`CompositeCalls`: 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 `CompositeCalls` design pattern.

All techniques mentioned above are designed for multiple-domain[1] applications. In all of them, 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 describe the `CompositeCalls` pattern and identify eight classes of existing techniques that instantiate it. We, then, discuss the circumstances in which the pattern should and should not be used. Finally, we present some work done with *a priori* knowledge of the pattern.

*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 a grant from CAPES, the Brazilian Research Agency, proc.# 1405/95-2.

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

[1]By "domain" we mean either a process, a node in a network, or a protection domain. Communication between objects residing at different domains is referred to as a "domain-crossing" or a "cross-domain" call.

Copyright © 1999, Marta Patiño, Francisco J. Ballesteros, Ricardo Jiménez, Sergio Arévalo, Fabio Kon and Roy Campbell. Permission is granted to copy for the PLoP 1999 conference. All other rights reserved.

```
cat() {
while (FileServer::aFile.read(buf))
  write(FileServer::otherFile.write(buf));
}
```

Figure 1: Cat Code

1 Introduction

Applications such as code downloading, message batching, gather/scatter and mobile agents follow the client-server model of interaction. But that is not the only feature they share. A closer look reveals that all of them bundle a set of operations, and submit them to a server for its execution. The submission of operations is aimed to reduce domain-crossings and/or allow dynamic server extension. For instance, code downloading is 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.

Suppose `FileServer::File::read` and `FileServer::File::write` are handled by the same server. Besides, suppose 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.

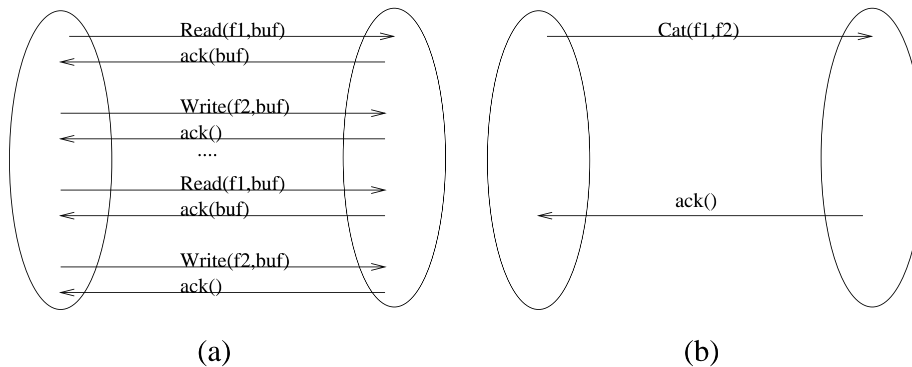

Figure 2: Interactions corresponding to read/write services and a cat service

In what follows, the problem and its solution are described in sections 2 and 3. The `CompositeCalls` design pattern is described in section 4. Section 6.1 shows how clients can build programs for `CompositeCalls`. Execution of programs is explained

in section 5 as well as how to deal with exception handling. Applicability of the pattern is discussed in section 8. Section 9 presents some well-known design patterns related to `CompositeCalls`. A more detailed description of `CompositeCalls` applications can be found in section 10. Finally, some conclusions are presented in section 11.

2 The problem

Both cross-domain data traffic and cross-domain call latency have a significant impact in the efficiency of a multiple-domain application.

Note that cross-domain calls and data cross-domain data transfers happen also 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). Besides, centralized environments using multiple processes have a domain boundary around every process considered. Of course, in a distributed system, the network behaves also 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. an 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 some piece of data to another object at 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.)

On many application domains, like file systems and databases, data movement can be the actual performance bottleneck for the application considered. Therefore, avoiding unnecessary data transfer operations can be crucial.

Moreover, under many circumstances, extra data transfers occur just because the object controlling the operation being performed resides far from the data source and/or the data sink. That is precisely what happened 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.

Another issue is call latency. A call between two objects residing at different domains is much more expensive 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 the operating system kernel (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 which could use fewer domain crossings to perform the set of calls the application makes, would be much more efficient.

When designing a solution, it should be taken into account that, under certain circumstances (e.g. when *cheap domain crossing* is available and efficiency is your primary objective), the overhead introduced to solve the problem might actually degrade performance. However, even when cheap domain crossing is available, overhead caused

by cross-domain data transfers (e.g. messages sent over a network) might still be a problem.

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

3 The solution

By composing separate method calls into a single cross-domain call, unnecessary data copying can be avoided and the number of cross-domain calls can be reduced.

Clients can build a "composite call" and send it once to the server. The composite call contains the interaction with the server (i.e. it knows what has to be done in the server). Such composite call might perform *multiple* operations on that server even though the client only had to send it *once*.

In our example (see figure 2, the interactions for cat), if the `CompositeCalls` pattern were not used, file contents would travel twice across the network. However, if a `cat` composite call is submitted to the server, the file will not leave the server, that is, it will be copied locally. Moreover, a single cross-domain call suffices (the one sending the `cat` program to the server).

4 Pattern structure

4.1 Participants

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

`InterpServer` behaves as a facade [5] to services provided by the server. An object of this class is located at the server side. It supplies interpretation facilities to service callers, so that clients could now 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 starts program execution and returns any results to the client.

`ConcreteServer` represents a server being used. This class is only present at the server side. It provides the set of entry points which can be called by the client.

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

`CompositeCall` is an abstract class that represents the program to be interpreted. It is built by the client and then sent to the `InterpServer` for execution. It is also responsible for maintaining an associated table of (program) variables. The `run` method of a `CompositeCall` evaluates it.

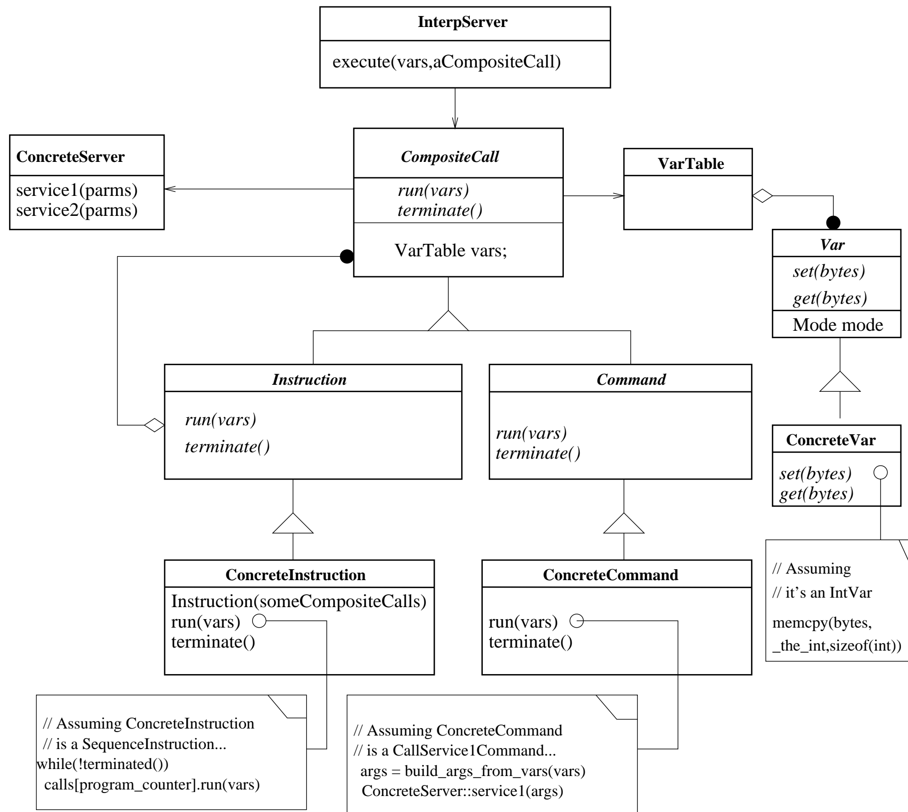

Figure 3: `CompositeCalls`

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

The name of the `CompositeCall` comes from the fact that it borrows from the Composite pattern [5] its basic structure.

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

`ConcreteInstruction` represents 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. `ConcreteInstruction` constructors can be used at the client side to build complex `CompositeCalls`.

`Command` is a leaf `CompositeCall` which represents a single operation to be performed. (It resembles the command pattern shown in [5], hence the name).

ConcreteCommand is a concrete operation to be performed. Example ConcreteCommands can be arithmetic operations, logic operations and calls to ConcreteServer entry points. The only purpose of the CompositeCalls is to bundle several ConcreteCommands together.

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

There is a variable table per CompositeCall (pairs of VarTable and CompositeCall are sent together to the InterpServer). Thus, all components of a concrete CompositeCall share a single variable table so that they could share some variables.

Var is an abstract class representing a variable of the program sent to the server. It has some storage associated (bytes, in the figure). Var instances are kept within a VarTable. Variables have a *mode*, which can be either input (parameter given to the CompositeCall), output (result to be given to the user), inputoutput (both), or none (local variable). By including the mode qualifier, this class can be used for local variables as well as it can be used 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 CompositeCall. 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

In terms of our file server example, the concrete structure of classes is as shown in figure 4. Intuitively, the CompositeCalls instance for the file server adds an interpreter (see the Interpreter pattern in [5]). That interpreter can execute programs which (1) call to 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.

As it can be seen, we have simplified a bit the typical interface provided by a file server. A typical file server would contain several File objects which would supply read and write methods. To illustrate the pattern in a more simple way, we have omitted the actual file being used[2].

Starting with the pattern instantiation, an InterpFileServer has to be implemented. It will be collocated with the FileServer, providing a new execute service which

[2] Nevertheless, obtaining a complete implementation is a matter of adding a File class and adding File parameters to those methods accepting a buffer to be read or written.

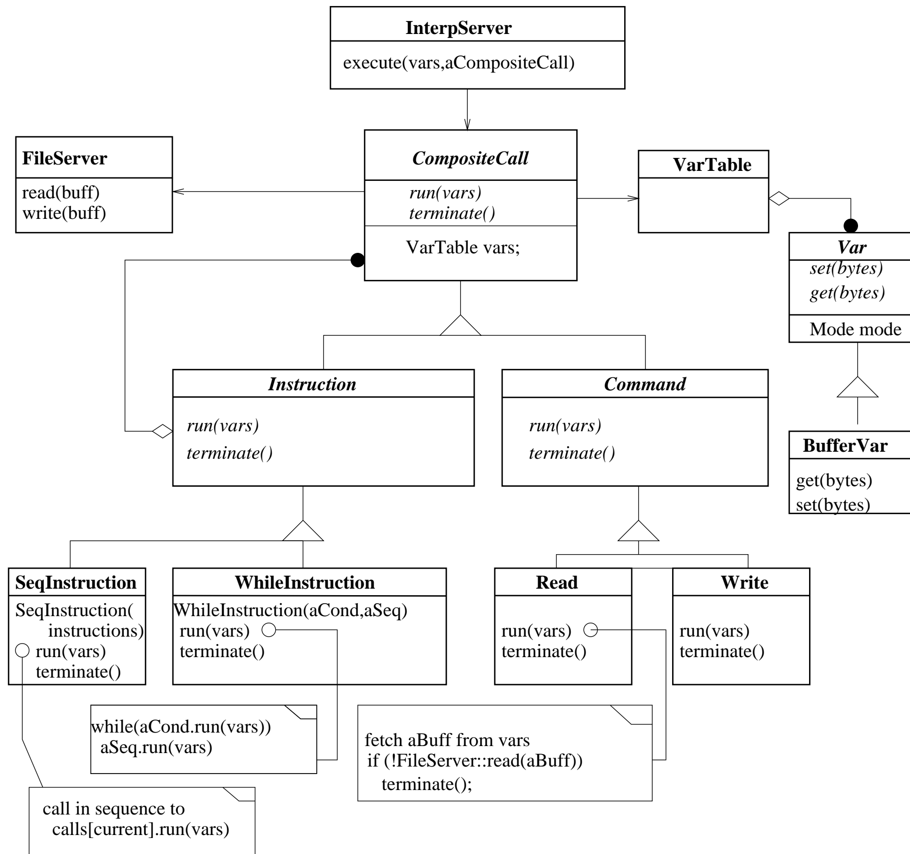

Figure 4: File server `CompositeCalls`

supplies an interpretative version of `FileServer` services. The `InterpFileServer` corresponds to the `InterpServer` in the pattern (see the pattern diagram in figure 3).

The `InterpFileServer` will accept a `CompositeCall`, which is a program built in terms of `Instructions` and `Commands`.

To execute

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

the `CompositeCall` sent to the `InterpFileServer` should be made of a `WhileInstruction`, using a `Read` as the condition. The body for the `WhileInstruction` should be a sequence made of a single `Write` command.

Here, `WhileInstruction` and `SeqInstruction` correspond to `ConcreteInstructions` in the pattern. Besides, `Read` and `Write` match `ConcreteCommands` in the pattern.

The buffer to be read and written is handled by a `Buffer` class instance, which corresponds to a `ConcreteVar` in the pattern.

Once the pattern has been instantiated, a client is able to build a composite call (for the file server) using constructors provided by `WhileInstruction`, `SeqInstruction`, `Read`, and `Write`. Such `CompositeCall` can be later submitted, by the client, to the `InterpServer::execute` method.

5 Dynamics

When a program is received at the server side, it will be deserialized.

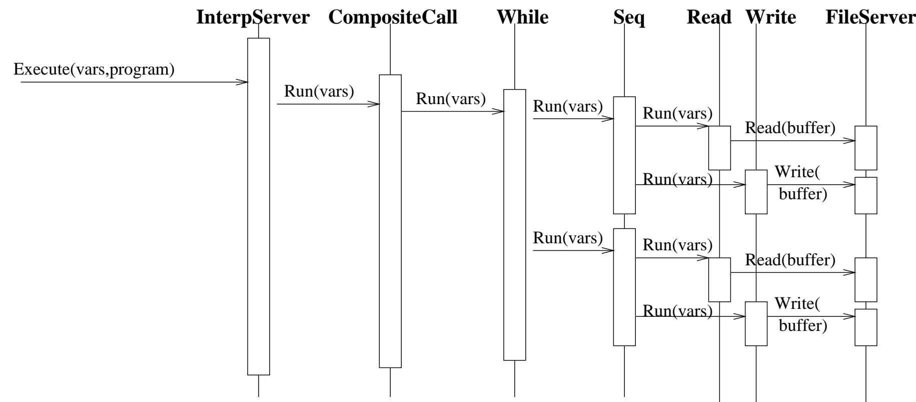

Figure 5: Interaction diagram for a cat composite call.

An `InterpServer` 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 will call the `Run` method of the program providing the table of variables; this method will interpret the program. Once `Execute` finishes results are returned.

The `Run` method of the `CompositeCall` class implements recursive interpretation. When the program has been interpreted, that is, the `Run` method has finished, results of the program execution are still in the variable table. As part of the `Execute` method, the table will be serialized and sent back to the client.

In our example, we can see[3] the interaction diagram for a cat composite call in figure 5.

The `Run` method of the `CompositeCall` will call the `Run` method of the `Instruction` representing the program (the `While` in the interaction diagram). `Instructions` provide a `Run` method to interpret themselves. That is, a program has built-in its own interpreter; it is an instance of the `Interpreter` pattern [5]. So, the `While command` calls the `Run` method of its `commands` (`Seq` in the interaction diagram for cat).

It must be noticed that instruction sets suggested in the pattern are very simple compared to the ones used in other systems. For instance, μChoices [9] and Aglets

[3] Calls to `Open` and `Close` has been supressed for the sake of simplicity.

[12] use a Java interpreter. A Modula-3 compiler is used in SPIN [4], and NetPebbles [11] uses a script interpreter.

6 Implementation issues

Two important aspects are how to build composite call programs and what to do if they fail.

6.1 Composing programs

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

`Instruction` constructors are functions. Thus, code in the client for a `CompositeCall` 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 allow usage of expressions within the `CompositeCall`, functional services inheriting from an `Expr` class could be provided (see figure 6). `Expr` represents an expression, and can be used as a function within expressions. Note that nothing prevents from encapsulating a functional service within a procedural call and vice-versa.

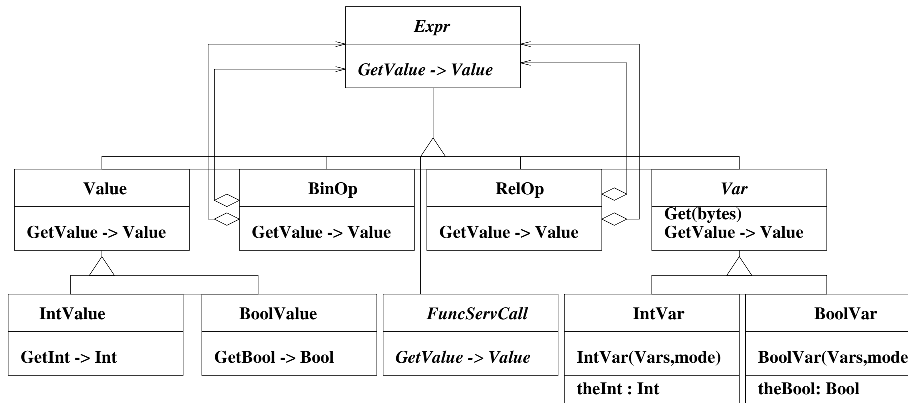

Figure 6: Expression Hierarchy

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 class `VarTable`. When variable objects are constructed, they are constructed and stored in that table with an initial value, if any, and their mode `In, Out, InOut` or `None`. When a variable table is sent to the server only values of `In` and `InOut` variables have to be copied to the server. The `None` type means that a variable is a local one, that is, its initial and final values

```
vars: VarTable;
program : CompositeCall;
f1, f2 : IntegerVar(vars, None);
car : CharVar(vars, None);

   Program:= Seq((
      Open(f1, StringLit("name1")),
      Open(f2, StringLit("name2")),
      While(Read(f1, car),
         Write(f2, car)),
      Close(f1),
      Close(f2)
   ));
   Execute(program, vars);
```

Figure 7: Program for Cat

are not needed by neither the server nor the client respectively. After the execution of 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 implementation of the table will depend 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, whilst a remote server could represent it as a heterogeneous list of concrete variables.

`CompositeCalls` programs have the advantage that most of type-checking is done at compilation time. Note, that server calls are type-checked, as the parameters of constructors of server call commands are typed.

Revisiting our example, the code for the `cat` program is shown in figure 7. In the figure, constructors are functions which build objects within the composite call.

In this example `Seq` and `While` are concrete `Instructions` of the language. `Open`, `Close`, `Read`, and `Write` are classes derived from `Command` and clients invoke their constructors to let the `CompositeCall` 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 `None`.

Finally, note that the concrete instruction set used through this section is just an example. Any other one, like a byte-code based program could be used too.

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/she can decide when a call has failed to terminate the program. This can be done by calling the `Terminate` method of the `CompositeCall` 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 (due to iterative interpretation) 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: Run calls of CompositeCalls will propagate calls to the Run method of its components. To stop that program, it will be 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 will be enough to raise and propagate an exception in the code of Terminate, catching it in the Run code of CompositeCall. In other languages setjmp can be used in the top-level Run method before calling any other Run, and longjmp in the body of the Terminate, for the same purpose.

7 Variants

Servers can be extended by accepting CompositeCalls from clients. Those programs could be kept within the server, and be 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.

In fact, system extension by code downloading (like in SPIN [4] or μChoices [9]) can be considered to be an instance of this pattern. Such 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, ConcreteServer is the system being extended; the CompositeCall is the extension performed; the set of Instructions 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).

8 Consequences

The pattern has the following benefits:

1. *Provides an Abstract machine view of the server.* When using CompositeCalls, 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 those calls, which can be made, together with some general-purpose control language.

   Therefore, it is feasible for users to reuse instructions (and programs) for different composite calls. Common programs to interact with the server can be built, and reused later.

2. *Reduces protection-domain crossings*, as the `cat` program did above. Should it be 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 can be approximated as the difference between the time to perform a cross-domain call and the time to interpret and dispatch a server call.

3. *Reduces the number of messages* between clients and servers; provided that the client 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 the program construction and interpretation. Therefore, careful timing must be done prior to pattern adoption.

4. *Decouples client/server interaction from the call mechanism.* `CompositeCalls` 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. *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.

6. *Allows dynamic extension of servers*. Servers can be extended by accepting `programs` from clients. Those programs could be kept within the server, and be 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.

The pattern has the following drawbacks:

1. *Client requests might take a non-fixed time* to complete. A composite call might lead to a never ending program, which could be downloaded into the server for execution. If server correctness depends on shortly terminated 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 `CompositeCalls`, or handle multithreading issues like a side-effect (i.e. arrange for each `CompositeCalls` to use its own thread, using a giant lock[4] to protect non MT-safe servers[5]).

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

[5] Those servers not prepared to handle concurrent requests.

2. *High security servers* 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 `CompositeCalls`, or reduce the complexity of your command set to the bare minimum.

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

9 Related patterns and Collaborations

Both `CompositeCall` and `Instruction` are instances of the `Interpreter` pattern [5]. Indeed, the interpreter of a `CompositeCall` is behind its `Run` method.

Of course, `CompositeCall`, `Instruction`, and `Command` are an instance of the `Composite` pattern [5]. Composite commands, such like `Sequence`, `Conditional`, etc., are aggregates of `Assignments`, `ServerCalls`, and other primitive commands.

If an instruction set for a `CompositeCall` language is to be compiled, `CompositeCall` might include a method to compile itself into a low-level instruction set. Besides, `CompositeCalls` 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 `CompositeCall` might fail or trigger an exception. Be that the case, the whole `CompositeCalls` can be aborted and program state transmitted back to the client—so that the client could fix the cause for the error, and resume `CompositeCalls` execution. The `Memento` pattern [5] can be used to 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 into the server.

As a program can lead to an endless client request, single threaded or a-request-at-a-time servers can get in trouble. To accommodate this kind of servers so that `CompositeCalls` could be used, the `ActiveObject` [8] and the `RendezVous` [7] patterns can be used.

A `Composite Message` can be used to transfer the `Composite Call` from the client to the server. The `Composite Messages` design pattern [14] applies when different components must exchange messages to perform a given task. `Composite Messages` allows to bundle several messages together in an structured fashion (it does with messages what `CompositeCall` does with server entry-points). In that way, extra latency due to message delivery can be avoided, and components decoupled from the transmission medium. The main difference is that `CompositeCall` is targeted to the invocation of concrete server-provided services, and not to package data structures to be exchanged.

Last, but not least, `ComposedComands` [16] are similar to `CompositeCalls` in that they bundle several operations in a single one. However, `CompositeCalls` are more

generic in spirit.

10 Applications

Our experience with `CompositeCalls` 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 try to show how different existing systems match the pattern described in the previous section—hopefully, this will allow a better understanding of the pattern, as it 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 `CompositeCalls` design allows a single instance to handle some of the "applications" below. As the server being handled is specified on a per-`CompositeCall` run basis, the same piece of code could perfectly handle most of the applications shown below. On the other hand, 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).

System extensions by code downloading (like in SPIN [4] or μChoices [9]) can be considered to be an instance of this pattern. Such 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, `Services` are those of the existing system; the `Program` is the extension performed; the set of `Commands` 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 move 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 `CompositeCalls` are thought to stay at the server until termination, and there is no `go` statement. However, `CompositeCalls` already include most of the machinery needed to implement an agent system. On the other hand, a `go` statement could be provided by the command language employed.

Gather/Scatter IO. Gather/Scatter input/output is yet another example where this pattern appears. In gather/scatter IO a list of input or output descriptors is sent to an IO device in a single operation. Each descriptor describes 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 aim is mainly 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 those IO operations requested. Services provided (i.e. call commands needed) are just `read` and `write`.

Note how by considering this pattern, gather/scatter IO could be generalized so that the IO 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

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 which will be interpreted by the packet *disassembler*. This is the `CompositeCalls` application which more closely resembles `ComposedComands` [16].

Deferred calls. They can be used to support disconnected operation. Clients build programs while they perform operations on non-reachable servers—whenever we are in a disconnected state. On 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 `CompositeCall`. The interpreter could be either:

1. the piece of code sending each command in turn when it is reconnected, or
2. an actual `CompositeCalls` interpreter in the server domain, accepting just a list of commands (a composite call)—to save some network traffic.

Futures or Promises [10] can be used to allow users to synchronize with server responses.

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

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

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

The first language we implemented was byte-code based. 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 implemented was a high-level one, designed specifically for *Off* ++. It includes just those commands needed to `repeat` a given operation `n` times, and to perform a `sequence` of operations.

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

The `CompositeCalls` can be used to request allocation of multiple heterogenous resources in a single call. It can be done even using an empty `ControlCommand` family.

Being *Off* ++ a system modeled as a set of hardware resource unit providers, `CompositeCalls` have been used to improve *Off* ++ applications [1].

Transaction processing. A transaction is a set of operations executed atomically on isolation [6]. 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".

Commitment typically involves multiple disk writes for different data items. Writes must follow a carefully chosen order in order to preserve persistence of results, even when failures occur. One of the strategies is to use a `redo` algorithm [3]. Such algorithm does not modify the persistent data until commitment: it works on a volatile copy of the data until the transaction commits.

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 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.

The `CompositeCalls` can be used both to implement commit, and crash recovery. Performance of transactional distributed object systems (e.g. Arjuna [15]) could be improved due to the reduced number of domain crossings.

We will use the transaction log as our legacy server, and then apply `CompositeCalls` to it. Operations performed on the log are: `readRedo` reads a redo record from the log, `writeRedo` appends a redo record to the log, `writeCommit` appends a commit record and `writeObj` updates state for the given object in persistent storage.

All of them will be wrapped in `CallCommands`. For control commands, a while loop (`while`) and a sequence (`seq`) will suffice.

11 Conclusions

A new design pattern, `CompositeCalls`, has been presented. `CompositeCalls` unifies many—apparently unrelated—techniques. Most notably, the pattern integrates

techniques both to reduce domain crossings and to dynamically extend existing servers and to avoid copying data more than needed.

One of the advantages of the pattern is that it allows instances simpler and more efficient than known instances like [4].

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 have shown also eight different applications where the `CompositeCalls` has been previously used, and several cases where the pattern has been applied with *a-priori* knowledge of it.

12 Acknowledgments

We are sincerely grateful to our shepherd, Frank Buschmann. He helped us so much, and contributed to the final version of the paper with so many useful comments, that he should actually be considered a co-author of this paper.

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 (ME-SELA '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.