BATCHING: A Design Pattern for Efficient and Flexible Client-Server Interaction

Marta Patiño[1†] Francisco J. Ballesteros[2*] Ricardo Jiménez[1†]
Sergio Arévalo[1†] Fabio Kon[3‡] Roy H. Campbell[3§]

[1]Distributed Operating Systems Group. Technical University of Madrid
[2]Systems and Communications Group. Carlos III University of Madrid
[3]Systems Research Group. University of Illinois at Urbana-Champaign

Abstract

What do well-known techniques such as gather/scatter for input/output, code downloading for system extension, message batching, mobile agents, and deferred calls for disconnected operation have in common? Despite being rather different techniques, all of them share a common piece of design (and, possibly, implementation) as their cornerstone: the BATCHING design pattern.

All techniques mentioned above are designed for applications running across multiple domains (e.g., multiple processes or multiple nodes in a network). In these techniques, multiple operations are bundled together and then sent to a different domain, where they are executed. In some cases, the objective is to reduce the number of domain-crossings. In other cases, it is to allow for dynamic server extension.

In this paper, we present the BATCHING pattern, discuss the circumstances in which the pattern should and should not be used, and identify eight classes of existing techniques that instantiate it.

1 Introduction

Applications such as code downloading, message batching, gather/scatter and mobile agents follow the client-server model of interaction. A closer look reveals that all of them group a set of operations, and submit them to a server for execution. The submission of operations is aimed at reducing domain-crossings and/or allow dynamic server extension. For instance, code downloading into operating system kernels is

*Partially supported by Spanish CICYT grant # TIC-98-1032-C03-03.

†Partially supported by the Spanish Research Council CICYT grant # TIC-98-1032-C03-01 and by the Madrid Regional Research Council grant number CAM-07T/0012/1998.

‡Fabio Kon is supported in part by CAPES, Brazil, proc.# 1405/95-2.

§The Systems Research Group is supported by a grant from the National Science Foundation, NSF 98-70736.

intended to save domain-crossings and at the same time to allow system extension. Message batching and mobile agents are intended to save domain-crossings.

Consider a program using a file server like that of figure 1. On typical client-server interaction, the client sends a command (`read`, `write`) to the server, waits for the reply, and then continues.

```
cat(File aFile, File otherFile) {
while (aFile.read(buf))
  write(otherFile.write(buf));
}
```

Figure 1: Cat Code

Suppose that `read` and `write` are handled by the same server and that cross-domain calls (i.e. calls from client to server) are much heavier than calls made within the server. Then it would be much more efficient to send the whole `while` loop to the file server for execution.

Instead of having multiple cross domain calls (figure 2.a) a single one suffices if the client sends the code to the server for its execution (figure 2.b). To do so, it is convenient to extend the file server to allow the execution of programs submitted by different clients.

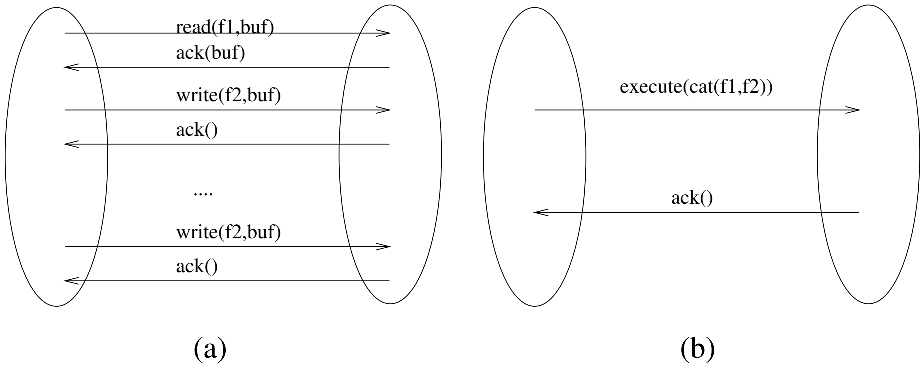

Figure 2: Interactions corresponding to read/write services and a cat service

2 The problem

Both cross-domain data traffic and cross-domain call latency have a significant impact on the efficiency of multi-domain applications. Cross-domain calls and cross-domain data transfers also happen on centralized environments. For instance, almost every operating system has a domain boundary between user space and kernel space (both entering and leaving the kernel requires a domain crossing). An application using multiple processes has a domain boundary between every two of its processes. Besides, in a distributed system, the network behaves as a domain boundary.

Under many circumstances, unnecessary data transfers occur just because the object controlling the operation resides far from the data source and/or the data sink. That is precisely what happens in the file copy example in the previous section: the client object performing the copy and the file server objects were placed at different domains. Thus, data came to the client just to go back to the server.

3 The solution

By batching separate method calls, i.e. transforming them into a single cross-domain call, one can avoid unnecessary data copying and reduce the number of cross-domain calls.

Clients can build a program (a "batch call") and transfer it to the server at once. The program performs multiple operations on that server even though the client had to send it only once.

In our example (see figure 2, the interactions for cat), if BATCHING is not used, the file content has to travel twice across the network. When a `cat` program is submitted to the server, however, the file does not leave the server, it is copied locally.

4 Pattern structure

BATCHING, also known as COMPOSITECALL.

4.1 Participants

The class hierarchy corresponding to the BATCHING pattern is shown in figure 3. It follows the OMT notation [13] variant used in [5].

`BatchServer` behaves as a façade [5] to services provided by the server. An object of this class is located on the server side. It supplies interpretation facilities to service callers, so that clients can send a program to the server side instead of making direct calls to the server. The `execute` method is an entry point to the interpreter [5], which interprets the "batch" program and returns its results to the client.

`ConcreteServer` This class is only present on the server side. It provides the set of entry points that can be called by the client.

Note that the `ConcreteServer` is actually the class (or the set of classes) one has in the server side before instantiating the pattern. It is mentioned here for completeness.

`Program` is an abstract class that represents the program to be interpreted. Clients build `Program` instances and send them to the `BatchServer` for execution. It is also responsible for maintaining an associated table of variables. The `run`

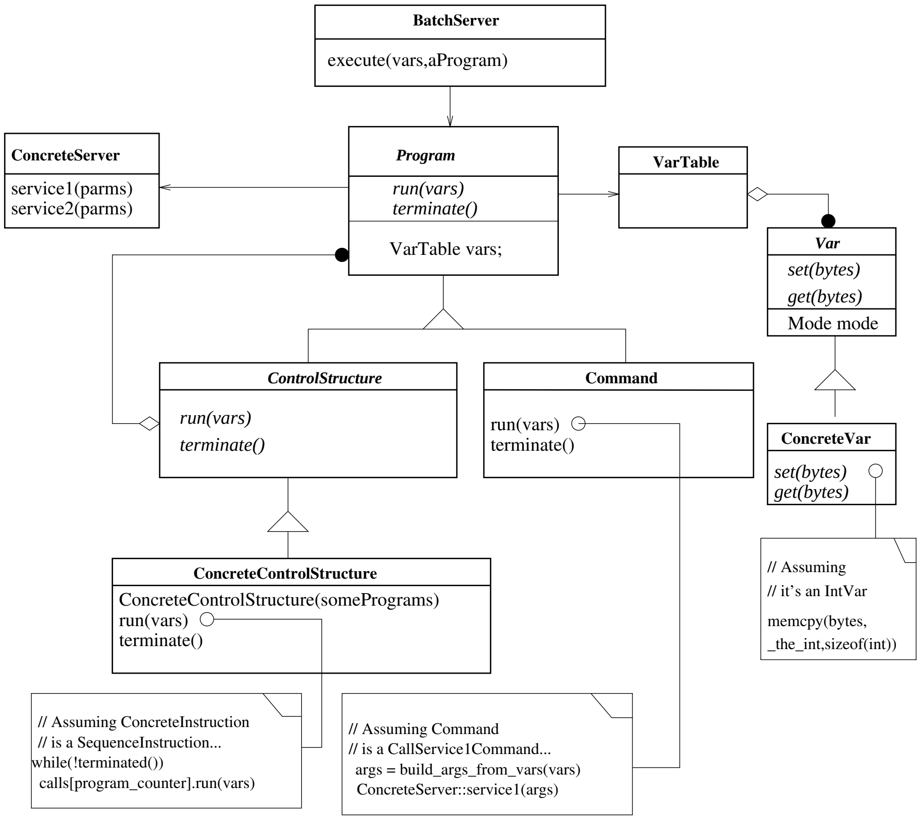

Figure 3: BATCHING

method of a `Program` class implements the interpreter needed to run it on the server.

The `Program` is also responsible for performing an orderly program termination when an error occurs. The `terminate` method is provided as an abstract interface for program termination.

The alternate name for BATCHING, namely, COMPOSITECALL, comes from the fact that `Program`, together with the next couple of classes, is an instance of the COMPOSITE pattern [5].

`ControlStructure` is a construct made of `Programs`. Its purpose is to bundle several `Programs` together according to some control structure (e.g. sequence, iteration, etc.).

`ConcreteControlStructures` represent concrete control structures like conditionals, `while` constructs, `sequences`, etc. At the server side, this class is responsible for executing the concrete control structure represented by the class.

`ConcreteControlStructure` constructors can be used at the client side to build complex `Programs`.

`Command` is a `Program` which represents a single operation. (It resembles the COMMAND pattern shown in [5], hence the name). Examples of concrete `Commands` can be arithmetic operations, logic operations, or calls to `ConcreteServer` entry points. The only purpose of BATCHING is to bundle several concrete `Commands` together using `ConcreteControlStructures`.

`VarTable` keeps the variables of the `Program`. It provides local storage and also holds any input parameter for the program. Output values from the program are also kept within the `VarTable`. The table is built at the client using the set of input parameters for the `Program`. Then, it is used within the server, while the `Program` is interpreted. The table is finally returned back to the user after completion of the `Program`.

There is a variable table per `Program` (pairs of `VarTable` and `Program` are sent together to the `BatchServer`). Thus, all components of a concrete `Program` share a single variable table so that they can share variables.

`Var` is an abstract class representing a variable of the program sent to the server. It has some associated storage (bytes, in the figure). `Var` instances are kept within a `VarTable`. Variables have a *mode*, which can be either `in` (parameter given to the `Program`), `out` (result to be given to the user), `inout` (both), or `local` (local variable). By including the `mode` qualifier, this class can be used for local variables as well as for input/output parameters.

`ConcreteVar` is a variable of a concrete type (integer, character, etc.). Its constructor is used at the client to declare variables or parameters to be used by the `Program`. At the server side, instances of this class are responsible for handling single, concrete pieces of data used by the program.

4.2 The pattern applied to a file server

The concrete structure of classes for our file server example is shown in figure 4. Intuitively, this BATCHING instance simply adds an interpreter (see the INTERPRETER pattern in [5]) to the file server. That interpreter can execute programs that (1) call `read` and `write` and (2) can use `while` as a control structure.

We took as an starting point the `FileServer` class, which provides both `read` and `write` methods to operate on a file.

We simplified the typical interface provided by a file server. A typical file server would contain several `File` objects that would supply `read` and `write` methods. To illustrate the pattern in a simple way, we omitted the file being used[1].

The `BatchFileServer` is co-located with the `FileServer`, providing a new `execute` service that supplies an interpreted version of `FileServer` services. The `BatchFileServer` corresponds to the `BatchServer` in the pattern (see the pattern diagram in figure 3).

[1]Obtaining a complete implementation is a matter of adding a `File` class and adding `file` parameters to the `read` and `write` methods.

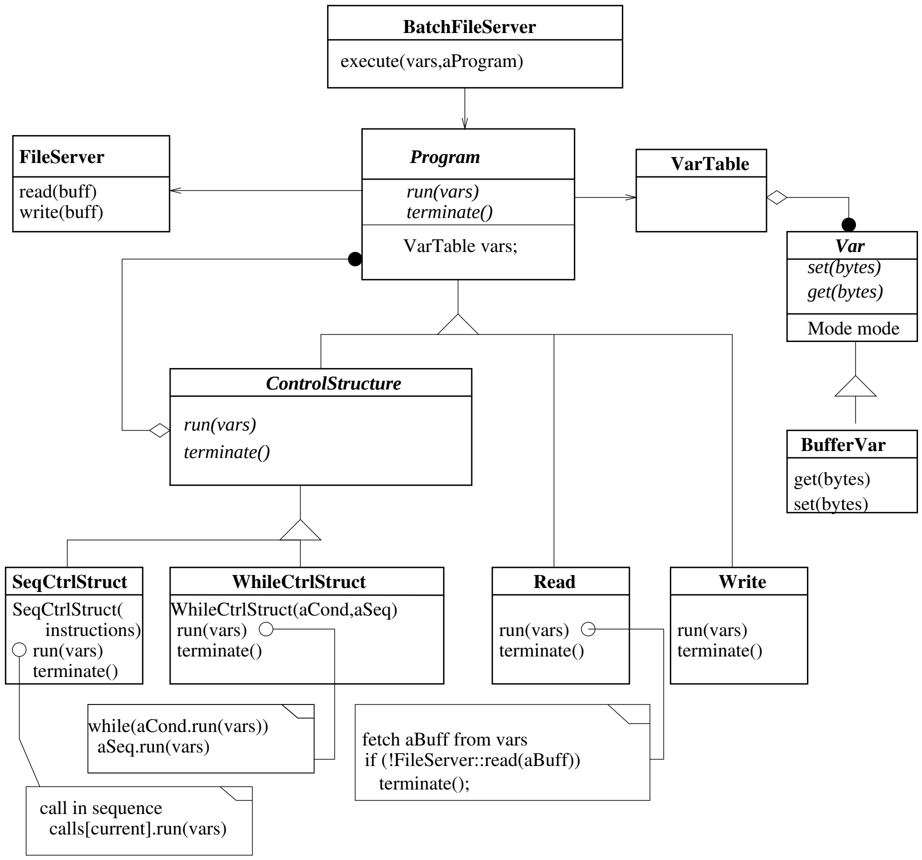

Figure 4: File server BATCHING

The `BatchFileServer` accepts a `Program`, which is built in terms of `ControlStructures` and `Read` and `Write` commands.

To execute

```
while (read(buf))
  write(write(buf));
```

the `Program` sent to the `BatchFileServer` must be made of a `WhileCtrlStruct`, using a `Read` as the condition. The body for the `WhileCtrlStruct` must be a sequence made of a single `Write` command.

Here, `WhileCtrlStruct` and `SeqCtrlStruct` correspond to `ConcreteControl-Structures` in the pattern. `Read` and `Write` match `Commands` in the pattern. The buffer used in the read and write operations is handled by a `BufferVar` class instance, which corresponds to a `ConcreteVar` in the pattern.

A client can build a program (accessing the file server) by using constructors provided by `WhileCtrlStruct`, `SeqCtrlStruct`, `Read`, and `Write`. This batched call can be submitted later, by the client, to the `execute` method of the `BatchFileServer`.

5 Dynamics

When a program is received at the server side, it is deserialized. The interaction that follows is as shown in figure 5.

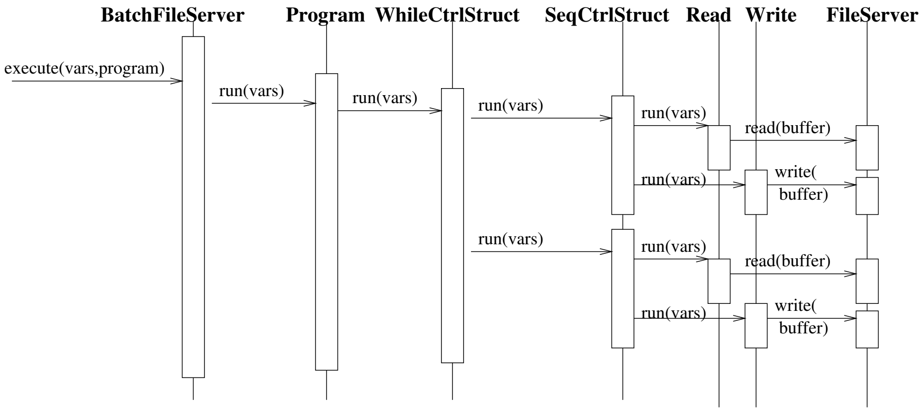

Figure 5: Interaction diagram for a `cat` program.

6 Implementation issues

We now discuss two important aspects of using BATCHING: how to build programs for BATCHING and what to do if they fail.

6.1 Composing programs

Programs are made out of statements and variables. In a BATCHING `Program`, each statement corresponds to a `ConcreteControlStructure` or concrete `Command`. Variables are instances of a `ConcreteVar` class. To build a program, clients declare an object of the `Program` class and invoke its constructor method.

Ideally, the client side for an instance of BATCHING would be exactly like the code of a client making direct calls to the server; i.e. like a client not using BATCHING at all. In practice, `ConcreteControlStructure` constructors (which are functions) are used. Thus, code in the client for a `Program` looks like the code that the user would write without using the pattern. `Command` objects are not declared; they are built with functional constructors.

Revisiting our example, the code for the `cat` program is shown in figure 6. In the figure, constructors are functions that build objects within the program. In this example, `SeqCtrlStruct` and `WhileCtrlStruct` are `ConcreteControlStructures` of the language. `Open`, `Close`, `Read`, and `Write` are classes derived from `Program` and

```
VarTable vars;
Program program;
IntVar f1(vars, local), f2(vars, local);
CharVar car(vars, local);

Program= SeqCtrlStruct((
      Open(f1, StringLit("name1")),
      Open(f2, StringLit("name2")),
      WhileCtrlStruct(Read(f1, car),
         Write(f2, car)),
      Close(f1),
      Close(f2)
   ));
   execute(program, vars);
```

Figure 6: Program for Cat

clients invoke their constructors to let the `Program` issue calls into the server. Program variables are stored in the `vars` variable table. In this case, `f1`, `f2` and `car` are local variables, so their mode is `local`.

One of the problems of submitting the client code to the server is what happens when a call fails. The server programmer knows when a server call has failed, so he or she can decide to terminate the program, in that case. This can be done by calling the `terminate` method of the `Program` class from a `run` method. However, the client could wish to continue the program despite any failures. To support this, we have included two commands in our pattern instances: `AbortOnError` and `DoNotAbortOnError`. They allow the user to switch between the two modes. When `AbortOnError` has been called, a call to `terminate` causes program termination; otherwise it has no effect. In this way, the client can control the effects of a failed call.

The implementation of `terminate` depends on both the kind of instruction set being implemented and on the implementation language. A byte-code based program can be stopped very easily as there is a main control loop (in the `run` method), just by setting a `terminated` flag to true. Stopping an structured program (e.g. the one used in our file server example) is a little more complicated. This is due to recursive interpretation: calls to `run` in `Programs` propagate calls to the `run` method of its components. To stop that program, it is necessary to finish all the nested `run` calls. Depending on the implementation language it can be done in a way or another. In a language with exceptions, like C++ or Ada, it suffices to raise and propagate an exception in the code of `Terminate`, catching it in the `run` code of `Program`. In languages like C, `setjmp` can be used in the top-level `run` method before calling any other `run`, and `longjmp` can be used, for the same purpose, in the body of `terminate`.

7 Consequences

The pattern has the following benefits:

1. *It provides an Abstract machine view of the server*. When using BATCHING, clients no longer perceive servers as a separate set of entry points. Servers are now perceived as *abstract machines*. Their instruction set is made of the set of server entry points, together with some general-purpose control language.

   Therefore, it is feasible for users to reuse programs for different BATCHING calls. Programs that interact with the server can be built, and reused later.

2. *It reduces protection-domain crossings*, as the `cat` program did above. If this is the main motivation to use the pattern, domain crossing (client/server invocation) time must be carefully measured. Whenever complex control structures are mixed with calls to the server, or when client computations need to be done between successive calls, the pattern might not pay.

   In any case, the time used to build the program must be lower than the time saved in domain crossing. The latter is approximated by the difference between the time to perform a cross-domain call and the time to interpret and dispatch a server call.

3. *It reduces the number of messages* between clients and servers; provided that the `Program` issues repeated calls to the server and the control structure is simple enough.

   Again, the improvement due to the reduced number of messages can be lower than the overhead due to program construction and interpretation. Therefore, careful measurement must be done prior to pattern adoption.

4. *It decouples client/server interaction from the call mechanism.* BATCHING provides a level of indirection between the client and the server. The client can perform a call by adding commands to a `Program`; while the `Program` can be transmitted to the server by a means unknown to the client.

5. *It decouples client calls from server method invocations.* As said before, a client can perform calls by adding commands to a `Program`. The resulting `Program` can be sent to the server at a different time. Therefore, there is no need for the client and the server to synchronize in order for the call to be made.

The pattern has the following drawbacks:

1. *Client requests might take arbitrary time* to complete. A batched program might lead to a nonterminating program. If server correctness depends on bounded client requests, it may fail. As an example, a server can use a single thread of control to service all client requests. Should a `Program` not terminate, the whole server would be effectively switched off by a single client.

In such case, either avoid using BATCHING, or handle multithreading issues as a side-effect (i.e., arrange for each `Program` to use its own thread, using a giant lock[2] to protect non MT-safe servers[3].)

2. *Security can be compromised.* The more complex the command set, the more likely the server integrity can be compromised due to bugs in the command interpreter. If *high* security is an issue, either avoid BATCHING or reduce the complexity of the command set to the bare minimum.

3. *It might slow down the application.* When cheap domain crossing is available and efficiency is the primary objective, using BATCHING might slowdown the application if the time saved on domain crossings is not enough to compensate the overhead introduced by BATCHING.

8 Related patterns

Both `Program` and `ControlStructure` rely on instances of the INTERPRETER pattern [5]. Indeed, the interpreter of a `Program` is behind its `run` method.

`Program`, `ControlStructure`, and `Commands` make up an instance of the COMPOSITE pattern [5]. Composite programs, such as `Sequence` and `Conditional`, are aggregates of `Assignments`, `ServerCalls`, and other primitive commands.

If an instruction set for a BATCHING language is to be compiled, `Program` might include a method to compile itself into a low-level instruction set. Moreover, `Programs` should be (de)serialized when transmitted to the server. Once in the server, they can be verified for correctness. All these tasks can be implemented following the VISITOR pattern [5].

A server call issued within a `Program` might fail or trigger an exception. If that is the case, the whole `Program` can be aborted and program state transmitted back to the client—so that the client could fix the cause of the error and resume `Program` execution. The MEMENTO pattern [5] can encapsulate the program state while in an "aborted" state. As said before, such program state can be used to resume the execution of a failed program (e.g. after handling an exception.)

MEMENTOs can also be helpful for (de)serializing the program during transmission to the server.

As a program can lead to an endless client request, single threaded or a-request-at-a-time servers can get into trouble. To accommodate this kind of server so that BATCHING could be used, the ACTIVEOBJECT [8] and the RENDEZVOUS [7] patterns can be used.

COMPOSITEMESSAGES can be used to transfer the `Program` from the client to the server. The COMPOSITEMESSAGES pattern [14] applies when different components must exchange messages to perform a given task. It allows grouping several messages together in an structured fashion (it does with messages what BATCHING does with server entry-points). In that way, extra latency due to message delivery can be avoided,

[2] A single lock protecting the entire server. It must be gained prior to any server call and released right after every server call. `Program` instructions not calling the server can execute without locking the server.

[3] Servers not prepared to handle concurrent requests.

and components decoupled from the transmission medium. The main difference is that BATCHING is targeted at the invocation of concrete server-provided services, not at packaging data structures to be exchanged.

Last but not least, COMPOSEDCOMMAND [16] is similar to BATCHING in that it bundles several operations into a single one. However, BATCHING is more generic in spirit.

9 Known uses

Our experience with BATCHING started when we noticed that a single piece of design had been used to build systems we already knew well. Then we tried to abstract the core of those systems, extracting the pattern. Once we identified the pattern, we tried to find some *new* systems where it could be applied to obtain some benefit. We did so [1] and obtained substantial performance improvements.

For us, this pattern has been a process where we first learned some "theory" from existing systems and then applied what we had learned back to "practice." In this section we show how existing systems match the pattern described in the previous sections—hopefully, this will allow a better understanding of the pattern, as happened in our case. We also include a brief overview of the two systems where we applied the pattern ourselves with *a priori* knowledge of the pattern.

Note that the BATCHING design allows a single implementation of the pattern to handle some of the various applications described below. As the server is specified every time a `Program` runs, the same piece of code could perfectly handle most of the applications shown below. Nevertheless, existing systems, built without *a priori* knowledge of the pattern hardly share the common code needed to implement applications described below (e.g. gather/scatter is always implemented separately from message batching facilities, when both are provided.)

Operating System extensions by code downloading into the kernel (like in SPIN [4] and μChoices [9]) can be considered to be an instance of this pattern. These systems use code downloading as the means to extend system functionality. The mechanism employed is based on defining new programs, which are expressed in terms of existing services.

In this case the `Program` is the extension performed; the set of `Concrete-ControlStructures` depends on the extension language; and the `run` method is implemented either by delegation to the extension interpreter or by the native processor (when binary code can be downloaded into the system.)

Agents. An agent is a program sent to a different domain, which usually moves from one domain to another [12]. The aim is to avoid multiple domain crossings (or network messages) and also to allow disconnection from the agent home environment.

Programs built using BATCHING are meant to stay at the server until termination, and they possess no `go`[4] statement. However, BATCHING already includes most of the machinery needed to implement an agent system. A `go` statement could be provided by the command language itself.

Gather/Scatter I/O. Gather/Scatter input/output is yet another example where this pattern appears. In gather/scatter I/O a list of input or output descriptors is sent to an I/O device in a single operation. Each descriptor specifies a piece of data going to (or coming from) the device. Data written is gathered from separate output buffers. Data read is scattered to separate input buffers. Its major goal is to save data copies.

In this case, the program is just the descriptor list, where each descriptor can be supported by a `Command`. The program `run` method iterates through the descriptor (i.e., command) list and performs the requested I/O operations. The services (i.e., commands) are simply `Read` and `Write`.

Note how by considering this pattern, gather/scatter I/O could be generalized so that the I/O device involved does not need to be the same for all descriptors sent by the user. Moreover, separate `Read` and `Write` operations could be bundled in a single one.

Improving latency in Operating Systems. Many user programs happen to exhibit very simple system call patterns. That is an opportunity for using BATCHING to save domain crossings and, therefore, execution time.

As a matter of fact, we have done so by instantiating BATCHING for two systems, Linux and *Off*++ [2]. In both systems, we obtained around 25% speedups for a `cat` program written with BATCHING [1].

We implemented two new domain-specific languages (i.e., `ControlStructures` and `Command` sets) that allowed users to bundle separate calls into a single one, like in the `cat` example of section 1.

The first language we implemented was based on byte-codes. We included just those commands needed to code loops, conditional branches, and simple arithmetic. This language was used both on Linux and *Off*++.

The second language we implemented was a high-level one, designed specifically for *Off*++. It includes just the commands needed to `Repeat` a given operation `n` times and to perform a `Sequence` of operations.

10 Acknowledgments

We are sincerely grateful to our shepherd, Frank Buschmann, and to John Vlissides whose help was invaluable; this paper owes much to them. We are also grateful to the members of the "Allerton Patterns Project" group of PLoP'99 for their feedback on the pattern.

[4] The `go` instruction is typical on Agent systems and is meant to trigger the migration of an agent to a different location.

References

[1] F. J. Ballesteros, R. Jiménez-Peris, M. Patiño-Martínez, F. Kon, S. Arévalo, and R. H. Campbell. Using Interpreted CompositeCalls to Improve Operating System Services. Submitted for publication, 1998. Also available in http://www.gsyc.inf.uc3m.es/off/interp-os.ps.

[2] Franscisco J. Ballesteros, Fabio Kon, and Roy H. Campbell. A Detailed Description of Off++, a Distributed Adaptable Microkernel. Technical Report UIUCDCS-R-97-2035, Department of Computer Science, University of Illinois at Urbana-Champaign, August 1997.

[3] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, 1987.

[4] B.N. Bershad, S. Savage, P. Pardyak, E.G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*. ACM, December 1995.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Object-Oriented Software*. Addison-Wesley, 1995.

[6] J. Gray. *Operating Systems: An Advanced Course*. Springer, 1978.

[7] R. Jiménez-Peris, M. Patiño-Martínez, and S. Arévalo. Multithreaded Rendezvous: A Design Pattern for Distributed Rendezvous. In *Proc. of ACM Symposium on Applied Computing.* ACM Press, Feb. 1999.

[8] R. Greg Lavender and Douglas C. Schmidt. Active object – an object behavioral pattern for concurrent programming. In *Proceedings of the Second Pattern Languages of Programs conference (PLoP).*, Monticello, Illinois, September 1995.

[9] Y. Li, S. M. Tan, M. Sefika, R. H. Campbell, and W. S. Liao. Dynamic Customization in the μChoices Operating System. In *Proceedings of Reflection'96*, San Francisco, April 1996. Reflection'96.

[10] B. Liskov. Distributed Programming in Argus. *Communications of the ACM*, 31(3):300–312, Mar. 1988.

[11] Ajay Mohindra, Apratim Purakayastha, Deborra Zukowski, , and Murthy Devarakonda. Programming Network Components Using NetPebbles: An Early Report. In *Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems*, Santa Fe, New Mexico, April 1998. USENIX.

[12] P.E.Clements, Todd Papaioannou, and John Edwards. Aglets: Enabling the Virtual Enterprise. In *Proc. of the Managing Enterprises - Stakeholders, Engineering, Logistics and Achievement Intl. Conference (MESELA '97)*, Loughborough University, UK, 1997. Also available in http://luckyspc.lboro.ac.uk/Docs/Papers/Mesela97.html.

[13] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorenson. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.

[14] Aamod Sane and Roy Campbell. Composite Messages: A Structural Pattern for Communication between Components. In *OOPSLA '95 workshop on design patterns for concurrent, parallel, and distributed object-oriented systems*, 1995.

[15] S. K. Shrivastava, G. N. Dixon, and G. D. Parrington. An Overview of Arjuna: A Programming System for Reliable Distributed Computing. *IEEE Software*, 8(1):63–73, Jan. 1991.

[16] Jeniffer Tidwell. Interaction patterns. In *Proceedings of PLoP98*, 1998.