could also be used. Ultimately, the option taken would trade simplicity in decision-making, and possibly more updates to be propagated, for more overhead in making a hopefully better decision, and fewer updates to be propagated.

The current implementation of the FilterManager makes the object in the process requesting the establishment of a filter the target object. When neither object belongs to the process, the source object is taken to be the second object identified for the filter.

5.3. Non-satisfiable Constraints

Not all constraints in a filter network can always be satisfied, even if they are all consistent with each other. For example, a Fahrenheit thermometer in one application process may be unbounded in its upper limit and may be manipulated to read an extremely high value, whereas a Celsius thermometer in another application process has an upper limit on its temperature readings. If both thermometers are supposed to be reading the temperature of the same room, then a constraint can be established between the two objects, which are in different application processes. When the temperature in the room is raised beyond some limit, the constraint between the two thermometers cannot be satisfied, since the Fahrenheit thermometer will be able to reflect the correct temperature while the Celsius thermometer will not, and their values will not reflect the same temperatures.

The FilterManager, however, does not concern itself with whether a constraint may be satisfied. It simply specifies the changes required of all target objects as a result of a change in a source object, without attempting to find out if a target object was able to perform a specified update. There are four possible ways of handling a constraint that cannot be satisfied.

One is for the target object to consider that the constraint is satisfied beyond a limiting value. For example, as long as the Fahrenheit thermometer has a reading beyond the equivalent maximum reading of the Celsius thermometer, the Celsius thermometer keeps its maximum temperature reading. The constraint is considered satisfied, and the

Fahrenheit thermometer may continue to register even higher readings.

Another approach is to force a compromise between the two objects to keep the constraint satisfied, so that the states that one object can have is bounded by the possible states of its related object. An example of this is where the two applications had thermostats instead of thermometers. When the Fahrenheit thermostat is moved beyond the limit of the Celsius thermostat, maintaining the constraint would mean that the setting on the Fahrenheit thermostat be limited to the equivalent of the maximum setting on the Celsius thermostat.

The application process with the target object could reject the update entirely, and restore the original value of the target object, effectively causing another update phase that cancels out the first change. The Celsius thermostat is reset to the setting that was in effect before the Fahrenheit thermostat was raised, and the constraint has to be satisfied by requiring the Fahrenheit thermostat to be reset to its initial setting also.

Finally, the filter could be flagged as unsatisfied, and the two application processes are notified.

The first approach considered a constraint satisfied within some limitations, whereas the other approaches require a constraint to be strictly satisfied. Hierarchies of constraints [Bor87], where constraints are given different strengths, allow a mix of strategies, so that weaker constraints can be violated to preserve stronger ones. New types of filters can be created to handle the first approach because it identifies how the target object is to be

updated, i.e., to a maximum limit, without requiring any subsequent update in the source object. When a subsequent update is necessary to strictly satisfy the constraints, however, a separate update phase is needed in which the object that was previously the target object becomes the source of a change. Requiring a separate update phase for an update resulting from a constraint that cannot be satisfied sets a bound on the number of updates required within a single update phase as a result of a change in one object.

The FilterManager currently does not provide for unsatisfied filters to be flagged and for application processes to be notified. Each process has to have its own policy to handle non-satisfiable constraints.

5.4. Methods of Propagating Updates

Assuming that only consistent filters are established, propagation of updates to enforce the consistency of the source and target objects could take one of two forms, or even possibly a combination of the two.

A target object could be made consistent with the source object by being told how it should update itself so as to correspond to the source object. For example, if a filter connects two thermometers, and one thermometer has its reading raised from T1 to T2 degrees, then the target thermometer could be sent an update message to raise its temperature reading by (T2 - T1) degrees. Propagating updates by sending target objects update messages corresponding to the updates in the source objects involves

a more thorough type-checking when filters are established. The filter must check that both of its objects will know how to respond to any update message that it might send to them.

Alternatively, a target object could be reconstructed to be made consistent with the updated source object. In the example, the target thermometer could be replaced in its application process with a thermometer whose temperature reading is T2 degrees. Propagating updates by replacing the target objects makes it necessary for the FilterManagerClient in a process to reconstruct an object for the process each time the object becomes the target of a filter. This method of propagating an update requires the structure and value of the updated object to be passed from one application process to another; the description of the object is passed as parts of socket messages between the FilterManagerServer and the relevant FilterManagerClients.

In the model for the FilterManager, the filters do the work of identifying how the target object should be updated. To propagate updates by update messages, a filter needs to know how the source object has changed, and then determine how the target object should be changed. To replace a target object instead, a filter uses the changed source object and the type of constraint it represents to reconstruct the target object. In both of these methods of updating target objects, if an update results in a type change of the target object, its object handler also has to be changed to the appropriate type. In an environment where filters are generally established on large and complex objects, sending update messages would be a more

efficient way of propagating updates, since replacement would require reconstruction of large objects. However, when filters are usually established on simple objects, perhaps parts of these same large objects, then replacement is a simple way to propagate updates.

The current implementation of the FilterManager propagates updates by reconstructing the target object; no type-checking is implemented.

5.5. Other Issues

5.5.1. Floating the Registration Socket

The final goal for the FilterManager is to manage constraints established between application processes that reside on different systems. Ideally, the application processes registered with the FilterManager should not have to depend on the FilterManagerServer being operational on a specific system. With this goal in mind, it is necessary to allow the port number and host name of the FilterManager's registration socket to be variable.

To ensure that the FilterManagerClient in a registering application process can determine where the registration socket is, this information is written in a well-known location, a file whose name is known to both the FilterManagerServer and every FilterManagerClient. Since the contents of this file changes each time the FilterManagerServer is terminated and reinitialized, the FilterManagerClient can always find the current location of

the registration socket. As long as application processes are registered, the FilterManagerServer cannot be terminated and the address of the socket remains valid. To prevent any FilterManagerClient from attempting to connect to a communication port when the FilterManagerServer is not operational, the FilterManagerServer sets the contents of this file to nil when it is terminating its services.

When the application processes utilizing the services of the FilterManager span machines, this file containing the location of the registration socket must be located on a machine whose file system is accessible and recognized by all the machines. Ideally, the processes on the various machines should not have to be concerned with the exact location and accessibility of this file; a filename service would require the FilterManagerClient in each process to provide only the name of the file. However, the provision of a filename service is non-trivial for heterogeneous systems, as each system has its own convention for filenames and file formats. This has been addressed by others [Ber, Com, Not] and has not been included in the scope of the FilterManager itself.

To provide the accessibility desired, a name server is needed for the address of the FilterManager's registration socket.

5.5.2. Signals

The model of the FilterManager includes an open connection for each registered application process, maintained by both the FilterManagerServer and the FilterManagerClient in the process. It serves as a communication path for message exchanges initiated by the FilterManagerServer to the FilterManagerClient, since the FilterManagerServer cannot initiate an exchange until a connection is accepted from the FilterManagerClient.

A much tidier way of handling this problem would involve installing a signal handling routine for each FilterManagerClient, causing it to initiate a connection upon receipt of a specific user-defined operating system signal. The FilterManagerServer would then send the signal to the relevant process whenever it needed to initiate a message exchange. However, the UNIX operating system limits a process to sending signals only to processes within the same process group, unless it executes in privileged mode. Hence this option is not viable, at least for testing and implementation.

CHAPTER 6

IMPLEMENTATION OF THE FilterManager

6.1. Implementation Environments

Implementation of the FilterManager was done in Smalltalk-80 to take advantage of the possibility of using previous work done in filters and constraint systems, including the FilterBrowser [Ege87, EMB87] and ThingLab [Bor81]. There was also a view to a simpler integration with some of these systems, which would be the application processes utilizing the services of the FilterManager.

A characteristic of the implementations of Smalltalk used for experimenting with and implementing the FilterManager is that the Smalltalk image assumes that it is always invoked from the console, and any attempt to invoke Smalltalk while remotely logged in results in its invocation at the console.

6.1.1. Tektronix Smalltalk

Experimentation with the design of the FilterManager started with Tektronix Smalltalk Version 2.3a on a Tektronix 4405 workstation with the UTek operating system. The operating system interface provided in

Smalltalk made the task of executing an operating system call from within this version of Smalltalk a simple one.

The design of the FilterManager was implemented in parts and each part was tested for practicality and the results of the testing were used to refine the design. Both design and implementation were carried to the stage where the FilterManagerServer and FilterManagerClient needed to introduce synchronization in their communication protocol to abstract from specific socket implementations. Since this implementation of Smalltalk assumes exclusive control of both the mouse and the console display when invoked, and does not execute in a window, it was not possible to view multiple Smalltalk images simultaneously. Multiple Smalltalk images can be invoked, but only one can be active while the rest have to be suspended and represented as icons. To see what is happening in another Smalltalk image, the current image has to be suspended into an icon and the icon of the other expanded.

A Smalltalk image puts itself into an idle state when suspended as an icon and refuses to respond to wakeup signals from other processes, waking up only when made the current image to be displayed. This makes it impossible for the FilterManagerServer and the FilterManagerClient in an application process to have any form of synchronous two-way communication using Tektronix Smalltalk. Therefore the implementation of the FilterManager on Tektronix Smalltalk was abandoned.

6.1.2. ParcPlace Smalltalk-80

The implementation of the FilterManager was eventually moved to ParcPlace Smalltalk Version 2.3 on a Sun-3 Workstation with the SunOS 3.5 operating system, and then ported to ParcPlace Smalltalk Version 2.4 on a Sun-4 Workstation with the SunOS 4.0 operating system. With either version of ParcPlace Smalltalk, each Smalltalk image can be invoked in a separate window on the console. With some modification to the Smalltalk image [Sch], it is possible to make Smalltalk time-share the operating system with other application processes. Two or more Smalltalk images, each in its own window on the console, can then execute simultaneously and communicate with each other.

Variables in workspaces implemented in ParcPlace Smalltalk must be declared before each use, and are not persistent (unlike workspace variables in Tektronix Smalltalk), because the workspaces are implemented without local variable dictionaries. Neither implementation of the FilterManager attempts to do anything about this, since the workspaces in the Tektronix implementation of Smalltalk do provide persistent local variables, and it can safely be assumed that persistent local variables can also be provided for ParcPlace Smalltalk. Instead, a global dictionary is created for each registered application process for the duration of its registration with the FilterManager. The process uses the dictionary to declare the required variables, and those variables are used when persistent variables are needed.

The implementation of the FilterManager in Version 2.3 makes use of an operating system interface that is provided with the Smalltalk code as a separate utility. As that interface is insufficient for the purposes of the FilterManager, some user-defined primitives[1] for Smalltalk had to be defined and installed to provide the remaining operating system interface that was required. The implementation uses stream sockets from the Internet domain, since the provided interface uses those. Socket communication for sockets from the Internet domain is not limited to the same file system, and the FilterManager can thus handle constraints defined between application processes on different systems.

Alternative socket implementations are stream and datagram sockets from the UNIX domain, and datagram sockets from the Internet domain. UNIX domain sockets restrict communication to processes sharing a single file system. Datagram sockets require an interface to ensure that the sequence of socket operations is identical on both the server and client ends, and to handle message resends in case of errors. Hence stream sockets in the Internet domain was the most practical implementation to use.

ParcPlace Smalltalk Version 2.4 arrived just about the time the implementation of the FilterManager on Version 2.3 was completed. Due to substantial differences in the two versions [Parc24], particularly in the

[1]Primitive methods are methods built into the Smalltalk virtual machine. They are not expressed in the Smalltalk language and cannot be changed by the programmer. A user-defined primitive is a piece of code provided by the user to Smalltalk to be used as a primitive method.

operating system interface for sockets, porting the FilterManager involved more than simply recompiling the source code. Although Version 2.4 contains operating system calls for socket operations as a part of the standard system, some other system calls that were provided in Version 2.3 are not available, and user-defined primitives had to be defined and installed for them.

The operating system on the Sun-4 Workstation, SunOS 4.0, schedules processes with a least-recently-used policy. This policy does not always work well when any of the application processes is on the same machine as the FilterManagerServer, since the FilterManagerServer is continuously active and polling for connections. Eventually the processor is given to the application process which is actively waiting for a reply from the FilterManagerServer, and the FilterManagerServer is waiting for the processor to schedule it so that it could send a reply. It helps to get the processor allocated to the FilterManagerServer if the cursor is placed in its window.

6.2. Organization of the FilterManager

The Smalltalk classes implementing the components of the FilterManager and its supporting classes are as follows:

FilterManager

    FilterManagerServer

    FilterManagerClient

ExternalFilter

    EqualityFilter

ProcessObject

ObjectHandler

    CollectionHandler

        TextHandler

    IntegerHandler

TreeNode

FMWkspDict

The socket interface with the operating system in ParcPlace Smalltalk Version 2.3 required enhancing the class Socket with server socket operations. For time-sharing support, the class InputState was modified, and a user-defined primitive was added. User-defined primitives consist of subroutines written in C and called from Smalltalk to provide additional functionality

that the Smalltalk implementation does not, and are used in the implementation of the FilterManager to perform operating system calls that do not have an interface in the Smalltalk implementation. In Version 2.3, user-defined primitives provide server socket operations to create and terminate a socket and to accept connections. In Version 2.4, user-defined primitives allow a process to determine its process id and to terminate itself.

The implementation of the FilterManager in Smalltalk classes corresponds very closely to the roles described in the implementation model, except for a few additional supporting classes. In addition, ParcPlace Smalltalk Version 2.4 is substantially different from Version 2.3 in the organization of classes supporting socket operations. Only details of specific interest, and differences between the implementation model and the actual implementation will be highlighted here.

The *FilterManager* class is a superclass of the FilterManagerServer and the FilterManagerClient classes, implementing what is common beween the two. In particular, because messages sent over sockets are byte strings, the FilterManager class contains methods to encode and decode these strings into the formats useable by its subclasses to ensure that consistency is maintained for socket messages. Some constants, such as the name of the file to which the location of the FilterManagerServer's registration socket is written, are also maintained by the FilterManager class.

There is only a single instance of the *FilterManagerServer* class, identified as a Smalltalk global variable, #FM, in the Smalltalk image that is installed as the FilterManager application process (Fig. 6.1). The

FilterManagerServer puts the name each registering application process

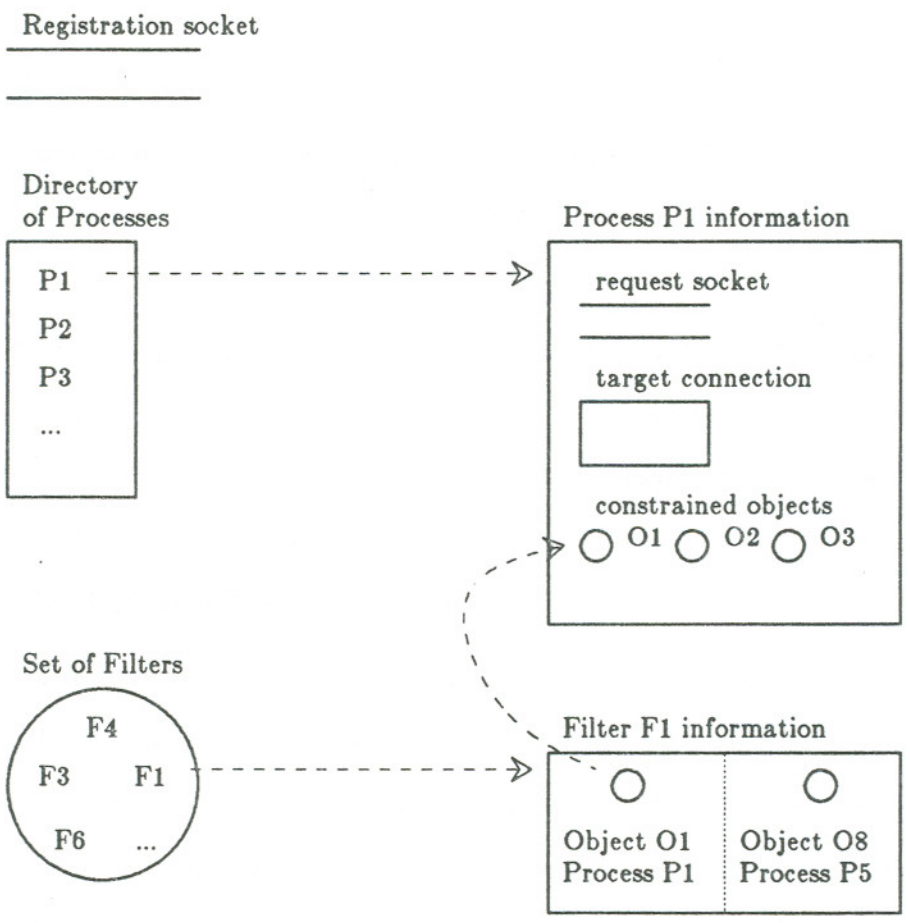


Fig. 6.1 Organization of the FilterManagerServer.

gives, into its directory of processes, mapping the name to the process itself. This name defaults to the process id of the process, as known to the operating system, if another process is already registered by that name, so that each registered process has a unique name.

Although the model of the FilterManagerServer includes knowledge of the definitions of filter types, the FilterManagerServer implemented in Smalltalk does not require a separate directory for these definitions, since the set of subclasses of the class ExternalFilter serves that purpose. Initialization of the FilterManagerServer starts up an instance of the Smalltalk class Process that performs the polling of sockets of registered application processes and handles the servicing of requests. The current implementation of the FilterManagerServer also does not allow the termination of the FilterManager's services while constraints are still established or client processes still registered.

The *FilterManagerClient* is installed as a single instance of the FilterManagerClient class in an application process, also identified as a Smalltalk global variable #FM (Fig. 6.2). Since the FilterManagerClient appears in a separate application from the FilterManagerServer, and each process can only have a single instance of the FilterManagerClient class, there is no conflict in this usage.

The FilterManagerClient class provides its application process with an interface to the FilterManager by means of utilities for FilterManager services, and prompts for an application name at registration time. The FilterManagerClient, when activated by the use of these utilities, serves as

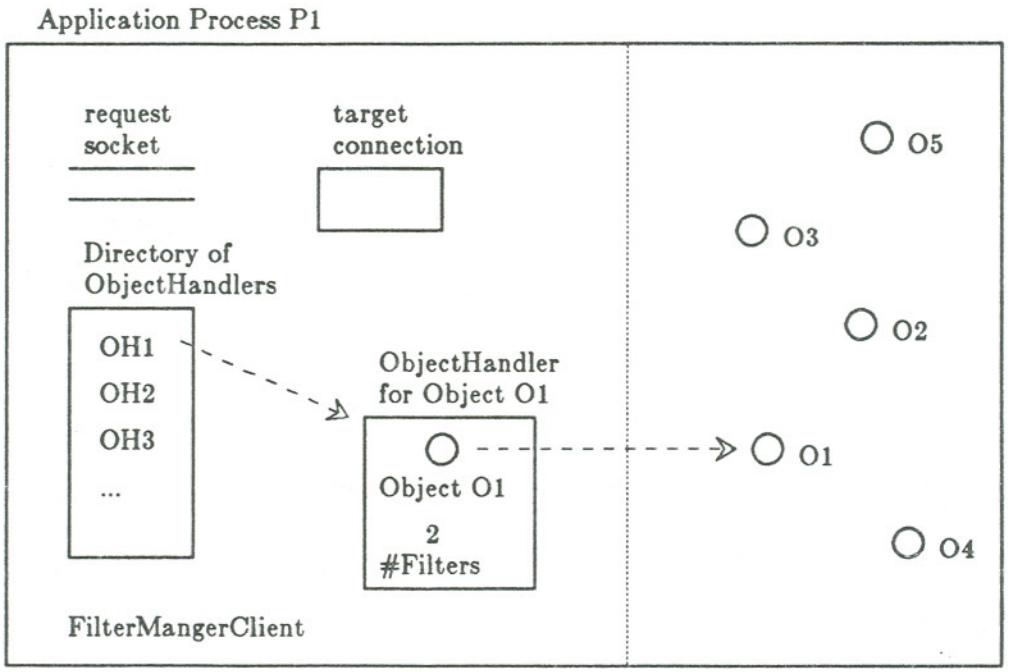


Fig. 6.2 Organization of the FilterManagerClient.

the parent process initiating message exchanges with the FilterManagerServer. Its spawned child process, an instance of the Smalltalk class Process, is started up at process registration and lies dormant waiting for messages initiated by the FilterManagerServer via the open connection. The application process is interrupted while a socket message from the FilterManagerServer is being processed and any object updates effected.

The Smalltalk global variable #FMDict, an instance of *FMWkspDict* which is a subclass of Dictionary, is set up by the FilterManagerClient for the application process it resides in. Variables that need to remain persistent for the duration of the time the process is registered have to be put into the dictionary with their names. This dictionary allows the programmer of an application process to view the objects without seeing their object handlers while the process is registered.

Each object that is declared as having the potential to be a source or target object of a filter has an instance of the *ObjectHandler* class created to manipulate it. The *become:* message in Smalltalk is used to swap identities of the instance of the ObjectHandler and the object. All messages meant for the object are thus received by its object handler instead, until the object handler is discarded and the object restored with its original identity. The class CollectionHandler implements object handlers for strings, arrays and sets, the class IntegerHandler for numbers, and the class TextHandler for text.

Instances of the class *ProcessObject* represent registered application processes.

Each subclass of the *ExternalFilter* class represents a different type of filter; the current implementation has a single subclass, the EqualityFilter class, which constrains its source and target objects to be identical in structure and equal in value. An instance of a filter represents a constraint between two objects. Since the FilterManager performs updates by reconstruction of target objects, each message to a filter about an updated

object carries information about the structure and value of that object.

Instances of the *TreeNode* class are used to represent nodes in the subgraph to be traversed when the FilterManagerServer is propagating updates of an object. The breadth-first search tree of the subgraph to be traversed is not specifically built for each update request, but a linked list of TreeNode instances, representing nodes in the tree to be visited, is maintained during a single update phase. The update phase is completed when traversal is completed and the list is empty.

Each instance of the *Socket* class represents a socket; instances of Socket are kept for the server sockets maintained by the FilterManagerServer, and for any open connections (both source and target). The operating system and the Smalltalk primitives implementing socket operations manipulate a socket through its socket identifier found in its instance variable *handle*. Since only client services are provided for Version 2.3, all server socket services are user-defined primitives coded in C and installed in the Smalltalk image. In Version 2.4, the classes for socket operations are organized differently, with the classes *UnixSocketAccessor*, *UnixIOAccessor* and *IOAccessor* providing both server and client socket services.

6.3. Implementations

The FilterManager is implemented in both Versions 2.3 and 2.4 of ParcPlace Smalltalk. Version 2.3 runs on a Sun-3 Workstation with the SunOS 3.5 operating system, and Version2.4 on a Sun-4 Workstation with the SunOS 4.0 operating system.

Each implementation of the FilterManager was tested using sample applications written in the same version of Smalltalk and executing on the same machine. In addition, the implementation of the FilterManagerServer in Version 2.3 handles filters whose processes span both versions of Smalltalk, each version on a different machine. The implementation in Version 2.4 cannot do the same because the operating system interface provided in that version of Smalltalk uses a different socket protocol.

Appendix A.4 contains an example of a network of filters spanning two applications that was used to test the FilterManager.

6.4. Portability and Environment Dependencies

The implementation of the FilterManager has been abstracted as much as possible to tie in with the implementation model and hence to avoid dependency on a specific environment. Some dependencies do exist, however, partly as a result of using Smalltalk as an implementation environment, and partly due to the interface with the operating system.

6.4.1. Time-Sharing Modification

The line between the environment and the application is indistinct for Smalltalk applications. The implementation of the FilterManager as an application was kept as separate from the Smalltalk environment as possible by creating subclasses of system classes, wherever possible, instead of modifying the system classes themselves. Because of the need to make Smalltalk time-share the operating system with other processes, however, this principle gets violated right from the start. It was necessary to modify the Smalltalk system code and state to achieve time-sharing. Furthermore, a user-defined primitive is used to add this functionality.

6.4.2. User-defined Primitives and Sockets

Since the installation and numbering of user-defined primitives are different in Versions 2.3 and 2.4 of ParcPlace Smalltalk, an application that is written identically in both versions would become two separate implementations once it uses the FilterManager.

As mentioned, Smalltalk primitives providing socket operations are implemented using different protocols in Versions 2.3 and 2.4, and different subsets of primitives required for the implementation of the FilterManager are available. Another dependency, mentioned earlier, is the difference in the organization of classes that provide an interface to the operating system, in

particular, file and socket operations.

6.4.3. File for Registration Socket

The file for the FilterManager's registration socket, FMServerFile, is created on a file system configured to be accessible to both the Sun-3 and Sun-4 Workstations. This file reference must be modified (in the class method *setup* of the FilterManager class) to the desired home directory before use.

When the FilterManagerServer is not active as an application process, or is not ready to provide services, such as when it is terminating, the file is nullified so that the FilterManagerServer in any process attempting to register will know that the services of the FilterManager are not available.

6.4.4. Policies

For simplicity, the FilterManager implements propagation of updates by replacing the target object with a reconstructed object containing the update corresponding to the change in the source object. Since equality filters are used, this involves reconstructing an exact match of the source object for the target object.

The policy for selecting the source object for initial constraint-satisfaction is also the simplest one — the source object belongs to the process identifying the constraint. When neither object of a filter belongs to the process requesting its establishment, the first object identified for the filter is taken to be the source object for initial constraint-satisfaction. This policy is implemented in the FilterManagerServer with the method *initialConstraint*, and by switching the objects used in that method, the process identifying the constraint could be made to have the target object instead. Similarly, the first object identified for a filter could be made a target object.

6.5. Integration of Applications

An attempt was made to install ThingLabII and to use it as an example of an application process that would use the FilterManager. This was successful only for the simplest objects created in ThingLabII because the current implementation of the FilterManager propagates updates by reconstruction of objects rather than by update messages. ThingLabII objects are generally complex objects with a structural depth that take up too much object space for the FilterManager to handle.

CHAPTER 7

EXTENSIONS TO THE FilterManager

7.1. Extensions to the Implementation

The FilterManager can be extended in two ways; extensions to the implementation involve an enhancement of the implemented FilterManager, whereas extensions to the model involve some redesign of the model that has been discussed.

7.1.1. One-Way Filters

The current implementation of the FilterManager model deals only with equality filters that are reversible, i.e., work in both directions. In particular applications, it might be useful to have filters that work in one direction only.

An example of a one-way constraint would be between a thermostat setting and the temperature of a room (Fig. 7.1). When the setting on the thermostat rises, the temperature of the room should eventually warm up to the temperature that was set on the thermostat. Similarly, when the setting on the thermostat is lowered, the room should also cool down. In other words, the temperature of the room is constrained to take the temperature at

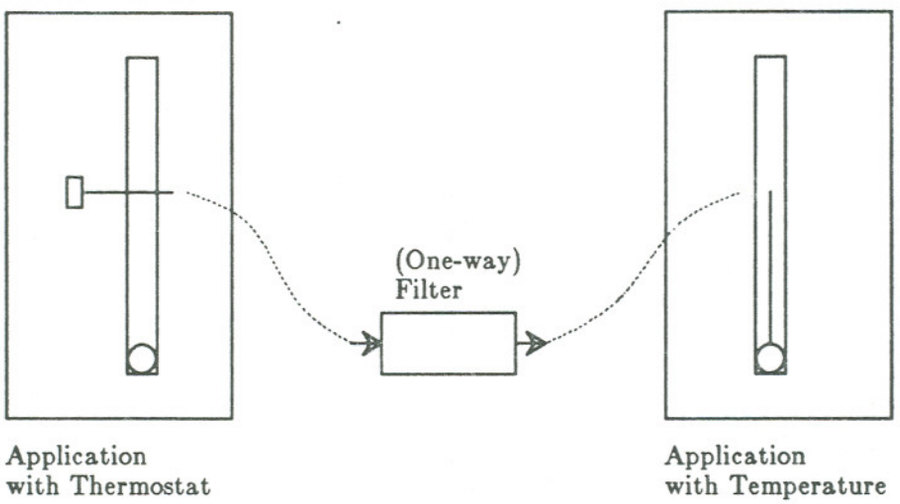


Fig. 7.1 A one-way filter.

which the thermostat is set.

On the other hand, if the temperature in the room should change due to some other causes, the setting on the thermostat does not fluctuate with it.

This example can be modelled using bidirectional filters (Fig. 7.2). Application process P1 has the thermostat T1 with the temperature setting and application process P2 has the temperature reading of the room. An additional thermostat T2 is created for process P2 and a filter PF, internal to process P2, is set up between thermostat T2 and temperature reading R to

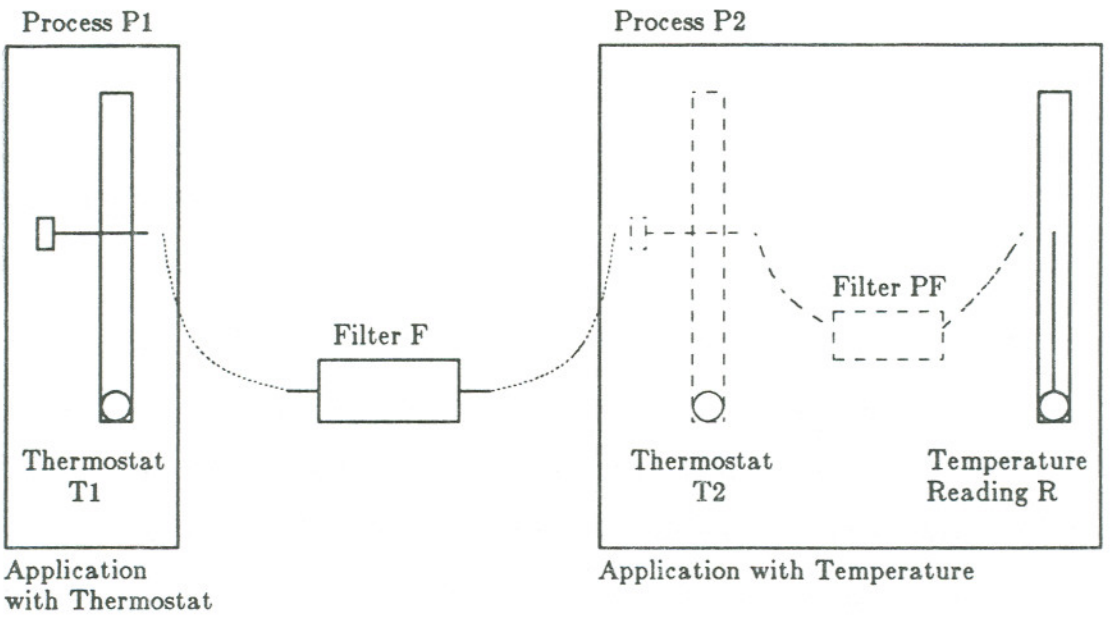


Fig. 7.2 Using a bidirectional filter.

handle the relationship between the thermostat setting and the temperature reading. A bidirectional equality filter F is then established between the thermostats T1 and T2.

When the thermostat setting on T1 is adjusted, filter F constrains thermostat T2 to take an identical setting, and filter PF causes the temperature reading to follow suit. If the temperature reading R fluctuates, however, filter PF that is maintained by the process P2 prevents a

corresponding change in the thermostat setting T2, and hence the thermostat setting T1 does not change. Although bidirectional filters can be used in this example, some overhead is involved, in both the additional object and filter that are internal to the process P2.

7.1.2. Type-checking

Since the endpoints of a filter are typed, it is possible to check that when a filter is being established, the types of its endpoints do correspond with the types of the objects that are being related. This type-checking has not been implemented, however, and the current implementation of the FilterManager assumes consistency without checking, leaving the responsibility to the user. To include type-checking, the FilterManager has to retrieve the types of the objects being related, and compare them with the types of the endpoints of the filter. Type-checking could be done by either the FilterManagerServer or the FilterManagerClient.

7.1.3. Policies and Options

Some of the issues in the design of the model for the FilterManager depend very much on the context in which its services are to be used, and the registered application processes. Since this is the case, the simplest options were generally chosen for implementation. However, in specific

circumstances, these options would probably not prove to be the best options, and it would be nice to implement more of these policies as options that the initiator of the FilterManager can select when starting up the FilterManagerServer. Policies that could be specified as options would be the strategies for initial constraint-satisfaction and non-satisfiable constraints.

7.1.4. Terminating the FilterManager's Services With Active Clients

In the current implementation, the FilterManagerServer, which represents the FilterManager as an application process providing services, will not respond to a shutdown request if any constraints still exist or any processes are still registered. When no processes are registered, the FilterManager needs only to know the state of its registration socket in order to terminate. To allow a termination of the FilterManager's services while processes are still registered, many more conditions need to be accounted for, including the states of the registered processes and their sockets.

7.1.5. Error Recovery

The FilterManager has been implemented without error-checking and recovery for broken communication paths. It assumes that a communication path, once established, is reliable until closed. In principle, this assumption

should be true since a large part of the error-handling and recovery is managed by the socket implementations and the communications software. In practice, however, a process might terminate abnormally and the FilterManagerClient in the process may not have the opportunity to deregister the process. Abnormal termination results in a broken communication path, and can be checked for during each interaction with a process over a socket.

7.2. Extensions to the Model

7.2.1. Filters and Environments

In the model of the FilterManager, the focus has been on equality filters. Equality filters are sufficient for representing a constraint between two objects that exist in the context of different application processes of similar environments. For example, an equality filter between two Smalltalk objects each in a different Smalltalk application process can constrain the two objects appropriately, since the two objects will have the same definitions if they are identical in structure.

In contrast, a C application process and a Smalltalk application process will not have the same definitions for identical objects, and this situation will generally be the case for application processes with dissimilar environments. To establish filters among objects in processes with dissimilar environments,

the FilterManager would also have to handle more complex filters that would perform the necessary transformation on an object from one environment to its equivalent object in the other environment.

In including complex filters, the conceptual model of the FilterManager remains unchanged; it is the implementation model that may change, as more information might be needed about the environments a process belongs to. A possible way to handle complex filters would be for the FilterManager to have its own independent type system. The FilterManagerClient installed in each application process will then perform the necessary type conversions on a constrained object into the corresponding type of the FilterManager and vice versa. This approach would also involve implementing a version of the FilterManagerClient component for each environment in which an application process might need the services of the FilterManager. Equality filters can then be used for this implementation, since environment transformations are handled by the FilterManager.

Another approach would be to define the transformations in the filter definitions; a filter type must be defined for object transformation between every pair of environments in which an application may need the services of the FilterManager. This approach would not only cater for multiple environments but also for constraints in which there is a bound beyond which the constraint is relaxed.

7.2.2. Persistence

The FilterManager has been modelled for a dynamic and transient set of application processes. The constraints established between two processes hold only as long as the FilterManager is in operation, both applications are executing, and the constraints are not removed. These constraints are not persistent beyond the deregistration of an application, and no provision is made for a constraint that really should be persistent. When an application that deregistered with the FilterManager registers again, its constraints must be re-established.

Although the FilterManager serves a purpose different from that of databases, persistence might be useful where a group of related applications are commonly in execution simultaneously. Each time more than one of these applications is invoked, the constraints between them must hold. If the FilterManager could manage persistent constraints as well, the users of these applications would find it very convenient not to have to check for the other application processes and re-establish the necessary constraints.

REFERENCES

[Art87] Artsy, Y., Chang, H., Finkel, R., "Interprocess Communication in Charlotte," IEEE Software, Vol 4, No 1, Jan 1987, pp 22-28.

[Bas77] Baskett, F., Howard, J.H., Montague, J.T., "Task Communication in Demos," ACM Proceedings of the Sixth Symposium on Operating Systems Principles, Nov 1977, pp 23-31.

[Bea82] Beach, R.J., Beaty, J.C., Booth, K.S., Plebon, D.A., Fiume, E.L., "The Message is the Medium: Multiprocess Structuring of an Interactive Paint Program," Computer Graphics, ACM SIGGRAPH '82 Conference Proceedings, Vol 16, No 3, Jul 1982, pp 277-287.

[Ber88] Bershad, B.N., Levy, H.M., "A Remote Computation Facility for a Heterogeneous Environment," IEEE Computer, Vol 21, No 5, May 1988, pp 50-60.

[Bor81] Borning, A., "The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory," ACM Transactions on Programming Languages and Systems, Vol 3, No 4, Oct 1981, pp 353-387.

[Bor87] Borning, A., Duisberg, R., Freeman-Benson, B., Kramer, A., Woolf, M., "Constraint Hierarchies", Proceedings, OOPSLA '87, Oct 1987, pp 48-60.

[Bro88] Brown, M.H., "Exploring Algorithms Using BALSA-II," IEEE Computer, Vol 21, No 5, May 1988, pp 14-36.

[Bro85] Brown, G.P., Carling, R.T., Herot, C.F., Kramlich, D.A., Souza, P., "Program Visualization: Graphical Support for Software Development," IEEE Computer (Feature on Visual

Programming), Vol 18, No 8, Aug 1985, pp 27-35.

[Bro84] Brown, M.H., Sedgewick, R., "A System for Algorithm Animation," Computer Graphics, ACM SIGGRAPH '84 Conference Proceedings, Vol 18, No 3, Jul 1984, pp 177-186.

[Com86] Comer, D., Murtagh, T.P., "The Tilde File Naming Scheme," Proceedings of the 6th International Conference On Distributed Computing Systems, May 1986, pp 509-514.

[Dio78] Dionne, M.S., Mackworth, A.K., "ANTICS: A System for Animating LISP Programs," Computer Graphics and Image Processing, Vol 7, No 1, 1978, pp 105-119.

[Dui86] Duisberg, R.A., "Constraint-Based Animation: Temporal Constraints in the Animus System," Computer Research Laboratory, Tektronix, Technical Report No. CR-86-37, Aug 1986.

[Ege87] Ege, R.K., "Automatic Generation of Interfaces Using Constraints," PhD Dissertation, Oregon Graduate Center, July 1987.

[EMB87] Ege, R.K., Maier, D., Borning, A., "The Filter Browser - Defining Interfaces Graphically," Proceedings, European Conference on Object-Oriented Programming, Jun 1987, Lecture-Notes in Computer Science 276, Goos, G., ed, Hartmanis, J., ed, pp 140-150.

[Gol83] Goldberg, A., Robson, D., Smalltalk-80: The Language and its Implementation, Addison-Wesley, 1983.

[Gro85] Grossman, M.B., "Humanizer - A Framework for Implementing Flexible Human-Machine Interfaces," unpublished manuscript, Dept of Computer Science and Engineering, Oregon Graduate Center, May 1985.

[Hoa78] Hoare, C.A.R., "Communicating Sequential Processes," Communications of the ACM, Vol 21, No 8, Aug 1978, pp 666-677.

[Joy] Joy, W., Fabry, R., Leffler, S., McKusick, M.K., Karels, M., "Berkeley Software Architecture Manual 4.3BSD Edition," UNIX Programmers' Supplementary Documents.

[Lef] Leffler, S.J., Fabry, R.S., Joy, W.N., Lapsley, P., "An Advanced 4.3BSD Interprocess Communication Tutorial," UNIX Programmers' Supplementary Documents.

[Lon85] London, R.L., Duisberg, R.A., "Animating Programs Using Smalltalk," IEEE Computer (Feature on Visual Programming), Vol 18, No 8, Aug 1985, pp 61-71.

[Mai86] Maier, D., Nordquist, P., Grossman, M., "Displaying Database Objects," Oregon Graduate Center, Technical Report No CSE-86-001, Jan 1986.

[Mor83] Morgenstern, M., "Active Databases as a Paradigm for Enhanced Computing Environments," Proceedings, Ninth International Conference on Very Large Databases, Schkolnick, M., Thanos, C., ed., Oct 1983, pp 34-42.

[Mye83] Myers, B.A., "Incense: A System for Displaying Data Structures," Computer Graphics, ACM SIGGRAPH '83 Conference Proceedings, Vol 17, No 3, Jul 1983, pp 115-125.

[Not88] Notkin, D., Black, A.P., Lazowska, E.D., Levy, H.M., Sanislo, J., Zahorjan, J., "Interconnecting Heterogeneous Computer Systems," Communications of the ACM, Vol 31, No 3, Mar 1988, pp 258-273.

[Parc23] The Smalltalk-80 Programming System Version 2.3, ParcPlace Systems.

[Parc24] The Smalltalk-80 Programming System Version 2.4, ParcPlace Systems.

[Pur87] Purdy, A., Schuchardt, B., Maier, D., "Integrating an Object Server with Other Worlds," ACM Transactions on Office Information Systems, Vol 5, No 1, Jan 1987, pp 27-47.

[Rei86] Reiss, S.P., "An Object-Oriented Framework for Graphical Programming," Brown University, Mar 1986.

[Rei84] Reiss, S.P., "Graphical Program Development with PECAN Program Development Systems," Proceedings, ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Apr 1984, pp 30-41.

[Rei83] Reiss S.P., "PECAN: Program Development Systems That Support Multiple Views," Brown University, Technical Report No. CS-83-29, Dec 1983.

[Sch88] Schiffman, A.M., "Time-Sharing Citizenry for Smalltalk-80 under UNIX," ParcPlace Newsletter, Vol 1, No 2, Winter 1988, pp 9-10.

[Sec] Sechrest, S., "An Introductory 4.3BSD Interprocess Communication Tutorial," UNIX Programmers' Supplementary Documents.

[Ste86] Stefik, M.J., Bobrow, D.G., Kahn, K.M., "Integrating Access-Oriented Programming into a Multiparadigm Environment," IEEE Software, Vol 3, No 1, Jan 1986, pp 10-18.

[Ste83] Stefik, M., Bobrow, D.G., Mittal, S., Conway, L., "Knowledge Programming in LOOPS: Report on an Experimental Course," AI Magazine, Vol 3, No 3, 1983, pp 3-13.

[Sun88] System Services Overview, Part No. 800-1753-10, Revision A, of 9 May 1988, Sun Microsystems.

[Tek87] Tektronix Smalltalk User's Manual, Part No. 070-6454-00, Nov 1987.

[Unix] UNIX Interface Reference Manual

[Veg86] Vegdahl, S.R., "Moving Structures between Smalltalk Images," Proceedings, OOPSLA '86, Sept 1986, pp 466-471.

APPENDIX

A.1 SMALLTALK CLASSES FOR THE FilterManager

The Smalltalk classes created to implement the FilterManager in both Versions 2.3 and 2.4 of ParcPlace Smalltalk are listed below. A count of the methods is given for each class.

Classes implementing the FilterManager:

FilterManager	21
FilterManagerServer	50
FilterManagerClient	50

Classes supporting the FilterManager's structures:

ExternalFilter	15
EqualityFilter	1
ProcessObject	14
ObjectHandler	14
CollectionHandler	14
TextHandler	4
IntegerHandler	1
TreeNode	9
FMWkspDict	5

In Version 2.3, the class Socket was enhanced with ten methods to provide server socket operations.

A.2 USER-PRIMITIVES FOR THE FilterManager

The following user-primitives were written in C for the implementation of the FilterManager in ParcPlace Smalltalk Version 2.3.

setupServer.c	"Create a server socket."
acceptConnection.c	"Accept an outstanding connection."
shutdownInMode.c	"Terminate a server socket."

The following user-primitives were written in C for the implementation of the FilterManager in ParcPlace Smalltalk Version 2.4.

retrievePid.c	"Get the process id."
terminateProcess.c	"Terminate a process."

A user-primitive was also added to provide time-sharing capability to the Smalltalk implementations:

awaitEvent.c

Installation of user-primitives to the Smalltalk executable requires interface routines to be added to it:

installPrim.c

A.3 HOW TO USE THE FilterManager

These instructions on the installation and use of the FilterManager assume that the user already has a ParcPlace Smalltalk image in use for some application. Installation comes in two parts; installing the FilterManager as an application providing services, and installing the FilterManager within an existing application.

General Installation

This installation must be done for the Smalltalk image that is to serve as the FilterManagerServer application process, as well as for each Smalktalk image that is to have the capability of utilizing that service. Relevant files can be found in the directory *installFM*.

1. Compile the user-defined primitives; execute the makefile.
2. Install the user-defined primitives; use *st80User* with the usual command-line parameters to invoke Smalltalk.
3. For Version 2.3, ensure that the operating system interface is in place; file-in the files *UnixSystemCall.st* and *Socket.st* from the directory */usr/smalltalk/unix.code*.
   For Version 2.4, perform updates to the socket interface; file-in the files *ioWait.st* and *connectTo.st*.
4. File-in the source code for the FilterManager: for Version 2.3, the files are *SocketServer.st*, *FM2.3-Filters.st*, *FM2.3-Handlers.st* and *FM2.3-Components.st*; for Version 2.4, the files are *FM2.4-Filters.st*, *FM2.4-Handlers.st* and *FM2.4-Components.st*.
5. If time-sharing is desired, apply the modifications given in [Sch]; the instructions can be found in the file *timeShare.ws*.
6. Initialize the FilterManager: edit the shared file in the method *createConstantsFile* and re-execute the method, open a workspace and do *FilterManager initialize*.
7. Save the Smalltalk image. The FilterManager is installed in the image. The Smalltalk executable is st80User.

Starting up the FilterManager's Services

Once installed in a Smalltalk image as the FilterManager, the FilterManagerServer must be started up as an application process providing services to other application processes.

1. Start Smalltalk using *st80User* with the usual command-line parameters and the Smalltalk image that is the FilterManager application.

2. Open a workspace and do *FilterManagerServer bootstrap*. This starts up the FilterManagerServer in this Smalltalk image.

Using the FilterManager's Services from an Application Process

Once the FilterManager has been installed in an application process, the process needs to be registered with the FilterManager before it can utilize its services.

1. Start Smalltalk using *st80User* with the usual command-line parameters and the Smalltalk image that contains the application.
2. Open a workspace and do *FilterManagerClient install*. A name will be requested for, and the FilterManagerClient is started up in this Smalltalk image as the global variable #FM.
3. Put the objects on which filters may be established into the dictionary at the Smalltalk global variable #FMDict as (object-name to object) associations. Identify these objects to the FilterManager with an *allowFiltersOn:* or an *allowFiltersOnSet*.
4. The FilterManager's services are now available; an example of some of the utilities can be found in the file *ClientExample.ws*. Sample applications can be found in the files *FMCdemo.ws* and *FMC2demo.ws*. (Please refer to Appendix A.4.)
5. The FilterManager's services may be terminated by a *FM shutdown*. It is necessary to do this before exiting from the Smalltalk image.

Terminating the FilterManager's Services

The FilterManager's services may only be terminated when all filters have been removed and all application processes have deregistered.

1. In a workspace, do *FM shutdown*. This terminates both the FilterManager and the Smalltalk image.

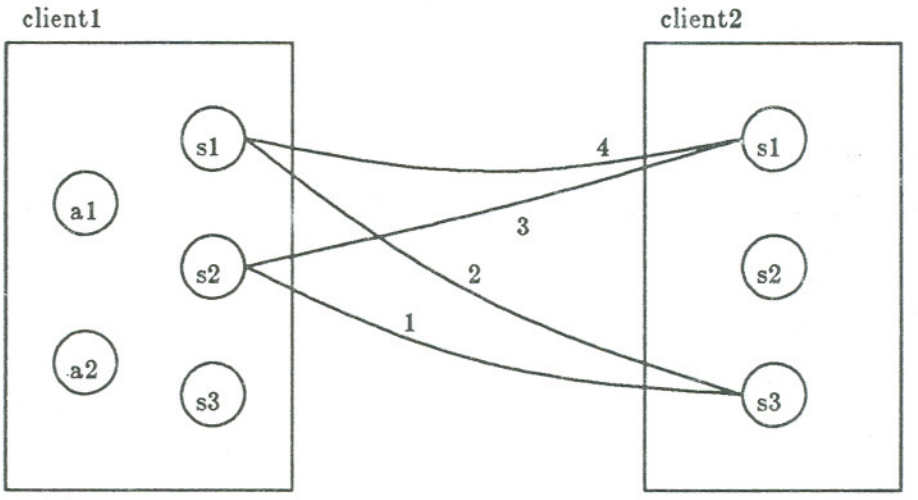
Implemented Utilities:

allowFiltersOn:	"Register an object."
allowFiltersOnSet:	"Register a set of objects."
constrain:of:and:of:with:	"Request a filter."
constrain:of:by:with:	"Request a filter."
disallowFiltersOn:	"Deregister an object."
getObject:ofProcess:	"Retrieve object."
objectsOfProcess:	"Retrieve object names."
otherRegisteredProcesses	"Retrieve process names."
release:and:of:	"Remove a filter."
updateObject:with:	"Update utility."

A.4 EXAMPLE APPLICATIONS

This example demonstrates the use of the FilterManager by two applications registered with the FilterManager as *client1* and *client2*. A network of filters is established among their objects. The files *FMCdemo.ws* and *FMC2demo.ws* contain the code for this demonstration to be used in the application processes registered as *client1* and *client2* respectively.

The following diagram shows the filter network used for Example 1; the filters are requested and released in numbered stages spanning both processes.



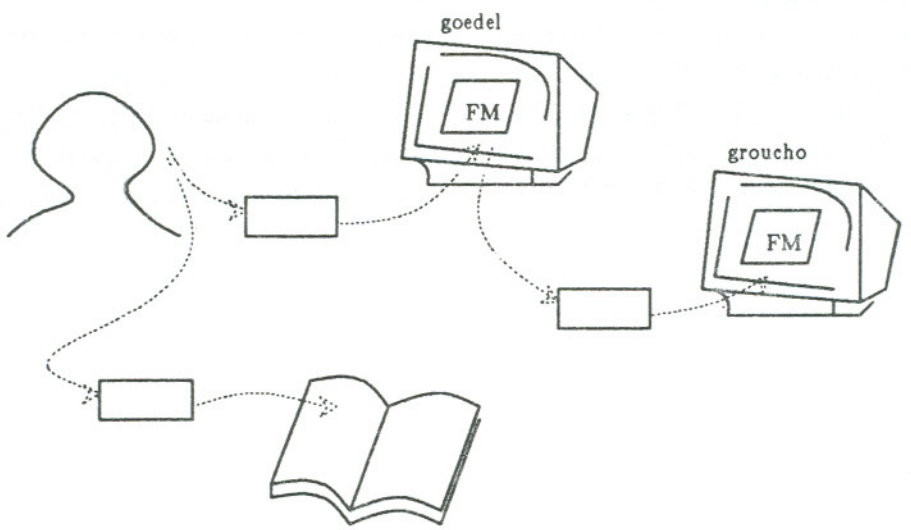

```
"INITIALIZATION - CLIENT1"

FilterManagerClient install.            "Process registration;  a name will be prompted for."

FMDict at: #s1 put: 'a first string'.   "Identify local variables."
FMDict at: #s2 put: 'workspace dictionary variable'.
FMDict at: #s3 put: 'also a FilterManager object'.

FM allowFiltersOnSet: (Set with: #s1 with: #s2 with: #s3).   "Register objects by name."
```

```
"UTILITIES"

"Registration and deregistration of objects."
FMDict at: #a1 put: (Set with: 'set one' with: 'set two' with: 'set three').
FMDict at: #a2 put: (Array new: 2 withAll: 'different type').
FM allowFiltersOnSet: (Set with: #a1 with: #a2).
FM allowFiltersOn: #s1.
FM disallowFiltersOn: #a1.

"Updates"
FMDict at: #s1 put: 'local variable update'.
(FMDict at: #s1) replaceFrom: 1 to: 14 with: 'process update' startingAt: 1.
FM updateObject: #s2 with: 'FilterManager update utility'.


"QUERIES"

FMDict.                                  "Local variables"
FM openObjects.                          "Object handlers"
FM otherRegisteredProcesses.             "Process names"
FM objectsOfProcess: #client2.           "Object names"
FM getObject: #s1 ofProcess: #client2.   "Object structure"


"NETWORK - EG 1"

(1)  FM constrain: #s3 of: #client2 by: #s2 with: EqualityFilter.   "Filter 1"
     FMDict at: #s2 put: 'one-to-one constraint'.

(3)  FM constrain: #s3 of: #client2 by: #s1 with: EqualityFilter.   "Filter 2"
     FMDict at: #s2 put: 's1 indirect from s2'.

(6)  FMDict at: #s1 put: 'head of a chain'.                         "Filter 3"

(8)  FM release: #s2 and: #s3 of: #client2.                         "Filter 1"
     FM release: #s2 and: #s1 of: #client2.                         "Filter 3"
     FMDict at: #s1 put: 'broken cycle'.


"TERMINATION"

FM shutdown.                 "Process deregistration."
```

```
"INITIALIZATION - CLIENT2"

FilterManagerClient install.

FMDict at: #s1 put: 'a first string'.
FMDict at: #s2 put: 'workspace dictionary variable'.
FMDict at: #s3 put: 'also a FilterManager object'.

FM allowFiltersOnSet: (Set with: #s1 with: #s2 with: #s3).

"NETWORK- EG 1"

(2)  FMDict at: #s3 put: 'source object in client2'.                "Filter 1"

(4)  FMDict at: #s3 put: 'root of tree'.                           "Filter 2"

(5)  FM constrain: #s2 of: #client1 by: #s1 with: EqualityFilter.  "Filter 3"
     FMDict at: #s3 put: 'show change in s1'.

(7)  FM constrain: #s1 of: #client1 by: #s1 with: EqualityFilter.  "Filter 4"
     FMDict at: #s3 put: 'fun with a cycle'.
     FMDict at: #s1 put: 'cycle around again'.

"NETWORK- EG 2"

"No type-checking is performed."
FM constrain: #a2 of: #client1 by: #s2 with: EqualityFilter.
FMDict at: #s2 put: 'no type-checking'.

"TERMINATION"

FM shutdown.
```

Epilogue

Ah ! If I had a version of the FilterManager with one-way filters that could handle brain-waves, how much simpler and more consistent my task would have been !



Biography

The author was born on 23rd Dec 1961 in Sabah, East Malaysia, but moved to Singapore at an early age for the benefit of an English education. She graduated with a B.Sc. from the National University of Singapore in 1983 and received the Data Processing Managers' Association prize. In her first appointment with Systems & Computer Organization, she was involved with the prototyping of an application model using various database tools for evaluation purposes. She was subsequently appointed systems programmer and administrator for an IBM system for almost two years. In February of 1986 she left S&C to try a career in applied research with the then Research Division of the Institute of Systems Science. Initially working with encoding schemes on write-once optical disks, she received a merit prize, the Tan Kah Kee Young Inventors' Award, for the scheme her project team had designed. Encouraged by her managers to pursue graduate studies in the U.S., she left her project team in the middle of the design and development of a multi-media database management system, and joined OGC in Sept 1987.