Batching: A Design Pattern for Efficient and Flexible Client/Server Interaction

Francisco J. Ballesteros[1], Fabio Kon[2], Marta Patiño[3], Ricardo Jiménez[3], Sergio Arévalo[1], and Roy H. Campbell[4]

[1] University Rey Juan Carlos
nemo@lsub.org
[2] University of São Paulo
kon@ime.usp.br
[3] Technical University of Madrid
[4] University of Illinois at Urbana-Champaign

Abstract. The BATCHING design pattern consists of a common piece of design and implementation that is shared by a wide variety of well-known techniques in Computing such as gather/scatter for input/output, code downloading for system extension, message batching, mobile agents, and deferred calls for disconnected operation.

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 enable dynamic server extension.

In this article, 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 aims at reducing domain-crossings and/or enable dynamic server extension. For instance, code downloading into operating system kernels intends to save domain-crossings and, at the same time, enable system extension. Message batching and mobile agents intend to save domain-crossings.

Consider a program using a file server such as the one in Figure 1. In a typical client/server interaction, the client sends a command (`read` or `write`) to the server, waits for the reply, and then continues.

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

J. Noble and R. Johnson (Eds.): TPLOP I, LNCS 5770, pp. 48–66, 2009.

Research partially supported by Spanish Research Council grant TIN-2007-67353-C02-02

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

Fig. 1. File copy code

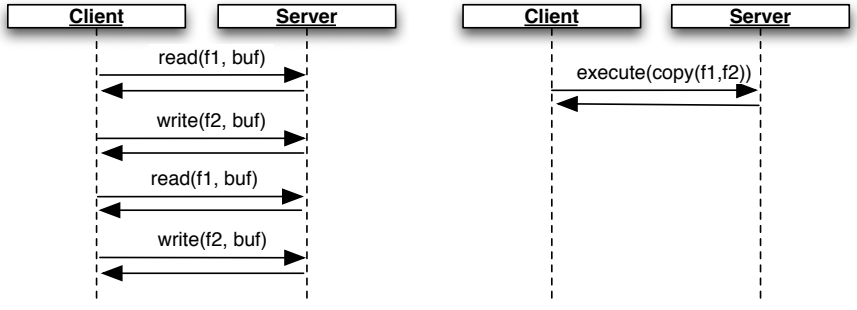

Fig. 2. Interactions corresponding to read/write services and a copy service

Instead of having multiple cross-domain calls, as depicted in the left-hand side of Figure 2, a single call suffices if the client sends the code to the server for execution, as shown in right-hand side of Figure 2. To do so, it is necessary to extend the file server to support the execution of programs submitted by different clients.

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.

The line separating two different domains has to be considered while designing the application. There are two main issues causing problems to any application crossing the line: data movement and call latency.

Within a protection domain (e.g., a Unix process), an object can pass data efficiently to any other object. For passing a large amount of data, a reference can be used. However, whenever an object has to pass data to another object in a different domain, data has to be copied. Although some zero-copy networking frameworks avoid data copying within a single node in a network, data still has to be "copied" through the network in distributed applications.

In many domains, such as file systems and databases, data movement can be the major performance bottleneck. Therefore, avoiding unnecessary data transfer operations may be crucial. Under many circumstances, unnecessary data transfers occur just because the object controlling the operation resides far from the data source or 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 in different domains. Thus, data came to the client just to go back to the server.

Another issue is call latency. A call between two objects residing in different domains takes much more time to complete than a typical method call within a single domain. The reason is simply that a domain boundary has to be crossed; that usually involves either crossing the operating system kernel interface (in a single node), network messaging (in a distributed environment), or both. Therefore, avoiding domain crossing when performing calls is crucial for performance. Any solution reducing the number of domain crossings can make the application run faster.

When designing a solution, it should be taken into account that, under certain circumstances (e.g., when *inexpensive domain crossing* is available and efficiency is the primary objective), the overhead introduced to solve the problem might actually degrade performance. However, even when cheap domain crossing is available, the overhead caused by cross-domain data transfers (e.g., copying data or sending messages over a network) might still cause a performance problem.

Any solution must take into account carefully what is the real penalty caused by data copying and call latency. Also, this solution should be employed only when the overhead it causes is small compared to the penalties it avoids.

3 The Solution

Batching, also known as CompositeCall.

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 copy), if Batching is not used, the file content has to travel twice across the network. When a `copy` program is submitted to the server, however, the file does not leave the server, it is copied locally. It behaves as if we had extended the server functionality dynamically by adding support for a `copy` operation.

4 Pattern Structure

In the following, we present the general structure of the pattern in its more complete and sophisticated form. Specific instances of the pattern often apply simplified implementations of the pattern. In Section 4.2, we describe the application of the pattern to the file service domain.

4.1 Participants

The class hierarchy corresponding to the BATCHING pattern is shown in Figure 3.

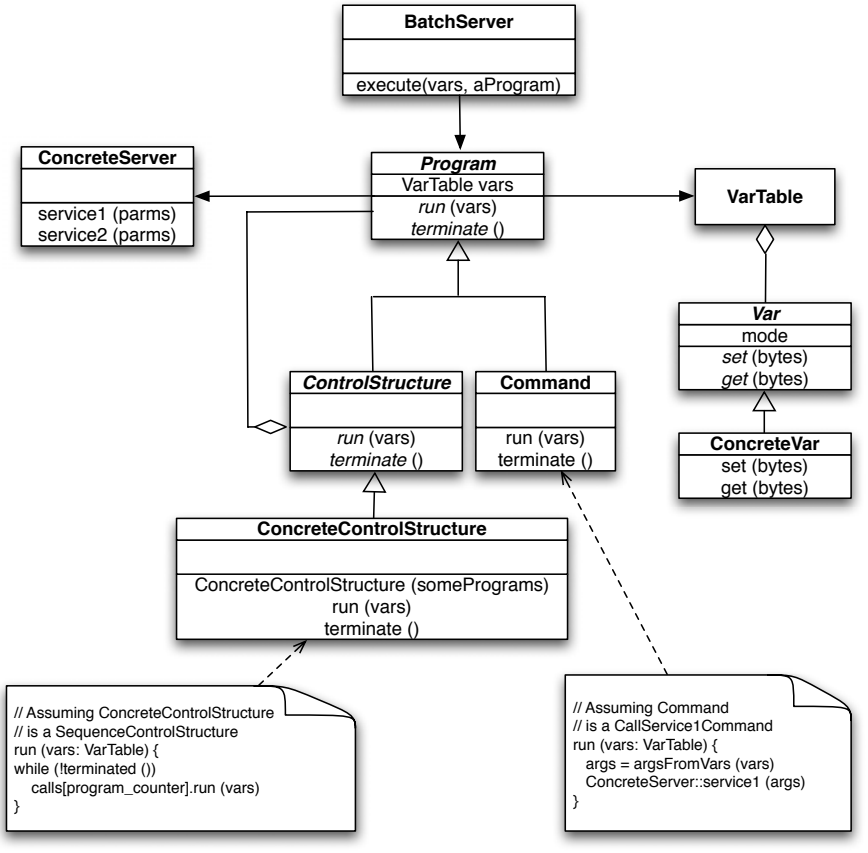

Fig. 3. BATCHING

`BatchServer` behaves as a FAÇADE [GHJV95] to services provided by the server. An object of this class is located on the server side. It supplies interpretative 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 [GHJV95], which interprets the "batch" program and returns its results to the client.

`ConcreteServer` 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 on the server side before instantiating the pattern. It is mentioned here for completeness.

`Program` is an abstract class that represents the batched calls or 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` method of a `Program` class implements the interpreter needed to run it on the server.

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

An alternate name for BATCHING is COMPOSITECALL since `Program` and the following couple of classes form an instance of the COMPOSITE pattern [GHJV95].

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

`ConcreteControlStructures` represent structures such as conditionals, `while` constructs, `sequence`s, 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 `Program`s.

`Command` is a `Program` that represents a single operation; it resembles the COMMAND pattern shown in [GHJV95], hence the name. Examples of concrete `Command`s are arithmetic operations, logic operations, or calls to `ConcreteServer` entry points. The only purpose of BATCHING is to bundle several concrete `Command`s together using `ConcreteControlStructure`s.

`VarTable` stores the variables (i.e., the state) 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 Figure 3). `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.

Note that in a concrete instance of the pattern, the `Program` (and related classes) may differ from the ones shown. That is, the core of the pattern includes `BatchServer`, `Program`, and `ConcreteServer` classes; but other classes

shown here may differ depending on how the pattern is instantiated. The structure shown here is a general form for the pattern and can be directly applied to any particular implementation. However, to solve a concrete problem where the pattern applies, this structure can be modified and a `Program` class might be implemented in quite different ways depending on the particular form for the "program" sent from the client to the server. Doing so would also require changes to `BatchServer` because it must be able to process the program.

For example, all classes used by a client to build a program might be compiled to a bytecode representation of the program, to be sent to the server. In this case the `BatchServer` would be actually a bytecode interpreter. As another example, serialized forms of the program structures shown in this paper could be directly sent to the server, and `BatchServer` would simply use them after unpacking to interpret the program. More radical examples exist, such as making `Program` use a textual representation of the program and `BatchServer`, a compiler or

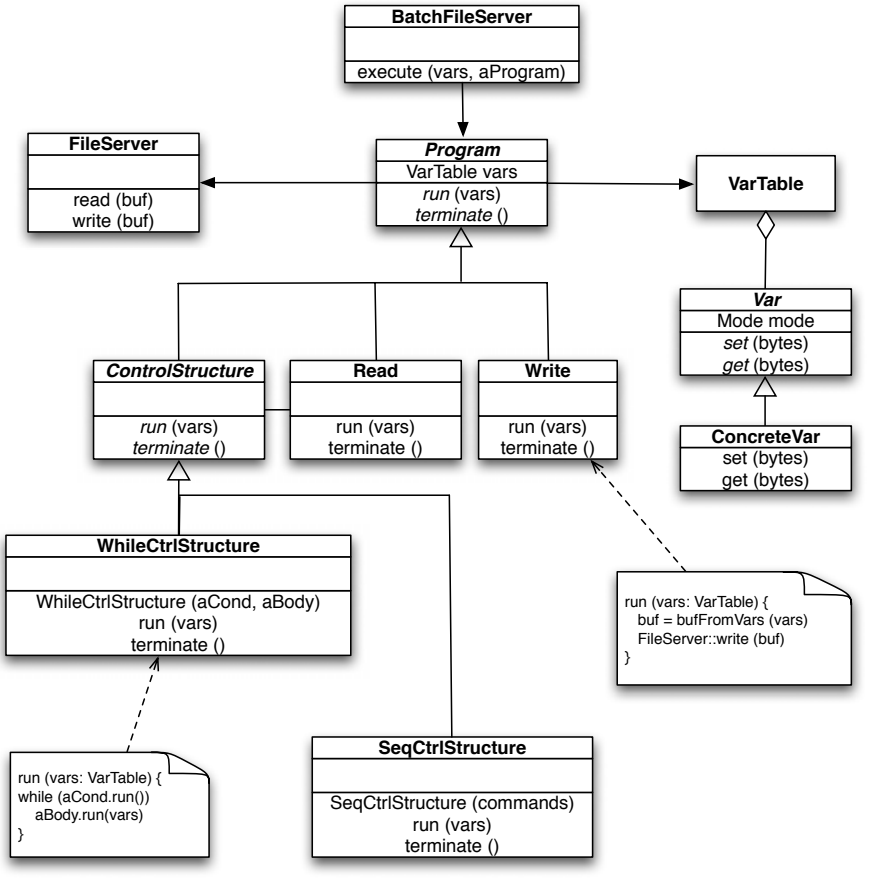

Fig. 4. Batching File server

interpreter for such "source"; or yet making `Program` use a binary representation of the program and implementing `BatchServer` simply as a call to the binary code received from the client.

All this said, in general, the structure shown for the pattern suffices and works well to solve the problem addressed by this pattern. In what follows, we use it in all our examples, but one should be aware that there are other useful variants of the pattern that use other forms of programs.

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 [GHJV95]) 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 a starting point the `FileServer` class, which provides both `read` and `write` methods that 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).

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

To execute

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

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

Here, `WhileCtrlStructure` and `SeqCtrlStructure` correspond to `ConcreteControlStructure`s in the pattern. `Read` and `Write` match `Command`s in the pattern. The buffer used in the read and write operations is handled by a `BufferVar` class instance, corresponding to a `ConcreteVar` in the pattern.

A client can build a program (accessing the file server) by using constructors provided by `WhileCtrlStructure`, `SeqCtrlStructure`, `Read`, and `Write`. The client can later submit this batched call to the `BatchFileServer` `execute` method.

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

5 Dynamics

The client builds a program (a "script" of commands) and sends it to the server, which interprets it. When the server receives a program, it first deserializes it. The interaction that follows is shown in Figure 5.

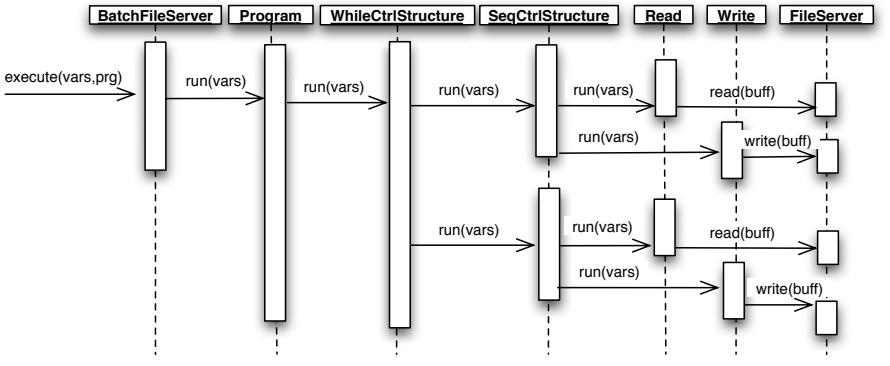

Fig. 5. Interaction diagram for a `copy` program

A `BatchFileServer` object at the server side is in charge of interpreting client programs. When its `execute` method is called, a program and a table of variables must be supplied. The `execute` method calls the `run` method of the program providing the table of variables; this method interprets the "batch" program. Once `execute` finishes, the results are returned.

The actual dynamics resulting from executing the "batch" program depends on the actual form used for the program. In our suggested form, the `run` method of the `Program` class implements recursive interpretation. When the program has been interpreted, that is, the `run` method has finished, the results of the program execution are still in the variable table. As part of the `execute` method, the table is serialized and sent back to the client.

Figure 5 shows the interaction diagram for our example copy batch program (with calls to `Open` and `Close` suppressed for the sake of simplicity). The `run` method of the `Program` calls the `run` method of the `ConcreteControlStructure` representing the program (the `WhileCtrlStructure` in the interaction diagram). `ControlStructure`s provide a `run` method to interpret themselves. That is, a program has built-in its own interpreter, an instance of the INTERPRETER pattern [GHJV95]. So, the `While` command calls the `run` method of its inner component (`SeqCtrlStructure` in the interaction diagram for copy).

6 Implementation Issues

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

6.1 Composing Programs

How to build a program depends on the structure used for it. As an example, we use in this section the form suggested for programs in the previous description of the pattern. Readers should be aware, however, that other forms of implementing the pattern do exist.

Programs are made 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.

To support the usage of expressions within the `Program`, subclasses inheriting from an `Expr` class can be provided (see Figure 6). `Expr` is a functional service representing an expression, and can be used as a function within expressions.

Program variables are stored in a table. They contain initial values as well as intermediate values and results of the program execution at the server side. To build that table, the programmer of the client must declare an object of the `VarTable` class. When variable objects are instantiated, they are constructed and stored in that table with an initial value, if any, and their mode, that is, `in`, `out`, `inout`, or `local`. When a variable table is sent to the server, only values of `in` and `inout` variables have to be copied to the server. After the execution of

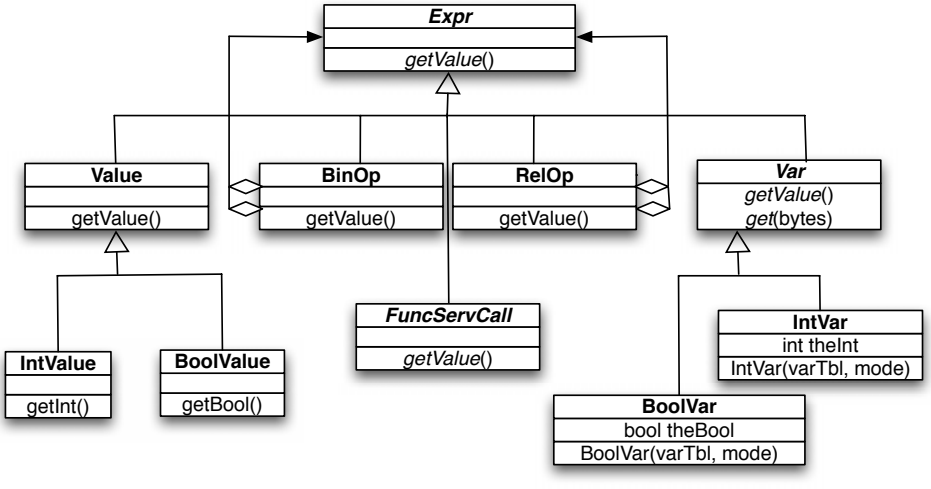

Fig. 6. Expression Hierarchy

the program, `out` and `inout` variables are sent back to the client. Variables on the table can be consulted and modified on both sides.

The adequacy of the table implementation depends on the context of the pattern instance. For example, it can be interesting in an operating system to implement the table as a chunk of raw memory, while a remote server could represent it as a heterogeneous list of concrete variables.

This kind of implementation of BATCHING programs has the advantage that most of type-checking is done at compilation time. Note that server calls are type-checked, because the parameters of constructors of server call commands are typed. In a different context, however, a system designer may opt for a different implementation, for example, based on dynamic typing.

Revisiting our example, the code for the `copy` program is shown in Figure 7. In the figure, constructors are functions that build objects within the program. In this example, `SeqCtrlStructure` and `WhileCtrlStructure` are `Concrete-ControlStructure`s of the language. `Open`, `Close`, `Read`, and `Write` are classes derived from `Program`. 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`.

Finally, note that the concrete instruction set used through this section is just an example. Any other one, like a bytecode-based program could be used, too. Instruction sets suggested in the pattern are very simple compared to the ones used in other systems. For instance, μChoices [LTS+96] and Aglets [PPE97] use a Java interpreter. A Modula-3 compiler is used in SPIN [BSP+95], and NetPebbles [MPZ+98] uses a script interpreter.

When implementing the pattern, the design of the instruction set and its interpreter is one of the most important things the designer should keep in mind. The quality of the implementation depends heavily on the instruction set and interpreter being well-balanced and secure.

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

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

Fig. 7. Program for Copy

6.2 Exception Handling

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 let the user 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 bytecode-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 a 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 a `Program` 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 one way or another. In a language with exceptions, such as C++, Java or Python, it suffices to raise and propagate an exception in the `terminate` code, catching it in the `Program` `run` code. In languages without exceptions, such as 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 `terminate` body.

7 Consequences

The pattern brings the following **benefits**:

1. *It provides a virtual 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 *virtual machines* [Nai05]. 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 `copy` 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 approximately 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* exchanged by 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 for the call to be made.
6. *It enables dynamic extension of servers.* Servers can be extended by accepting `Program`s from clients. Those programs could be kept within the server and used as additional entry points into the server. Should it be the main motivation to use the pattern, the concrete command set should be powerful enough.
7. *It makes the communication more secure* because one can encrypt the complete sequence of commands in a single session, with a single signature. Many attack techniques rely on having a large quantity of messages to work with. Having a single message exchanged between the client and the server prevents variants of man-in-the-middle, replay, or other attacks [GSS03] that would operate on individual commands if multiple messages were exchanged within a client/server communication session.

The pattern brings 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 entire server would be effectively switched off by a single client.

   In such case, either avoid using Batching, or implement `BatchServer` with support for multithreading. That is, arrange for each `Program` to use its own thread. In this case, make sure the instruction set is thread-safe, otherwise programmers will need to rely on locking to protect critical, shared resources.
2. *Server 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.

   On the other hand, note that the pattern does not add functionality to the server. It simply enables the use of existing functionality in a "batch". Any server must always check its inputs (from clients) and these checks must still be performed when the individual calls come from the pattern interpreter.

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 for the overhead introduced by BATCHING.
4. *Clients might become more complex* because they must build the program to be sent, instead of simply issuing the calls to the server when they are needed.

8 Related Patterns

Both `Program` and `ControlStructure` rely on instances of the INTERPRETER pattern [GHJV95]. Indeed, the interpreter of a `Program` is behind its `run` method.

`Program`, `ControlStructure`, and `Command`s make up an instance of the COMPOSITE pattern [GHJV95]. Composite programs, such as `Sequence` and `Conditional`, are aggregates of `Assignment`s, `ServerCall`s, 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, `Program`s should be serialized (and later deserialized) 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 [GHJV95].

A server call issued within a `Program` might fail or trigger an exception. If that is the case, the entire `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 [GHJV95] can encapsulate the program state while in a "frozen" 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 [LS95] and the RENDEZVOUS [JPPMA99] patterns can be used.

COMPOSITEMESSAGES can be used to transfer the `Program` from the client to the server. The COMPOSITEMESSAGES pattern [SC95] applies when different components must exchange messages to perform a given task. It groups several messages in a structured fashion, doing with messages what BATCHING does with server entry-points. In that way, extra latency due to message delivery can be avoided and components are 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.

COMPOSEDCOMMAND [Tid98] is similar to BATCHING in that it bundles several operations into a single one. However, BATCHING is more generic in spirit.

ADAPTIVE OBJECT-MODELS [YJ02] is an architectural style in which the users' object model is interpreted at runtime and can be changed with immediate

effects on the system interpreting it. It is normally seen in advanced commercial systems in which business rules are stored in places such as databases or XML files. Although its implementation might resemble the BATCHING pattern, its major goal is to provide more flexibility and enable runtime reconfiguration of business rules and not to improve the performance.

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 [BJP+00] 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 learned back to "practice." In this section, we show how existing systems match the pattern described in the previous sections—certainly, this will lead to 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 lets a single implementation of the pattern handle the various applications described below. As the activity carried out at the server is specified every time a `Program` runs, the same BATCHING implementation 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 all these applications (e.g., gather/scatter is always implemented separately from message batching facilities, when both are provided.)

Operating System extensions by downloading code into the kernel (as performed in SPIN [BSP+95], μChoices [LTS+96], and Linux [Hen06]) 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 `ConcreteControlStructures` 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 is downloaded into the system.)

Agents. An agent is a piece of autonomous code that can be sent to a different domain. Agents may move from one domain to another, carrying its runtime state [BR05]. The aim is to avoid multiple domain crossings (or network messages), improving performance, and support disconnection from the agent home environment.

Programs built using BATCHING are meant to stay at the server until termination, and they possess no `go`[2] 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. Nevertheless, even within the mobile agent paradigm, a very common situation is to have a single-hop agent that leaves a client, visits a single server and return to the client, as is the case with BATCHING.

Gather/Scatter I/O. 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. Written data is gathered from separate output buffers. Read data is scattered across 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 that, by using this pattern, gather/scatter I/O could be generalized so that the I/O device involved would not necessarily be the same for all descriptors sent by the user. Moreover, multiple `Read` and `Write` operations could be bundled into a single one.

Message batching. Grouping a sequence of messages into a single low-level protocol data unit is yet another instance of the pattern. In this case, the `run` method (i.e., the interpreter) is the packet *disassembler*. A program is a bunch of packets bundled together. Each packet, or each packet header, is a command that is interpreted by the packet *disassembler*. This is the BATCHING application that more closely resembles COMPOSEDCOMMAND [Tid98].

Deferred calls. BATCHING can be used to support disconnected operation [MRM06]. Clients build programs while they perform operations on non-reachable servers whenever they are in a disconnected state. Upon reconnection, each program is finally submitted to the target domain for interpretation. Note that *several* clients might add code to a *single* program to be sent to the server later on.

Each operation is a `Command`, the list of operations sent to a server is a `Program`. The interpreter could be either:

1. the piece of code sending each command when the client is reconnected to the server, or
2. an actual `Program` interpreter in the server domain, accepting just a list of commands (a program)—to save network traffic.

Futures or Promises [LS88] can be used by client code to synchronize with server responses.

Improving latency in Operating Systems. Many user programs happen to exhibit very simple system call patterns. This is an opportunity for using

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

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++* [BHK+99]. In both systems, we obtained around 25% speedups for a `copy` program written with Batching [BJP+00].

We implemented two new domain-specific languages (i.e., `ControlStructures` and `Command` sets) that let users bundle separate calls into a single one, like in the `copy` example of Section 1. The first language we implemented was based on bytecodes. We included just those commands needed to code loops, conditional branches, and basic 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 [BJP+00].

Heterogeneous resource allocation. Most operating systems are structured as a set of resource unit providers. Separate servers provide resource unit allocation for different types of resources. In these systems, users issue multiple requests at a time.

Batching can be used to request allocation of multiple heterogeneous resources in a single system call. *Off++* is an operating system modeled as a set of hardware resource unit providers and it uses Batching in this way to improve the performance of its applications [BJP+00].

Transaction processing. A transaction is a set of operations executed atomically in isolation [Gra78]. A given transaction can either terminate normally, by committing, or abnormally, by aborting. Should a transaction abort, its effects must be undone; otherwise (i.e., when it commits), its results should be made permanent.

Commits typically involve multiple disk writes for different data items. Writes must follow a carefully chosen order to preserve the consistency of results, even when failures occur. One of the strategies uses a `redo` algorithm [BHG87]. Such algorithm does not modify the persistent data until the transaction is completed, it works on a volatile copy of the data until the commit is performed. At commit time, a sequence of *redo records* is written into a disk log, followed by a commit record. Redo records contain new values for objects changed by the transaction. Finally, persistent state for objects involved is updated. If the system fails before the write of the commit record, the transaction is aborted and their redo records are ignored. If the system fails after writing the commit record, redo records are replayed.

Another instance of Batching is *group commit* [DKO+84], used to avoid the latency of forced disk writes of log records in each commit. In group commit, instead of forcing the log with each commit, a set of consecutive commits are batched. Then, a single forced disk write is performed to write all the log records associated with all the commits, amortizing the latency of forced disk writes across several commits. This results in a substantial improvement in throughput for commits.

BATCHING can be used both to implement commit and for crash recovery. Performance of transactional distributed object systems (e.g., Arjuna [SDP91]) could be improved due to the reduced number of domain crossings.

10 Variant

A widely-used variant of this pattern is the CLIENT-SIDE BATCHING pattern. In this case, instead of the client sending batched code to be executed in the server, it is the server that sends code to be executed in the client.

The CLIENT-SIDE BATCHING pattern is frequently used on the Web where the inter-domain communication latency is normally very large. Known uses include Java applets [Boe02], Javascript functions embedded in Web pages [Fla02], and Macromedia Flash applications [Abe02].

In both BATCHING and CLIENT-SIDE BATCHING, the goal is to improve the response time of the system as perceived by the client and, in both cases, this is achieved by avoiding multiple cross-domain calls. The difference is where the batched program is executed, in the client or in the server.

11 Conclusions

BATCHING unifies several, apparently unrelated, techniques. Most notably, the pattern integrates techniques to (1) reduce domain crossings and (2) avoid unnecessary data copying. Encapsulation of the command language has been a key feature in the integration of existing techniques, decoupling the command set from the submission method.

We showed eight different applications where the BATCHING pattern was previously used and cases where the pattern was applied with *a-priori* knowledge of it. A client-side variant of the pattern is also implemented by different technologies and is widely-used on current Web systems.

Acknowledgments

We are sincerely grateful for the help provided by our shepherd, Frank Buschmann, and for the valuable feedback provided by John Vlissides, who suggested the new name for this pattern (it was previously called COMPOSITECALL). Finally, we are also grateful to the anonymous TPLoP reviewers and to the members of the "Allerton Patterns Project" group of PLoP'99 for their comments and suggestions.

References

[Abe02] Aberdeen Group: Flash Remoting MX: A Responsive Client-Server Architecture for the Web. Technical report, Macromedia White paper (December 2002)

[BHG87] Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison-Wesley, Reading (1987)

[BHK+99] Ballesteros, F.J., Hess, C., Kon, F., Arévalo, S., Campbell, R.H.: Object Orientation in Off++ - A Distributed Adaptable μKernel. In: Proceedings of the ECOOP 1999 Workshop on Object Orientation and Operating Systems, pp. 49–53 (1999)

[BJP+00] Ballesteros, F.J., Jimenez, R., Patino, M., Kon, F., Arévalo, S., Campbell, R.H.: Using Interpreted CompositeCalls to Improve Operating System Services. Software: Practice and Experience 30(6), 589–615 (2000)

[Boe02] Boese, E.S.: Java Applets: Interactive Programming, 2nd edn. Lulu.com (2002)

[BR05] Braun, P., Rossak, W.: Mobile Agents: Basic Concepts, Mobility Models, and the Tracy Toolkit. Elsevier, Amsterdam (2005)

[BSP+95] Bershad, B.N., Savage, S., Pardyak, P., Sirer, E.G., Fiuczynski, M., Becker, D., Eggers, S., Chambers, C.: Extensibility, safety and performance in the SPIN operating system. In: Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles, December 1995, ACM, New York (1995)

[DKO+84] DeWitt, D.J., Katz, R.H., Olken, F., Shapiro, L.D., Stonebraker, M.R., Wood, D.: Implementation techniques for main memory database systems. In: Proceedings of the ACM International Conference on Management of Data (SIGMOD), pp. 1–8 (1984)

[Fla02] Flanagan, D.: JavaScript: the definitive guide. O'Reilly, Sebastopol (2002)

[GHJV95] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Elements of Object-Oriented Software. Addison-Wesley, Reading (1995)

[Gra78] Gray, J.: Operating Systems: An Advanced Course. Springer, Heidelberg (1978)

[GSS03] Garfinkel, S., Spafford, G., Schwartz, A.: Practical UNIX and Internet Security. O'Reilly, Sebastopol (2003)

[Hen06] Henderson, B.: Linux Loadable Kernel Module HOWTO. Technical report, Linux Documentation Project (September 2006)

[JPPMA99] Jiménez-Peris, R., Patiño-Martínez, M., Arévalo, S.: Multithreaded Rendezvous: A Design Pattern for Distributed Rendezvous. In: Proc. of ACM Symposium on Applied Computing, February 1999, ACM Press, New York (1999)

[LS88] Liskov, B., Shrira, L.: Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems. In: Proc. of ACM Conf. on Programming Language Design and Implementation, pp. 260–267 (1988)

[LS95] Lavender, R.G., Schmidt, D.C.: Active object – an object behavioral pattern for concurrent programming. In: Proceedings of the Second Pattern Languages of Programs conference (PLoP), Monticello, Illinois (September 1995)

[LTS+96] Li, Y., Tan, S.M., Sefika, M., Campbell, R.H., Liao, W.S.: Dynamic Customization in the μChoices Operating System. In: Proceedings of Reflection 1996, San Francisco (April 1996)

[MPZ+98] Mohindra, A., Purakayastha, A., Zukowski, D., Devarakonda, M.: 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)

[MRM06] Mikic-Rakic, M., Medvidovic, N.: A Classification of Disconnected Operation Techniques. In: Proceeding of 32nd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO 2006), pp. 144–151. IEEE Computer Society, Los Alamitos (2006)

[Nai05] Nair, R.: Virtual Machines: Versatile Platforms for Systems and Processes. Morgan Kaufmann, San Francisco (2005)

[PPE97] Clements, P.E., Papaioannou, T., Edwards, J.: Aglets: Enabling the Virtual Enterprise. In: Proc. of the Managing Enterprises - Stakeholders, Engineering, Logistics and Achievement Intl. Conference (ME-SELA 1997), Loughborough University, UK (1997)

[SC95] Sane, A., Campbell, R.: Composite Messages: A Structural Pattern for Communication between Components. In: OOPSLA 1995 workshop on design patterns for concurrent, parallel, and distributed object-oriented systems (1995)

[SDP91] Shrivastava, S.K., Dixon, G.N., Parrington, G.D.: An Overview of Arjuna: A Programming System for Reliable Distributed Computing. IEEE Software 8(1), 63–73 (1991)

[Tid98] Tidwell, J.: Interaction Design Patterns. In: Proceedings of the Conference on Pattern Languages of Programs (PLoP 1998), Monticello, Illinois (1998)

[YJ02] Yoder, J.W., Johnson, R.: The Adaptive Object Model Architectural Style. In: Proceeding of The Working IEEE/IFIP Conference on Software Architecture 2002 (WICSA3 2002). Kluwer Academic Publishers, Dordrecht (2002)