TELKOMNIKA Indonesian Journal of Electrical Engineering
Vol.12, No.6, June 2014, pp. 4521 ~ 4527
DOI: 10.11591/telkomnika.v12i6.5391

Object-Oriented Publish/Subscribe System

Biao Dong
Nanjing Institute of Industry Technology
No.1, Yangshan North Road, Qixia District, Nanjing 210023, China
e-mail: dongb@niit.edu.cn

Abstract

Existing methodologies based on the request/response model for system software design, rooted in principles of object-oriented design, lead to tightly-coupled interactions, and lack coordination capacities. The publish/subscribe (P/S) paradigm is particularly suitable for loosely-coupled communication environment. A design of event and subscription models under an object- oriented environment was motivated by the idea of using P/S paradigm for an object-oriented environment. The P/S service is comprised of two subsystem-compiler and executor. In each subsystem, its form, working process, and typical algorithms are analysed from the point of object-oriented and technology of P/S paradigm separately. Experiment was simulated in the following two aspects: user's model and middle code generation, and shows that using the P/S service improves the usability of P/S systems.

Keywords*: object-oriented, publish/subscribe, compiler subsystem, executor subsystem*

1. Introduction

In the distributed environment of enterprise internal network scale, the middleware based on remote procedure call (RPC) is successful [1], however, in the internet environment, it has the following problems:

(1) Communication model. The RPC-based middleware does not support the participants in the communications completely decoupling in space, time and control flow. Meanwhile, the communication model involved in client server interaction, is one to one model, isn't many to many model.

(2) Reactivity to input, exceptions and internal/external changes. The RPC-based middleware supports only a predefined serial and linear processing path, does not support dynamic (nonlinear, feedback processing), concurrent processing path. At the same time, it only reflects the closed environment, lack of ability to respond to external environment.

As the application environment of the local, closed and static state to Internet, open and dynamic, loosely-coupled and asynchronous implementation capacity in communication paradigm is an inevitable trend [2]. The publish/subscribe paradigm which offers asynchronous, multipoint communication, and support the participants of the communication completely decoupling in time, space and control flow, can well meet the large-scale, highly dynamic Internet environment [3, 4].

In an object-oriented environment[5], ways to realize P/S service has three kinds:

(1) Expansion method. Programmers determine where needs P/S service, and add the appropriate monitoring and operation code here.

(2) Reconstruction method. Through the reconstruction of the original programming language, programming language was transformed into a kind of programming language that supports P/S service.

(3) Redesign of P/S-based programming languages.

The first way is the most simple, but the system's P/S services are relatively simple and limited; the second approach is a compromise, P/S service function is limited by the source language; the third way is a complete solution, but due to the complexity of compiler itself, the workload for adding P/S service will be very heavy.

This paper proposes a P/S service model in an object-oriented environment. P/S service implementation approach was selected with a similar method to that of the third way.

Received December 26, 2013; Revised February 23, 2014; Accepted March 9, 2014

We don’t design a new object-oriented environment that support P/S service, but integrate P/S service into a known language model. This paper addresses two issues:

(1) The establishment of event model and subscription model in an object-oriented environment.

(2) The design and implementation of compiler and executor subsystem for an object-oriented environment that support P/S service.

2. Event Model and Subscription Model

In an object-oriented environment, communications between different objects are carried out by the event delivery. When an object receives an event, a method is called, therefore, the method calls can be treated as events. P/S system's events are considered as value events, a value event can be mapped to one or several method events.

Definition 1 method event. Method event in an object-oriented environment is defined as follows:

```
Create Event <Event> <EventClause>
<Event Clause>=[Before|After] <Name.MethodName>
```

The Event is a string uniquely identifying event, the Before and After events are produced before and after method execution, the Name is a class or object name, and the MethodName is a method name.

In an object-oriented system, method events are divided into pre-execution and post-execution method events, and include two kinds of method events: class method events and general method events. Class method events are generated before and after invoking class methods, such as creating an instance, deleting an instance. Similarly, general method events are generated before and after executing object methods.

Definition 2 subscription. Subscription in an object-oriented environment is defined as follows:

```
Create Sub <Subscription-Name>
On <Event-Trigger >
Condition: <Condition>
Act: <Action>
```

The Subscription-Name is a string uniquely identifying a subscription, the Event-Trigger is a list of Method events, the Condition is a Boolean function, and the Action denotes a procedure defined by an application program. As used in this definition, the subscription means as follows: when a method event in an event-trigger occurs, the condition is evaluated, and then if the condition is true, actions are executed.

3. Compiler Subsystem

3.1. Object Technology

Figure 1 shows a compiler subsystem for object oriented P/S systems.

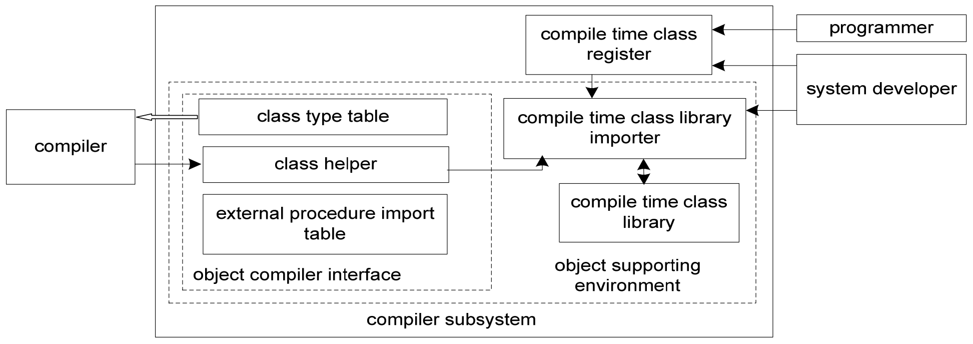
compile time class register
programmer
system developer
compiler
class type table
class helper
external procedure import table
object compiler interface
compile time class library importer
compile time class library
object supporting environment
compiler subsystem

Figure 1. Compiler subsystem

The working process of compiler subsystem is divided into two phases: the first phase is initialization, the second will focus on compiling. Now the compiler is capable of handling object.

The compiler subsystem completes the following works in the initialization phase: firstly, the compile time class library importer loads class library to provide support for object-oriented P/S systems on establishing a basic compiler time class library; secondly, Users import custom classes into the compile time class library by the compile time class library importer; finally, in the object compiler interface, the compiler subsystem establish the class type information, the class helper instance for each imported class, and relations between a class type and corresponding class helper instance.

Definition 3 external procedure. External procedures are virtual procedures that could be anything when compiled, including class methods, attributes, methods and events. External procedures are used to store class type information for the compiler subsystem.

The compiler subsystem completes the following works in the compile phase: when operations on objects are detected, the compiler checks class type tables whether class type information corresponding to the objects exists or not. If the class type information exists, the compiler finds the associated class helper which accesses the compile time class library through the compile time library importer. There are two main aspects to the class helper: helping the compiler for generating push instructions and filling in the import descriptions. According to the caller's parameter information acquired from the class items of the compile time class library, the compiler generates push instructions, creates or finds the number of the called external procedures, and then, fills in the import descriptions of the external procedures import table. If the class type information doesn't exist, the compiler provides false alarms. After the above works are completed, the compiler is capable of processing objects, and has been extended to the compiler subsystem.

3.2. Object Supporting Environment

The object supporting environment consists of three parts describing an approach to dealing with object. The first part is called the object compiler interface, which is divided into three sections: the class type table, the class helper, and the external procedure import table. The second part is called the compile time class library importer, which preserves lists of classes imported into the compile time class library. The last part is called the compile time class library, which is used to specify the compile time class sets and the compile time class item sets.

Each instance of a class type has its own class helper. The class helper provides support for the compiler to generate intermediate code at two ways. One is to invoke methods on a class. This second way is to support the basic operations of a class, such as cast, assignment, comparison and other operations. When the operations of class, including the calling class methods or the general methods, accessing properties, comparing objects, casting and assigning null, are handled, the compiler creates or finds an external procedure to handling this operation. If it's first time that the application calls the object's methods, properties and methods of class, the compiler will fill the appropriate procedure identification into the class items; if not, the compiler finds the procedure number. Method of class helper are classified into two categories:

(1) Finding method, which is directly used for finding the class item corresponding to the class method in the compile time class library according to the class method name.

(2) Calling method, which is directly used for creating an external procedure the application calls class methods.

Each external procedure corresponds to an import declaration. The import declaration used for the class library contains all information needed for external function registered to the compiler. When the compiler generates intermediate code, the import declaration must be written to the intermediate codes. When executor installs the external procedure, it has to read all information of external procedure. An import declaration is a string containing the following structure: class name+'|'+method name+'|'+cc+parameters, where cc is a calling convention, and the parameters decided by compile time class item is a string consisting of 1 and 0.

3.3. P/S Support for the Compiler Subsystem

In order to provide for P/S service support, the compiler subsystem completes two works. First of all, subscriptions are translated into the following procedure, and then the procedure are compiled into the intermediate codes.

```
Procedure <Subscription name> (<Triggering event parameter list>)
Begin
If <conditions> Then <actions>;
End.
```

Second, the association between event and subscription name is established.

4. Executor Subsystem

4.1. External Procedure Execution Technology

Figure 2 shows an executor subsystem for object oriented P/S Systems.

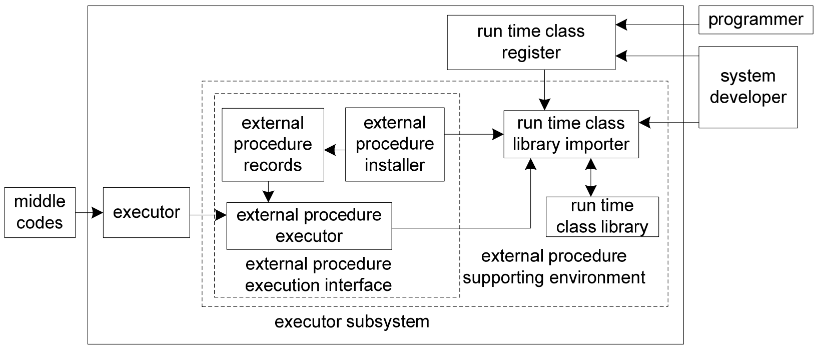
programmer
run time class register
system developer
external procedure records
external procedure installer
run time class library importer
middle codes
executor
external procedure executor
run time class library
external procedure execution interface
external procedure supporting environment
executor subsystem

Figure 2. Executor Subsystem

The working process of executor subsystem is divided into two phases: the first phase is initialization, the second phase will focus on execution. Now the executor is capable of handling external procedures.

The executor subsystem completes the following works in the initialization phase: firstly, the run time class library importer loads the classes to provide support for object oriented P/S systems on establishing basic run time class library; secondly, users import custom classes into the run time class library by run time class library importer; finally, in the external procedure execution interface, the external procedure installer sets up the external procedure records for each installed external procedure, and relations between the external procedures and the corresponding external procedure records.

The executor subsystem completes the following works in the execution phase: when the external procedure is called, the external procedure executor checks whether the external procedure record corresponding to the external procedure exists or not. If the external procedure record exists, the external procedure executor finds the associated external procedure record which accesses run time class library through the run time library importer. If the the external procedure record doesn't exist, the external procedure executor provides false alarms. After the above works are completed, the executor is capable of processing external procedures, and has been extended to the executor subsystem.

4.2. External Procedure Supporting Environment and Its Formal Description

During the execution phase of an external procedure, there are several general principles that should be followed.

(1) Calling conventions. The calling conventions influence two things: how parameters are passed to a function or procedure, how a caller's stack is cleaned up when the caller returns.

(2) Method call. Methods are functions and procedures that apply only to objects of a particular class and its descendants. Methods differ from ordinary procedures and functions in that every method has an implicit parameter called Self, which refers to the object that is the subject of a method call. Self is an address pointer. Within the implementation of a method, the identifier Self references the object in which the method is called.

(3) Returning results. There are two types of mechanisms for returning values. The first mechanism is to put the results in the ST(0) register of the float point unit(FPU), which corresponds to the top of the FPU stack. Secondly, put the results in the CPU registers.

(4) Constructors and destructors. Constructors and destructors use the same calling conventions as other methods, except that an additional Boolean flag parameter is passed to indicate the context of the constructor or destructor call. The flag parameter behaves as if it were declared before all other parameters.

(5) Virtual methods. A virtual method is a method that is bound at runtime instead of at compile time. At runtime, the method in the class's runtime tables, specifically the virtual method table(VMT), are looked up, and the method for the actual class is called. The actual class might be the compile time declared class, or it might be a derived class-it doesn't matter because a VMT provides the pointer to the correct method.

(6) Passing parameters. Passing parameters define a number of different aspects of method invokation: where parameters are located: in registers or on the stack, in which order parameters are passed: right to left or left to right, who is responsible for cleaning up the parameters afterwards, the caller or the callee.

Definition 4 external procedure supporting environment. The external procedure supporting environment EE=<T_1, T_2, CS_{run}, Stack, T_{other}>, where T_1: the external procedure record table, T_2: the class type table, CS_{run}: the runtime library; Stack: the data stack of the executor subsystem; T_{other}: the other tables in the executor subsystem.

Definition 5 execution vector. The external procedures can be performed, if and only if the external procedure supporting environment obtains an execution vector ∂=<∂_1, ∂_2, ∂_3>. Where: ∂_1 is a subset of the set {EAX, ECX, EDX, CurrStack}, which is associated with the input data, ∂_2 is a procedure pointer, ∂_3 is an element of the set {AL, AX, EAX, EDX: EAX, ST(0)}, which is used to save the returning results.

Definition 6 transform. Let $ExtPro_i$ be the current being invoked external procedure. There are six kinds of transforms defined as follows:

(1) Δ_1 (EE, $ExtPro_i$)=cc, where cc is a calling convention of the $ExtPro_i$.

(2) Δ_2 (EE, $ExtPro_i$)=Fself, where Fself is a parameter Self of the $ExtPro_i$.

(3) Δ_3 (EE, $ExtPro_i$)=n; where n is returning results of the $ExtPro_i$.

(4) Δ_4 (EE, $ExtPro_i$)=PList, which make EE to meet the fourth principle.

(5) Δ_5 (EE, $ExtPro_i$)=ptr, where ptr is a calling pointer of the $ExtPro_i$.

(6) Δ6 (EE, ExtProi)=ParList, which make EE to meet the sixth principle.

Theorem 1. Δ_i ($1 \le i \le 6$)) is achievable.

Proof: EE=<T_1, T_2, CS_{run}, Stack, T_{other}>, where T_1 is the external procedure record table, which has an external procedure import description consisting of a string that contains cc.For other transforms, we can verify one by one.

Theorem 2: The executor subsystem can perform external procedures.

Proof: By the transforms Δ_6, Δ_4, Δ_2 and Δ_1, the executor subsystem can get ∂_1. By the transform Δ_5, ∂_2 can be obtained. By the transform Δ_3, n can be obtained, and then according to the type of n, the executor subsystem can get ∂_3. Thus, By the transforms Δ_6, Δ_5, Δ_4, Δ_3, Δ_2 and Δ_1, the executor subsystem gets<∂_1, ∂_2, ∂_3>. Therefore, the executor subsystem can perform external procedures.

4.3. Structure and Process of the External Procedure Executor

The following conclusions can be made from the above analysis: First, the external procedure executor performs the transforms Δ_1, Δ_2, Δ_3, Δ_4 and Δ_5, the order of executing the transforms and the algorithms corresponding to the transforms must be specifically designed according to the type of the external procedures. Secondly, the external procedure executor performs the transform Δ6 to obtain <∂_1, ∂_2, ∂_3>. Finally, the external procedure executor performs the external procedure. Therefore, the external procedure executor has a hierarchical structure, which is divided into three parts from top to bottom, namely, the classification functions, the pre-handling and post-handling functions for the underlying call, and the underlying call functions.

The work processes of the external procedure executor are as follows:

(1) According to the external procedure record, the external procedure executor calls the corresponding classification function.

(2) The classification functions generate the call parameters according to the characteristics of each type of the call functions.

(3) According to the different calling conventions, method calls and parameter-passing modes, the external procedure executor pre-handles the call parameters, and post-handles the results returned by the underlying call functions.

4.4. P/S Support for the Executor Subsystem

The precondition for providing P/S services is that the object-oriented system must have a unified event detection mechanism. The timing of the pre-execution method events' detection is before calling the classification functions. Then, all of the required parameters including the external procedure identification, pointers and external procedure parameters have been produced, the executor subsystem calculate the required subscriptions according to the type of external procedure and the parameters at the top of stack. The timing of the post-execution method events' detection is at the end of calling the classification functions and before returning the caller. Then, the executor subsystem can get the generated method event's parameter values according to the stack of the external procedure executor and the type of the classification functions.

5. Simulation Experiment

By P/S mechanism, the simulation experiment realizes the following function: when the application displays a form f, a dialog box titled "Test" are displayed on screen. The experiment is discussed as follows:

(1) Coding to show a form f.

```
var f:TForm;
begin f:=TForm.CreateNew(f,0);
f.Show;
while f.Visible do
Application.ProcessMessages;
f.free;
end.
```

(2) Defining an event.

```
Create Event CreateForm After TForm.CreateNew(Aowner, Dummy);
```

(3) Defining a subscription.

```
Create Sub ShowForm
On CreateForm
Condition: Aowner=f;
Act: Begin
MessageBox(0,"Test","Form f!",0);
End.
```

The compiler subsystem generates intermediate codes for the form f and the subscription ShowForm, and then establishes association between the method event CreateForm and the subscription ShowForm. After the executor subsystem performs CreateNew(f, 0), a classification function is called. Before the function returns, the event CreateForm is triggered. The elements on the top of the external procedure executor's stack stands for the actual value that the executor subsystem want to use. Then the intermediate codes corresponding to subscription ShowForm are executed.

6. Conclusion

In this paper, we propose a compiler subsystem and an executor subsystem for object-oriented P/S systems. Event model and subscription model are defined in an object-oriented environment. We illustrates a conclusion through simulation. The result shows that the two subsystems can easily be constructed, and improve the usability of P/S systems.

Acknowledgements

This work was sponsored by Qing Lan Project 2012(Jiangsu Province, China) and open fund project of Jiangsu provincial research and development center of intelligent sensor network engineering technology, China(ZK11-02-01, research on publish in large-scale event system).

References

[1] Birman, Kenneth P. Guide to Reliable Distributed Systems. London: Springer London. 2012: 185-247.
[2] Kaye, Doug. Loosely coupled: the missing pieces of Web services. California: RDS Strategies LLC. 2003.
[3] Fotiou, Nikos, Dirk Trossen, George C Polyzos. Illustrating a publish subscribe internet architecture. *Telecommunication Systems*. 2012; 51(4): 233-245.
[4] Chen H, Kim M, Lei H, et al. Elastic and scalable publish/subscribe service. U.S. patent No. 20,130,007,131.
[5] Parkinson, Matthew, Gavin Bierman. Aliasing in Object-Oriented Programming. Types, Analysis and Verification. Berlin: Springer Berlin Heidelberg. 2013: 366-406.