Distributed Filter Processes

Rushikesh K. Joshi

Department of Computer Science and Engineering, Indian Institute of Technology Bombay, Mumbai - 400 076, India

Key words: distributed systems; concurrency; distributed processes; filter processes; software evolution; trasparency

---

1 Introduction

A programming model for distributed filter processes is presented. The model is based on a transparent first class filter object model [2] [3]. It is supported for Distributed Processes [1] through a transparency extension. A filter process is a transparent process which can act on messages in transit. Besides, a filter process itself is a full fledged distributed process. A filter process can be injected in an existing network of distributed processes to accomplish evolution without having to change the existing client or server processes.

The extensions to Distribued Processes are described in the next section and subsequently, an example application of filter processes is provided. The application describes an evolution based on a stand-alone filter process modeling a new requirement on an existing distributed-process based solution. As a benchmark problem for illustration of the method of evolution, an existing solution to the *Readers and Writers* problem and an evolutionary requirement on it has been chosen from liturature [1]. The evolved solution satisfies an additional constraint not programmed in the initial solution, without modifying the existing client or server code through an intermediate transparent filter process.

2 Filter Processes

This section describes the extensions to the Distributed Processes that incorporate transparent filter processes into it. A filter relationship between two processes can be specified. The filtering process can transparently filter messages sent to its corresponding server process.

In the filter relationship specified below, process *aFilter* acts as a *filter process* for its corresponding *server process aServer*.

process aFilter: **filter** aServer;

A filter process can specify filter procedures to transparently filter invocations of their corresponding server procedures. Two types of filter procedures may be specified. An *upfilter procedure* filters a procedure invocation in an onward direction, i.e. before the invocation is dispatched to the intended server. Whereas, a *downfilter procedure* filters an outgoing result from the server process. The following code provides an example filter process sepcification that implements a transparent cache process for a server. Filter process *Cache* provides an upward filtering procedure to filter an invocation of procedure *getMarks* in upward direction, and a downward filtering procedure to filter the downward results. The upfilter procedure may bounce the reply to the caller from the cached entires upon cache hit. If a cache miss is encountered, the invocation is passed on as a default action. In this case, the donward filter procedure sees the result and updates the cache.

```
process Cache: filter Evaluator;
        ...
upfilter proc getMarksUp (id:int, #marks:int) filters getMarks
        ... bounce; ....
downfilter proc getMarksDown (#marks:int)
        filters getMarks
        ...
 process Evaluator;
        ...
proc getMarks (id:int, #marks:int)
        ...
```

The upfilter procedure can choose to pass on the request to the intended server or bounce the call back to the caller. A **bounce** statement specifies that the call may be returned to the caller without passing it on to the server. An absence of a bounce statement implies a *pass* action, which is default. If the bounce option is chosen, the output argument values as provided by the filtering process are returned. Especially note the correspondance between argument types of filter procedures and their corresponding server procedures.

3 Evolution through Stand-alone Filter Processes

This section describes a evolution method through transparent filter processes through an example of an evolutionary solution to a *readers and writers* problem. A program for the readers and writes problem using the Distributed Processes model as described by Hansen [1] has been chosen as an initial solution. A new constraint which is posed as an exercise problem in the same work is chosen as an evolutionary requirement.

A traditional solution to this problem would have consisted of a complete rewrite of the initial distributed process. The solution consisting of a stand-alone filter process is an on-the-fly solution that keeps the old distributed process functional besides injecting an intermediate transparent filter process in the message path. The additional constraints are satisfied without touching the existing server or the client side code.

Given below is a Process *Resource* that implements an initial solution to the *readers and writers* problem on which an evolutionary requirement is imposed. With the evolved solution, further reader requests need to be delayed as long as some writers are either waiting for or are using the resource.

process Resource;
s: int
proc StartRead **when** $s > 0 : s++$ **end**
proc EndRead **if** $s > 1 : s--$ **end**
proc StartWrite **when** $s == 1 : s--$ **end**
proc EndWrite **if** $s == 0 : s++$ **end**
$s = 1$

A filter process *Filt* may be specified to transparently filter procedure invocations on process *Resource* and so that the new requirement is satisfied collectively. In the solution listed below, process *Filt* provides the required semantics with the help of three procedures, two upfilters and a downfilter.

process Filt : **filter** Resource ;
int WWW
upfilter proc SW-Ufilter: $WWW++$; pass **end**
upfilter proc SR-Ufilter **when** WWW == 0 : pass **end**
downfilter proc EW-Dfilter : $WWW--$; **end**
WWW = 0

The filter process maintains a local variable *WWW* specifying the number of waiting or writing writers. The variable is incremented whenever a new writer enters the system. The upfilter for *StartWrite* always allows the write request to be passed on to process *Resource*. The write request may get blocked at the process *Resource* and eventually get serviced by its *StartWrite* procedure. The upfilter for procedure *StartRead* gets blocked at the filter object if there are any writers writing or waiting to write. The downfilter for procedure *EndWrite* ensures that the value WWW is decremented in the filter process whenever a call to *StartWrite* returns from process *Resource*.

In this way, the superimposed transparent filter process achieves the new functionality in the existing system without having to change the existing code. It is also possible to superimpose network of filter processes in an existing system of processes to evolve a collective behavior of the system.

4 Conclusions

A model of programming transparent filter processes is described. The Distributed Processes paradigm is extended to support filter processes. It is possible to evolve the behavior of an existing system by means of superimposition of filter processes in a system of distributed processes without having to change the existing server or client code.

References

[1] B. Hansen, Distributed Processes: A Concurrent Programming Construct, *Communications of the ACM*, vol. 21, n. 11, Nov. 1978, 934-941.

[2] R. K. Joshi, Vivekananda, D. Janaki Ram, *Message Filters for Object-Oriented Systems*, Software Practice and Experience, June 1997, v. 27, n.6, pp. 677-700.

[3] R. K. Joshi, Modeing with Filter Objects in Distributed Systems, Proceedings of the 2^{nd} Workshop on Engineering Distributed Objects (EDO 2000), Nov 2000, To appear in LNCS Vol no. 1999, pp. 176-181.