Features of complementary objects usage in distributed virtual environment

Ilia O. Shalnev
St. Petersburg Federal Research Center of the Russian Academy of Sciences, St.-Petersburg
shalnev.i@iias.spb.su

Abstract. In this paper, complementary objects design and usage is considered. The presented examples illustrate the programming paradigms necessary for correct usage of complementary objects that are essential in distributed computation systems.

1. Introduction.
The increase of processed information and the need for permanent access to the information system exacerbate the problem of components interaction in the network system. Actually, everybody have a constant access to a space of personal and corporative information interaction. They participate in the information flow generation and management of signals, interacting with other users of a system. It should be noted that “other users of interaction” should be understood not only as people, but also different information systems, data bases, artificial intelligent systems, etc.

Distributed system is a standard in network software development. In contrast to the distributed systems, the centralized systems are designed as a set of autonomous processes interconnected in a network. Dynamic reconfiguration of devices connected by a network is an essential challenge in a distributed system development. To solve the problem of device dynamic reconfiguration in distributed systems, a concept of a distributed virtual machine was invented [1]. This concept is a modern realization of distributed object technology [2]. Distributed object computation is based on object-oriented programming (OOP) [3] and allows existence of an object in a heterogeneous network environment. Since a canonical object in OOP is not a network entity, it was suggested to expand the paradigm with a complementary object definition. Objects divided into execution and application program interface (API) components are called complementary. An execution object is presented as an executable code which has all implemented logic presented in a machine code. Such an execution object is called an actor object and can be controlled by an API object in distributed virtual environment. Control is provided by sending a set of commands called a bytecode. An actor object can also generate the bytecode in two cases: when emitting an event and when returning a result of function. These parts of complementary object can be placed on different nodes of network, thus providing remote control of an actor object by means of an API object (Fig. 1).

An API object has the same name as an actor object with adding the prefixV(for “virtual”). Although the use of usual OOP objects and API objects is similar, they have some differences that should be considered: processing of the object’s method output, and creation and destruction of the complementary objects.
2. **Processing of object’s method output.**

Distributed systems such as CORBA and Java RMI (Remote Method Invocation) are based on synchronous interaction. Synchronous invocations are indeed quite adequate to local network environments. These environments assume such characteristics as network reliability, zero latency, steady bandwidth. However, exclusively synchronous communication is not adequate to the modern network environments, such as Internet. Such networks usually have disconnection problems, latency and non-steady bandwidth [4]. One of the most serious problems of the synchronous approach used in non-reliable networks is program error handling. Thus, in case of failure of remote invoking procedure, the remote procedure will be “orphaned”. On the other hand, as a result of remote procedure failure, the invoking procedure will never obtain a result. This situation may occur when the network nodes disconnect. In this case, handling of disconnection is impossible, because the process is busy with waiting the result from a remote side.

Due to the problems mentioned above, the process of object’s method output should be asynchronous. While a usual object computes a result on the same device that handles it, in case of a complementary object computation and handling of the result can be executed on different devices. For this reason, waiting for method’s result evaluation at the moment of invocation (synchronous), as it is with classic OOP objects, is ineffective because of possible processor idling, which is another argument for asynchronous result handling. The API methods of complementary objects do not return their results by means of `C++` language tools and are marked with “void” keyword. The received result should be handled asynchronously through event-driven architecture [5]. At the moment when the result arrives at the API side, the virtual machine defines the object whose method was invoked, and emits a signal of ready result. The signal is handled by callback function. An example of object’s method output processing is demonstrated in Fig. 2b. For comparison, Fig. 2a shows a classical object’s method result handling. The demonstrated object whose method remotely sums up two numbers was created for illustrative purposes. The method’s third argument accepts the result handler, which in this case was defined as a lambda-function that outputs the result on display.

```
Adder adder;
int res = adder.add(10,23);
res = adder.add(res, 7);
qInfo() << “result:” << res;
```

```
Adder adder;
auto adder = new VAdder( getProcessor() );
adder->add(10, 23, [adder](int res){
    adder->add(res, 7, [])(int res){
        qInfo() << “result:” << res;
    }
} );
```

Fig. 2. Synchronous and asynchronous ways of methods result handling

It can be noticed that sequential method invoking is implemented in synchronous and asynchronous ways in quite different manners. If we want to invoke a method with a result of another method as an
3. Creation and destruction of complementary objects.
An API object creation and destruction lead to creation and destruction of an actor object. Information about the necessity for the actor object creation/destruction is sent from the API side in the form of bytecode [6]. Since the complementary objects are of asynchronous nature, creation of an object on stack can lead to unexpected behavior. Because of an object destruction by scope ending, in case of remote method invocation an output cannot be sent and, as a result, be handled. Therefore, it has to be created dynamically with asynchronous destruction after the appearance of any signal, for example, after the result has been returned from the actor side (Fig. 3).

```
auto adder = VAdder::create{ getProcessor() };
auto multiplier = VMultiplier::create{ adder, getProcessor() };
multiplier->multiply(3, 4, [multiplier](int res)
{ 
    qInfo() << "Multiply:" << res;
    delete multiplier
});
```

Fig. 3. Nested virtual object creation.

Nevertheless, the objects’ methods can be of exclusively managing nature. Usually, these methods have nothing to return, or do return an error code. Such objects can be created on stack. On the actor side, it will be seen as object creation, method invocation, and object destruction. Also, these objects can be defined on stack if it is the field of another class, because an object aggregating a virtual object manages its life cycle.

If the developer of a virtual object is sure that the created class is not managing and its methods return the results, they can secure the user from possible errors by hiding the object’s methods in a private or protected zone, and define the static methods creating and copying the virtual objects. In this case, the methods return pointers to created or copied objects.

There is a subset of classes based on other objects. Let us create another demonstration class called Multiplier and VMultiplier, accordingly. The only task of the object is to remotely multiply two numbers. This action is done by iterative adding a number to itself by means of already created Adder class. The pointer to the created Adder object is sent to the Multiplier’s constructor, and after the object destruction the Adder object will also be removed.

VMultiplier class realization should be disregarded from functionality as much as possible, and should duplicate only the signatures of constructors and methods, not their logic. Therefore, as demonstrated in Fig. 4, the procedure of the adder and multiplier objects destruction will be as follows: at first, the API multiplier object will be removed, with a bytecode being formed to remove the corresponding actor object. Then, while destructing the multiplier actor object, the aggregated adder actor object will be removed. Having detected this, the virtual environment of the actor object will generate a bytecode to remove the API object of VAdder class.

The implementation of remote multiplication demonstrated above is not the only possible realization. In this case, the logic of an Adder actor object usage is implemented on the functionality side of the Multiplier class. The VMultiplier class is a program interface for remote access that does not contain any algorithm components. We can create an alternative realization of Multiplier class by shifting the multiplication algorithm to API side. It will be a usual object whose method will request a sum sequentially until the multiplication result is received.

The implemented method (Fig. 5) invokes the remote summing method, and the number of calls will be defined by the method’s second argument, while the VMultiplier class invokes the remote method only once.
In this case, it may seem cost-ineffective, but there is a number of tasks which cannot be executed on a remote device because of weak hardware. Non-remote execution of some tiny part of the task can be a good compromise. For example, shifting some algorithmic part to another device will reconfigure the system for optimization purposes.

4. Conclusion.
This paper demonstrates a realization of complementary objects usage, based on contemporary asynchronous network methodology. Such a technique makes it possible to create network applications capable of fast reconfiguration in an actively changing work environment.

The proposed approach has the following advantages:
- the technique uses object-oriented approach to implement interaction between objects in the distributed system. It can be used in common tasks of object interaction of different types.
Using this method of object interaction, it is possible to perform dynamic control of the logic of different objects behavior from a remote device;

- the method provides a possibility for dynamic control of the functionality side objects from the API side, hence an opportunity to reconfigure software using pre-installed functionality. If a developer needs to change an algorithm, they only need to change a program on the API side, without burdening the client’s side with updating. The client just executes the new bytecode corresponding to changed logic on the server side.

References

[1] Shalnev I O 2020 Building distributed virtual machine based on object-oriented programming Proc. TSU 9 40-7
[2] Ta M, Thomborson C D, Nicolescu R 1999 Test bed for Distributed Object Technologies using Java 490.450 DT Project Report (The University of Auckland Information Technology Faculty) p 261
[3] Booch G, Maksimchuk R A, Engle M W, Young B J, Conallen J, Houston K A 2008 Object-Oriented Analysis and Design with Application (ACM SIGSOFT Software Engineering Notes) p 720
[4] Taveira W F, de Oliveira Valente M T, da Silva Bigonha M A, da Silva Bigonha R 2003 Asynchronous Remote Method Invocation in Java. J. UCS 9 761-75
[5] Signals & Slots Qt Documentation URL - https://doc.qt.io/qt-5/signalsandslots.html
[6] Shalnev I O 2018 Proc. of IV Int. Scientific and Practical Conf “Technological perspective: New Markets and Points Economic Growth” (St.-Petersburg) (“Asterion”) 151-8