Р10-2003-38

А. А. Карев, В. В. Галактионов, В. М. Добрянский

СПЕЦИФИКА ВЗАИМОДЕЙСТВИЯ КОМПОНЕНТОВ В ЗАДАЧАХ **АСУ ТП** И МЕТОДИКА РАСШИРЕННОГО ПРИМЕНЕНИЯ ТЕХНОЛОГИЙ MIDDLEWARE

Направлено в журнал
«Известия высших учебных заведений. Электроника»

1. Введение

Программы промежуточного слоя (middleware) обеспечивают интеграцию компонентов распределенной системы, которая заключается в сокрытии от прикладного разработчика взаимодействия этих компонентов. Промежуточный слой обеспечивает некий *стандартный набор* услуг по обмену различной сложности структурами данных, выполнению различной категории запросов в клиент-серверных взаимодействиях. В реальной жизни каждая распределенная система предназначена для выполнения конкретных задач, которые не всегда укладываются в стандарты выбранной технологии, т.е. существует проблема разрешимости нестандартных требований. Часть из них удается решить средствами самого промежуточного слоя, другая часть должна выполняться уже надстройкой – прикладным уровнем. С этой точки зрения классификация типов взаимодействия компонентов в распределенной системе АСУ ТП приведена в работе [7]. Если же рассматривать задачи АСУ ТП с точки зрения информационных технологий и с учетом их классификации [1,2], то промежуточный слой для АСУ ТП должен обеспечивать:

- выполнение нестандартных для технологий middleware запросов или *заявок* на выполнение операций;
- обмен структурированными типами данных (массивами чисел, потоками двоичных данных (массивами байтов) и др.).

В данной работе и будут рассмотрены вопросы возможности применения стандартных средств рассматриваемых технологий, а также возможностей выбранного языка программирования Java для решения этих задач. Будут приведены результаты практических исследований технологии **RMI** и **CORBA** (на примерах Java-реализаций пакета VisiBroker) [2-5]. В приведенных примерах указываются лишь фрагменты кодирования, поясняющие лишь суть демонстрируемого приема.

2. Управление заявками

В системах промежуточного слоя - **Java/RMI** и **CORBA** [6] - объекты-клиенты запрашивают выполнение операции у объектов-серверов. Эти запросы можно рассматривать как *заявки,* обладающие одинаковым свойством: они являются *синхронными* и затрагивают только объект-клиент и объект-сервер; они

определяются во время включения вызова *стаба* или *представителя* в код клиента; они же могут привести к ошибке, о которой клиент обязан получить уведомление. В реальной жизни некоторые *нефункциональные* требования (такие, как требования к производительности, универсальности или надежности) в общем случае не могут быть выполнены без использования *нестандартных заявок*. Эта часть проблемы управления заявками относится, главным обоазом, к формированию дополнительного программного слоя на клиентской стороне. Вторая проблема управления потоками заявок касается *обработки очередей* заявок в серверных объектах.

Примечание. Принципы реализации нестандартных заявок не зависят от промежуточного слоя и являются задачей вышестоящего *прикладного* уровня построения распределенной системы и зачастую зависят от *мощности* выбранного языка их программирования. Но естественным требованием к технологиям промежуточного слоя является предоставление *примитивов* для реализации нестандартных типов заявок разработчикам распределенных *приложений*.

Расширенные заявки обходятся дороже, как по причине большей сложности программирования, так и из-за увеличенного потребления ресурсов во время их выполнения. Рассмотрим возможности выбранных технологий для реализации нестандартных заявок с двух разных точек зрения: *синхронизации* заявки и *множественности* объектов, участвующих в выполнении заявки.

2.1. Синхронизация заявок

Стандартные объектные заявки, поддерживаемые **CORBA** и **Java/RMI** являются *синхронными*. Это означает, что объект-клиент блокируется во время выполнения запрошенной операции и управление возвращается к нему после того, как сервер завершит операцию или промежуточный слой выдаст уведомление об ошибке. При таком подходе однопоточные клиенты могут запрашивать выполнение операций только *последовательно*. Далее будет показано, как с помощью стандартного режима синхронных заявок можно реализовать наиболее используемые в распределенных системах три несинхронные формы взаимодействия:

- односторонние заявки,

- отсроченные заявки,
- асинхронные заявки.

Односторонние заявки. *Односторонние* (oneway) заявки позволяют вернуть управление клиенту сразу же после того, как заявка будет принята промежуточным слоем. Клиент продолжает функционировать одновременно с выполнением на сервере запрошенной операции. Сильно упрощая ситуацию, можно сказать, что декларированная в **IDL** объектным типом операция может быть определена как односторонняя только в том случае, если для возвращаемого значения метода указан тип **void**, отсутствуют параметры **out** или **inout** и не формируются типозависимые (пользовательские) исключения. Другой вариант реализации односторонних заявок заключается в организации *многопоточного* (multi-thread) режима работы клиента, при котором объект-клиент порождает дочерний поток и выполняет в нем синхронный (стандарный) запрос. Порожденный поток блокируется, но клиент продолжает обработку в родительском потоке.

Пример 1. Использование потоков Java для реализации одностронней заявки. В примере не выделяется специфика какой-либо системы middleware и приведенная схема может применяться как для RMI, так и для CORBA.

```
class  multiRequest {
   static void main (String[] args) {
      remoteObject objRef;
      . . . . .
      // инициализация объектной ссылки objRef опущена

      // инициализация объекта, реализующего поток
      oneWayRequest owr = new oneWayRequest(objRef, "Nick");

     // запуск потока
     owr.start();
    // продолжение работы клиентской программы
 }
}
```

```
//--- класс, реализующий поток для вызова удаленного метода
class oneWayRequest extends Thread {
    remoteObject ro;
    String name;

   //  конструктор класса
    oneWayRequest (remoteObject ro, String name) {
        this.ro = ro;
        this.name = name;
    }
    public void run () {
       ro.sayHello(name);  // вызов удаленного метода
    }
}
```

Пример 2. Задание для CORBA односторонней заявки в IDL-описании интерфейса сервера-объекта. Односторонние операции реализуются в CORBA клиентским стабом, сгенерованным на основе объявления операции в IDL как **oneway void**. Это означает, что данную форму синхронизации выбирает разработчик сервера, т.е. способ синхронизации определяется при компиляции определения интерфейса и генерации стаба.

```
module HelloApp
{
   interface Hello
   {
      oneway void  sayHello(in string message);
      string grtIP();
   };
};
```

Отсроченные заявки. Подача *отсроченных сихронных* (deferred synchronous) заявок аналогична односторонним заявкам. При этом клиент не блокируется и может выполнять другие вычисления, но несет сам ответсвенность за синхронизацию взаимодействия с сервером. Чтобы получить результат

выполнения заявки клиент выдает серверу стандартную синхронную заявку. Такая техника выполнения заявок называется *опросом* (polling).

Пример. Использование потоков Java для реализации отсроченных синхронных заявок. В примере не выделяется специфика какой-либо системы middleware и приведенная схема может применяться как для RMI, так и для CORBA.

```
class multiRequest {
    static void main (String[] args) {
        remoteObject objRef;
        . . . . .
        // инициализация объектной ссылки objRef опущена
        // инициализация объекта, реализующего поток

        oneWayRequest owr = new oneWayRequest(objRef, "Nick");

       // запуск потока
       owr.start();

      // ожидание окончания потока
      owr.join();

      // продолжение работы клиентской программы
      // запрос и печать результата выполнения заявки в порожденном
      // потоке
      System.out.println(owr.getResult());
 }
}
//  класс, реализующий поток для вызова удаленного метода

class oneWayRequest extends Thread {
    remoteObject ro;
    String name;
    String res = null;
```

```
  // конструктор класса
   oneWayRequest (remoteObject ro, String name) {
       this.ro = ro;
       this.name = name;
   }
   public String getResult() {
       return res;
   }
   public void run () {
     // вызов удаленного метода
      res = ro.sayHello(name );
     // прекращение потока
      stop();
  }
}
```

В CORBA реализован специальный режим выполнения отсроченных синхронных заявок. Они не могут выполняться с помощью клиентских стабов и реализуются в другом режиме работы брокеров объектных запросов ORB – с помощью интерфейса динамических вызовов (**DII,** Dynamic Invocation Interface).

Асинхронные заявки. *Асинхронные* (asynchronous) заявки являются улучшенным вариантом режима опроса (отсроченных заявок). В этом режиме клиент получает управление сразу же после принятия заявки промежуточным слоем. Сервер по завершении принятой заявки выполняет по своей инициативе передачу клиенту ее результатов. Такая операция называется также *обратным вызовом* (callback) и определяется клиентом при выдаче асинхронной заявки.

Идеи реализации режима **callback** в системах RMI и CORBA во многом совпадают: кроме декларации интерфейса удаленного сервера-объекта должен быть определен также интерфейс объекта-клиента в IDL (для CORBA) или Java-интерфейсе (для RMI), к которому возможно обращение из сервера-объекта.

Примечание: Для клиентского объекта также необходима регистрация в сервере именования.

Ниже будут приведены примеры реализации режима callback для RMI и CORBA. Это не совсем асинхронные заявки в полном понимании вышесказанного, поскольку первичное обращение к серверу-объекту выполняется

как обычная синхронная заявка. Но сделать полноценный пример с применением Java-потоков не представляет трудностей. В приложении приводятся полные тексты примеров для RMI и CORBA/Java IDL. Пример для RMI выполнен в виде Java-апплета и может запускаться стандартным Web-браузером.

Режим callback для CORBA

Задание интерфейса двух CORBA-объектов

В IDL-модуле задаются два интерфейса – для клиентского (**CallBack**) и серверного (**Hello**) объекта. Клиентский объект содержит один метод **callback()** с параметром, а серверный – один метод **sayHello()** с параметром, в котором серверному объекту передается объектная ссылка объекта-клиента

IDL-декларация интерфейсов

```
module HelloApp {
  //-----  интерфейс для клиентского объекта
  interface Callback {
     void callback(in string message);
  };
  //-----  интерфейс для серверного объекта
  interface Hello    {
     string sayHello(in Callback objRef);
 };
};
```

Фрагмент callback-объекта клиента.

```
//======    callback-объект клиента
class Callbackservant extends _CallbackImplBase {
   //------ метод запускается при обращении к нему из программы
   //       серверного  CORBA-объекта
    public void callback(String notification) {
      System.out.println(notification);
    }
}
```

Фрагмент программы клиента

```
public class HelloClient
{
   public static void main(String args[])  {
    . . . . . .
     // предполагается, что уже определены объекты ORB и объектная
     // ссылка на сервер-объект callRef

      //-----  создание клиентского объекта и его регистрация
       Callbackservant backRef = new Callbackservant();
       orb.connect(backRef);

      //-----  обращение к методу серверного объекта и передача в
      //        качестве параметра ссылки на клиентский объект
       String hello = callRef.sayHello(backRef);
       System.out.println(hello);
  }
}
```

Программа серверного CORBA-объекта

```
class HelloServant extends _HelloImplBase
{
   //------ конструктор объекта
   public HelloServant(String ns) {
       super(ns);
   }

   // этот метод вызывается из клиентской программы,
   //  в параметре сообщается объектная ссылка на callback- объект клиента
   public String sayHello(Callback callobj) {
```

```
        // обращение к методу callback() объекта клиента
        callobj.callback("Hello from server!");
        //------- возврат результата
        return "\nHello world !!\n";
    }
}
```

2.2. Множественность заявок

Традиционный тип заявок в распределенных системах – это одиночные (unicast) заявки, когда объект-клиент запрашивает выполнение операции только у одного объекта-сервера. Альтернативой этому типу заявок являются сложные заявки двух типов:

- *групповые заявки* (group request), когда клиент запрашивает одну и ту же операцию у нескольких объектов-серверов;
- *множественные заявки* (multiple request) при обращении к разным объектам-серверам с требованием выполнения разных операций.

Это один из наиболее сложных режимов выполнения операций в распределенных системах. Некоторые решения этой проблемы заложены в спецификациях CORBA, такие, например, как *служба событий* (Event service) для реализации групповых заявок на основе стандартных объектных заявок. Что же касается множественных заявок в CORBA, их нельзя определить статическим способом (с применением стабов), поэтому решение этой проблемы поддерживается интерфейсом динамических вызовов (DII). Такие заявки называются *множественными отсроченными сихронными заявками* (multiple deferred synchronous requests). В этом режиме возможно запрашивать выполнение операций у нескольких объектов-серверов CORBA с помощью операции send_multiple_requests, интерфейс которой определяется в **IDL** как:

```
module CORBA {
    interface ORB {
        typedef sequence<Request> RequestSeq;
        status send_multiple_request(in RequestSeq targets);
    }; };
```

Здесь надо подчеркнуть, что реализация спецификаций CORBA является прерогативой фирм-производителей. В практическом же плане наиболее простой вариант разрешения проблемы множественности заявок может быть реализован с применением возможностей языка программирования объектной технологии. В частности, в языке Java таким механизмом может служить свойство *многопоточности*. Основная идея такого подхода заключается в порождении параллельных потоков и выдаче в каждом потоке синхронной объектной заявки. Как только заявка будет выполнена, поток заявки **записывает** результат в *кортежную область* (tuple spaces). Клиентский основной поток **считывает** данные (кортежи) из этой области. Основная идея кортежных областей – наличие выделенного пространства для хранения помеченных данных, к которым имеют доступ одновременные процессы.

В приведенном ниже примере демонстрируется упрощенная схема построения и применения кортежных областей. Схема является универсальной и применима для любых технологий middleware, реализована она с использованием главным образом двух объектных структур языка Java:

- **Hashtable** – для манипуляций с именованными данными;
- **Vector** – для хранения данных.

Для доступа к данным кортежной области вводятся операции:

- **out** – для записи в область нового кортежа;
- **rd** – для считывания кортежа без удаления;
- **in** – для считывания кортежа с удалением его из области

Пример 1. Реализация операций с кортежными областями (фрагмент программы).

```
Hashtable tuples = new Hashtable();
public  void out(String key, Object data) {
   Vector v = new Vector();
   v.addElement(data);
   tuples.put(key, v);
}
 public  Object in(String key) {
    Vector v = (Vector) tuples.get(key);
    Object obj = v.firstElement();
    tuples.remove(key);
```

```
    return obj;
}
public  Object rd(String key) {
    Vector v = (Vector) tuples.get(key);
    Object obj = v.firstElement();
    return obj;
}
```

В примере 3 Приложения приводится полный пример реализации множественной заявки с использованием потоков и кортежных областей.

2.3. Управление очередями заявок

Задача сервера-объекта – принять и обработать заявку клиента в соответствии с синтакстсом IDL-интерфейса. В реальной жизни поступление заявок непредсказуемо, и в реализации методов серверного объекта должен быть предусмотрен механизм разрешения интерференционных ситуаций, т.е. взаимного влияния заявок при квазиодновременном их поступлении. Это не является задачей технологий промежуточного слоя, и решается она вышестоящим уровнем приложений. Алгоритм ее решения зависит от специфики задач данного компонента распределенной системы, и во многих случаях эффективный способ его реализации определяется выбраным языком программирования.

На языке Java для управления очередями заявок в серверном объекте можно предложить несколько способов:

- *синхронизация* очередей заявок средствами самого языка Java;
- создание *очередей* заявок и их обработку в самом серверном объекте;
- использование *многопоточного режима* языка Java для параллельного выполнения заявок;
- комбинирование многопоточного режима и создание очередей заявок. Это более сложное, но наиболее эффективное решение проблемы обслуживания множественных запросов, основанное на анализе их приоритетности.

Надо иметь в виду, что на некоторые типы заявок, допускающих создание очередей, могут быть наложены определенные ограничения.

Синхронизация очередей. Это наиболее универсальный, но не самый эффективный метод создания очередей заявок. Этот способ выполняет средствами языка Java блокировку метода объекта, реализующего заявку на время ее выполонения. Для тех заявок, время выполнения которых критично для подавших их клиентских приложений, это может быть неприемлемо. Выполняется этот тип заявок декларацией методов серверного объекта как **synchronized**. Это один из механизмов в Java - реализации синхронизации выполнения многопоточного режима выполнения задач. Распределенная среда является одним из видов многопоточного режима. Используя этот режим, можно синхронизировать отдельные методы объкета, как, например:

public synchronized void someMethod() { },

или одиночные данные:

```
synchronized (someDada) {
    // некоторые операции с этими данными
}
```

Надо отметить, в блокировке отдельных данных существует потенциальная опасность непроизвольного создания тупиковых ситуаций или *взаимных блокироваок* (deadlock), особенно при выполнении *рекурсивных* операций.

Очереди заявок. Этот метод предполагает безотлагательный прием заявки клиента, анализ ее и постановку ее в очередь. Реализация очередей типа **LIFO** или **FIFO** достаточно просто достигается средствами языка Java - объектными структурами – **Hashtable, Vector** и **Stack.** При этом параметры заявки для упрощения операций с очередями преобразуются в объектную структуру. Очевидно, что для этого варианта применимы *односторонние* либо *асинхронные* типы заявок

Пример формирования простой неструктурированной очереди:

```
class QueueElement {
    private int p1;
    private float p2;
```

```
    private String p3;
    public QueueElement(int p1, float p2, String p3) {
        this.p1 = p1;
        this.p2 = p2;
        this.p3 = p3;
    }
}
    // серверный объект
    Vector queue = new Vector();
    . . . . . .
    // метод серверного объекта
    public void firstMethod(int q1, float q2, String q3) {
       QueueElement qe = new QueueElement(q1, q2, q3);
       queue.addElement(qe);
    }
```

Многопоточный режим выполнения заявок. Этот режим предназначен для выполнения заявок с критичным временем ожидания клиента. Здесь также допустимы *односторонние* либо *асинхронные* типы заявок. Для выполнения каждой заявки запускается свой поток (класс **Thread)** или группа потоков (класс **ThreadGroup).** Здесь есть свои проблемы, связанные, в первую очередь с управлением многопоточного режима с непредсказуемым числом потоков (заявок), такими, как, например: идентификация и синхронизация потоков, управление приоритетами их выполнения. В языке Java имеется достаточно средств (примитивов) для эффективной реализации такого режима. Здесь можно также эффективно использовать рассмотренную в предыдущем разделе методику организации *кортежных областей.*

Пример демонстрации схемы запуска параллельного выполнения заявок:

```
class Request extends Thread {
    private int p1;
    private float p2;
    private String p3;
```

```
        // конструктор класса
        public Request(int p1, float p2, String p3) {
            this.p1 = p1;
            this.p2 = p2;
            this.p3 = p3;
        }
         public void run() {
                // выполнение заявки
         }
}
// метод серверного объекта
        public void firstMethod(int q1, float q2, String q3) {
           new Request(q1, q2, q3).start();
        }
```

3. Потоки данных в распределенных системах

В работе [7] были рассмотрены типы передаваемых данных в задачах АСУ ТП. В данном разделе будут рассмотрены приемы обмена данными этого типа в исследуемых технологиях распределенных систем. Надо сразу отметить, что проблемы организации обмена сложными типами данных в случае применения RMI-технологии не существует. Для RMI, представляющей однородную распределенную среду, нет особой разницы в передаче данных для локальных объектов или для удаленных серверных объектов. Для CORBA, по причине главного ее требования интероперабельности обмена данными, обмен неатомарными данными (числами, строками) требует применения усложненых приемов в рамках стандартных спецификаций **OMG/IDL**. Вторая проблема использования CORBA заключается в применении типов параметорв OUT и INOUT в методах удаленного объекта. Такая концепция не всегда поддерживается в популярных языках программирования, и для ее реализации используются специальные приемы.

Кроме того, пока не нашла практической реализации концепция *передачи объектов* в распределенной среде с CORBA-архитектурой. Однако с использованием свойств языка программирования, в частности свойств *сериализации объектов* в Java в большинстве случаев это оказывается

возможным. В данном разделе будут продемонстрированы оригинальные приемы решения перечисленных проблем.

3.1 . Передача массивов данных

В примере представлен образец обмена массивами данных (целых чисел и строк) между клиентской программой и серверным CORBA-объектом. Демонстрируются способы синтаксического задания массивов данных в интерфейсном IDL-модуле и применения классов типа **Holder**, генерируемых IDL-компилятором, для хранения сложных структур, каковыми являются массивы.

Файл описания интерфейса array.idl

В описании интерфейса применен опреатор **typedef** для создания нового типа данных – массивов для целых чисел и строковых данных. Приведено также описание четырех методов объекта (getInt(), getPutInt(), getString(), getPutString()), параметрами которых и возвращаемыми значениями являются массивы.

```
module Arrays {
  interface ArrayData  {
    typedef sequence<long> Idata;
    typedef sequence<string> Sdata;
    Idata getInt();
    Idata getPutInt(in Idata rd);
    Sdata getString ();
    Sdata getPutString(in Sdata rd);
  };
};
```

Фрагмент программы клиента

```
// ARef  - объектная ссылка на объект-сервер

//---  подготовка рабочих массивов данных-----
 int[]      intArr = new int[20];
```

```
String[] strArr = new String[20];

for(int j=0; j < 20; j++) {
        intArr[j] = j;
        strArr[j] = "String " + j;
}
//---- создание объектов типа Holder для хранения сложных типов
//      данных (массивов)
IdataHolder intDat = new IdataHolder(intArr);
SdataHolder strDat = new SdataHolder(strArr);

// обращение к методу getInt() удаленного объекта
// для получения массива целых чисел и распечатка результата
 int len = 0;
 int[]   outArr  =  ARef.getInt();
 len = outArr.length;
 for(int i = 0; i < len; i++)
      System.out.print(" " + outArr[i]);

 //---- обращение к методу getPutInt() для передачи массива чисел,
 //      получение и распечатка  результата
   int[]   outArr2 =  ARef.getPutInt(intDat.value);
   len = outArr2.length;
   for(int i = 0; i < len; i++)
      System.out.print(" " + outArr2[i]);

 //----- обращение к методу getString() для приема массива
 //      строковых данных и их распечатка
  String[] outStr  =  ARef.getString();
  len = outStr.length;
  for(int i = 0; i < len; i++)
      System.out.print(" " + outStr[i]);

 //----- обращение к методу getPutString() для передачи и приема
 //      массива строковых данных и их распечатка
```

```
String[] outStr2 =  ARef.getPutString(strDat.value);
 len = outStr2.length;
 for(int i = 0; i < len; i++)
     System.out.print(" " + outStr2[i]);
```

Сервер-объект с четырьмя методами

```
class ArrayServant extends _ArrayDataImplBase {
    public ArrayServant(String sn) {
      super(sn);
    }
    public  int[] getPutInt(int[] rd){

     //---- создание массива целых чисел для возврата результата
      int[] intarr = new int[rd.length];

     //   проверка принятого массива и заполнение
     //   массива результата
      for(int i=0; i < 20; i++) {
        intarr[i] = rd[i]*i;
        System.out.print(" " + rd[i]);
       }
      return intarr;
    }
    //-------------------------------------
    public  int[] getInt(){
      int[] intarr = new int[30];
      for(int i=0; i < 30; i++) {
        intarr[i] = i;
      }
      return intarr;
    }
    public  String[] getPutString(String[] rd){
```

```
      //---- создание массива строк для возврата результата
      String[] strArray = new String[rd.length];

      //---- проверка принятого массива и заполнение
      //      массива результата
      for(int i=0; i < 20; i++) {
        strArray[i] = "Result " + rd[i];
        System.out.print(" " + rd[i]);
      }
      return strArray;
    }
    //-------------------------------------
    public  String[] getString(){
      String[] strArray = new String[20];
      for(int i=0; i < 20; i++) {
        strArray[i] = "Result " + i;
      }
      return strArray;
    }
}
```

3.2. Передача структур данных

Пример содержит образец обмена между клиентской и серверной программой простейшей структурной единицей данных, состоящей из двух элементов – целого числа и текстовой строки.

Файл описания интерфейса st.idl

```
module STRUCTUR {
  interface stru   {
    struct st {          //------ описание структуры на языке IDL
      long ID;           //------ целое число
```

```
        string str;        //------ текстовая строка
      };
      //---- метод CORBA-объекта. Содержит два параметра типа IN :
      // ----                    строку текста и структуру
      //---  возвращаемый результат: новая структура
      st getStruct(in string s, in st struc);
   };
};
```

Файл с CORBA-объектом stServant.java

```
class stServant extends _struImplBase {
   //------  конструктор объекта
   public stServant(String str){
      super(str);
   }
   //----- метод объекта
   public st getStruct(String name, st struct) {
     System.out.println("Recieve struct: " + struct.ID + "," +
                        struct.str);
     st stu = new st(20, name);
     return stu;
   }
}
```

Фрагмент программы клиента

```
   // strRef  - объектная ссылка на сервер-объект
   //------  создание структуры
   st struct = new st(10, "TEST for structure");

   //------ обращение к методу getStruct CORBA-объекта с двумя
   //       параметрами
   st res = strRef.getStruct("Name", struct);
```

```
//------ распечатка полученного результата
System.out.println("Res: " + res.ID + "," + res.str);
```

3.3. Передача массивов структур данных

Пример содержит образец обмена между клиентской и серверной программами (CORBA-объектом) массивами, элементами которых являются структуры данных из предыдущего примера. В этом примере используются классы типа **Holder**, генерируемые IDL-компилятором, для хранения сложных структур данных (не примитивных, типа int, float и др).

Файл IDL-описания интерфейса

```
module V {
  interface Ve   {
    struct vectorItem {   //-- описание элемента массива
        long ID;
        string str;
    };
    //----- нового типа данных как массива структур
    typedef sequence<vectorItem> Vect;

   //------ описание двух методов putVector и getVector
   //------ оба с параметрами типа IN
    void putVector(in Vect vec);
    //----- возвращаемый результат  – новый массив структур
    Vect getVector(in string st);
  };
};
```

Фрагмент программы клиента

```
    // VRef  - объектная ссылка на серверный CORBA-объект

    // Создание и заполнение массива структур
    vectorItem[] vit = new vectorItem[10];
```

```
VectHolder vv = new VectHolder(vit);

for(int j=0; j < 10; j++) {
   String st = "###" + j + j + j;
   vv.value[j] = new vectorItem(j, st);
  }
  //---- передача созданного массива методу putVector()
  VRef.putVector(vit);

  //---- Прием массива структур от метода -- getVector()
 vectorItem[] vitm = VRef.getVector("Name");

 //---- распечатка принятого массива структур
  int len = vitm.length;
  for(jj=0; jj< len; jj++) {
      String sr = vitm[jj].str;
      k = vitm[jj].ID;
      System.out.println("ID=" + k + ", text=" + sr);
  }
```

Файл CORBA-объекта VServant.java

```
class VServant extends _VeImplBase {
    public VServant(String str){       // конструктор CORBA-объекта
     super(str);
    }
    //-------- первый метод putVector().
   //  Входной параметр – массив структур
    public void putVector(vectorItem[] vit) {

     //---- проверка принятого массива
      int l = vit.length;
      for(int i = 0; i < l; i++) {
             int   jj = vit[i].ID;
             String str = vit[i].str;
```

```
            System.out.println(str + " " + jj);
    }
   }
  //----- второй метод, возвращаемый результат – массив структур
   public vectorItem[] getVector(String name) {
     Vector list = new Vector();
     String str;
     int jj;

    //----  промежуточная структура данных  (типа Vector) для результата
     for(int i=0; i < 10; i++) {
        str = "";
        for(int j=0; j<10; j++) str = str + i;
        list.addElement(str);
     }
    //--- создание массива для возврата результата
     vectorItem[] vit = new vectorItem[10];
     VectHolder vv = new VectHolder(vit);

      //------ заполнение массива
      jj = 0;
      for(Enumeration en = list.elements(); en.hasMoreElements(); jj++ ) {
            String st = "$$$" + en.nextElement().toString();
            vv.value[jj] = new vectorItem(jj, st);
      }
     return vit; // возврат результата
   }
}
```

3.4. Передача результатов в параметрах типа INOUT и OUT

Параметры типа **INOUT** предназначены для передачи данных методу CORBA-объекта и для размещения в них же возвращаемого результата. Параметры типа **OUT** предназначены только для возвращаемого результата.

Проблема параметров рассматриваемого типа в языке Java заключается в том, что значения в параметрах функций передаются как **by value**, т.е. в них нельзя размещать никаких данных для возврата результатов. Поэтому для реализации требований CORBA приходится использовать даже для простых переменных специальные конструкции в виде *классов,* которые передаются в качестве параметров типа **by reference**, т.е. в виде объектных ссылок. Для этой цели при компиляции IDL-интерфейсов автоматически генерируются классы типа **Holder,** которые собственно и используются для передачи сложных параметров, в том числе и параметров типа INOUT и OUT. Значения данных в этих классах содержит переменная со стандартным именем **value**. Для примитивных типов данных генерируются классы этого типа со стандартными именами, в которые включены символические имена таких данных: **IntHolder, ByteHolder, FloatHolder, DoubleHolder, CharHolder, StringHolder** и т.д.

В примере использованы три метода с применением параметров типа INOUT и OUT:

- string getput(in string st, **out** string os, **inout** string ios). В параметре ios метод получает входное значение текстовой строки, и туда же может быть помещена другая строка в качестве возвращаемого значения.
- Следующие два метода также демонстрируют использование параметров типа INOUT и OUT, но для передачи уже массивов целых чисел

  void writea(**inout** data wrdata);
  void writeb(**out** data ar);

Файл IDL-описания интерфейса

```
module in_out {
  interface INOUT   {
  typedef sequence<long> data;
  void writea(inout data wrdata);
  void writeb(out data ar);
    string getput(in string st,  out string os,  inout string ios);
  };
};
```

Файл CORBA-объекта inoutServant.java

```
class inoutServant extends _INOUTImplBase {
```

```
public inoutServant(String ns) {
  super(ns);
}
//------ метод с параметром типа INOUT ------------------------------
public void writea(dataHolder wr){
     System.out.println("write() started!" + wr.toString());

    //-- проверка принятого значения массива и запись в этот же
    //   массив нового значения в качестве возвращаемого результата.
      for(int j=0; j < 10; j++) {
       System.out.println("get=" + wr.value[j]);
       wr.value[j] = j*j;
      }
 }
//---------метод с параметром типа OUT -----------------
public void writeb(dataHolder wr){

 //---- создание нового значения переменной value
 //      в классе типа Holder
  wr.value = new int[20];

 //---- запись возвращаемого результата
  for(int j=0; j < 20; j++)
        wr.value[j] = j*j;
 }

//------ метод с параметрами os (типа  OUT) и ios (типа  INOUT)
public String getput(String st, StringHolder os,StringHolder  ios) {

   //--- запись возвращаемого результата через переменную value
  os.value = " 111111111111 ";
  ios.value = ios.value.toUpperCase();
   return st.toUpperCase();
} }
```

Фрагмент программы клиента

```
// ioRef – объектная ссылка на CORBA-объект сервера

//   метод writea(inout p)
//---- передача массива целых чисел в параметре типа INOUT и
//     прием результата в этом же иассиве
 int intar[] = new int[10];
 for(int j=0; j < 10; j++)   intar[j] = j;

 dataHolder outArray = new dataHolder(intar);
 ioRef.writea(outArray);

 //----- проверка результата
 for(int i=0; i < 8; i++) {
    System.out.println("Output element out array [" + i +"]="
                       + outArray.value[i]);
 }

//   метод void writeb(out data ar) -   параметр типа OUT
 dataHolder outb    = new dataHolder();
 ioRef.writeb(outb);

 // обработка и распечатка результата
 int[] res = outb.value;
 int le = res.length;
 for(int i=0; i < le; i++) {
      System.out.println("res=" + res[i]);
 }

 // метод getput(in string st,  out string os,  inout string ios)
 String          st        = new String("qwertyuiopasdfghjklzxcvbnm");
 StringHolder outstr    = new StringHolder();
 StringHolder inoutstr  = new StringHolder("java station");
```

```
//---- передача строки st в качестве параметра типа IN
//      и строки  inoutstr в качестве параметра типа INOUT

String Hello = ioRef.getput(st,outstr, inoutstr);

// прием результата:
// строка Hello – как  возвращаемый  результат функции,
// строковые данные outstr и inoutstr - как  результат в параметрах
// типа  INOUT и OUT

String ou = outstr.value;
String ios = inoutstr.value;
System.out.println(Hello + ou + ios);
```

3.5. *Передача потоков байтов. Сериализация и передача Java-объектов*

Передача массивов байтов практически не отличается от передачи массивов целых чисел, рассмотренных в одном из предыдущих примеров. Разница заключается в синтаксисе описания массивов в IDL-модуле, например:

```
typedef sequence<octet> bytes
```

Важность такого обстоятельства заключается в открывшейся возможности **передачи объектов**. Под объектом понимается классическое представление программных структур, принятое в объектно-ориентированных языках программирования, т.е. инкапсулированные в одной структуре данные и программы (методы) для доступа к этим данным. В спецификациях CORBA пока не определен механизм передачи объектов в распределенных гетерогенных системах. Вся сложность этого явления заключается в провозглашенном в архитектуре CORBA принципе *интероперабельности*, т.е возможности взаимодействия программ клиента и сервера, написанных на разных языках с соблюдением, однако, тем не менее требования объектной ориентации. Естественно, соблюсти совместимость программных объектов при их передаче в таких случаях пока не представляется возможным. Тем не менее вопрос о передачах объектов стоит в повестке дня дальнейшего развития CORBA-систем.

При использовании же однородных распределенных систем, например, разрабатываемых с использованием единого языка Java, передача объектов

допускается. Здесь может быть учтен опыт применения системы **RMI**, используемой для распределенных систем на Java-платформах. В CORBA-приложениях, разрабатываемых на языке Java, может быть эффективно использовано замечательное свойство Java-объектов – возможность их **сериализации.** Это означает, что в Java предусмотрен механизм преобразования объектов в последовательность байтов и обратного их восстановления. Эта процедура во многом совпадает для различных Java-объектов, но нельзя однозначно ответить на вопрос о ее применимости для всех объектов.

В рассматриваемом ниже примере приводятся в качестве наглядного пособия следующие действия:

- создание в клиентской программе графического объекта типа Frame (окна) с компонентами GUI (графического интерфейса пользователя) и программ для обработки событий;
- сериализация этого объекта;
- передача этого массива байтов программе-методу серверного CORBA-объекта;
- восстановление на сервере принятого массива в графический объект типа Frame, модификация его и запуск в работу;
- выполнение клиентской программы запроса на прием такого же объекта от серверной программы и выполнение аналогичных действий (сериализация объекта, передача его и восстановление) программами клиента и сервера.

В тексте программы даны необходимые пояснения в виде стандартных комментариев. В примере 4 Приложения приводится полный текст примера.

Файл IDL-описания интерфейса

```
module Serial {
  interface bStream   {
     typedef sequence<octet> bytes;
     void writeStream(in bytes  array);
     bytes readStream();
  };
};
```

Фрагмент программы клиента

```
  // strRef  - объектная ссылка на CORBA-объект
```

```
//---- создание графического объекта (окна)
Frame ft = new setFrame("Original frame");

//=======  сериализация и передача объекта ======
try {
  //-----  сериализация объекта типа Frame и размещение потока
  //       байтов в массиве bb
   ByteArrayOutputStream o = new ByteArrayOutputStream();
   ObjectOutput os = new ObjectOutputStream(o);
   os.writeObject(ft);
   byte[] bb = o.toByteArray();

   //--------  передача Frame-объекта CORBA-методу .writeStream()
   strRef.writeStream(bb);
} catch (Exception ex) { }

//  чтение и восстановление объекта
try {
  //-------- обращение к методу readStream() CORBA-объекта
  ByteArrayInputStream bs =
     new ByteArrayInputStream(strRef.readStream());
  ObjectInput is = new ObjectInputStream(bs);
  java.lang.Object obj = is.readObject();
  is.close();
} catch(Exception ex) {    }
```

Файл с методами удаленного CORBA-объекта

В программе выполняются операции с графическим объектом типа Frame (сериализация и восстановление), аналогичные и симметричные клиентской программе. Поэтому комментарии к программе сокращены.

```
class streamServant extends Serial._bStreamImplBase {
   Frame f;
   java.lang.Object obj;
```

```
    //----- конструктор объекта --------
    public streamServant(String str){
        super(str);
    }
    //---------метод readStream() в качестве результата отправляет массив
    //          байтов.
    //          В данном примере передается графический объект после его
    //          сериализации
    //          Логика прмера построена на том, что этот метод возвращает
    //          клиентской программе полученный от нее ранее графический
    //          объект типа Frame
public byte[] readStream() {
          try {
          ByteArrayOutputStream ov = new ByteArrayOutputStream();
          ObjectOutput osv = new ObjectOutputStream(ov);
          f.setTitle("Send from server");
          osv.writeObject(f);
          return ov.toByteArray();
} catch (Exception ex) { System.out.println("Server exception");
                    return null;
       }
 }
 //------ метод writeStream() в качестве входного параметра (массива байтов)
 //        получает графический объект типа Frame, выполняет его
 //        идентификацию и восстанавливает его изображение.
   public void writeStream(byte[] vct){
      try {
          ByteArrayInputStream bs = new ByteArrayInputStream(vct);
          ObjectInput is = new ObjectInputStream(bs);
          obj = is.readObject();
          is.close();
      } catch (Exception e) {    }
}
```

4. Заключение

Приведенные в работе результаты исследования промышленных средств развертывания распределенных систем показывают **применимость** этих технологий для создания интеграционной среды (архитектуры) для многомашинного комплекса проектируемой автоматизированной системы управления технологическими процессами.

Здесь же были проанализированы виды взаимодействия компонентов (машин или процессов) и обмена данными в задачах АСУ ТП, и результат анализа сформулирован в виде системы **типизированных заявок** и **потоков данных**. Показано, что некоторые типы взаимодействия являются *нестандартными* для программных реализаций middleware. Для разрешения этой проблемы **предложены методы** их реализации с применением как специальных средств языка программирования, так и примитивов исследуемых объектных технологий. На модельных примерах показана **работоспособность** предложенных методов.

Литература

1. Липаев В.В., Яшков С.Ф. Эффективность методов организации вычислительного процесса в АСУ. М.: Статистика, 1985. 205с.
2. Object Management Group. http://www.omg.org/.
3. OMG Specifications and process: the big picture. http://www.omg.org/gettingstarted/overview.htm.
4. Аншина М. Увлекательное путешествие с CORBA 3: по широким просторам распределенных приложений. Журнал "Открытые системы", №05-06,Изд-во "Открытые Системы", Москва,1999 .
5. Эммерих В. Конструирование распределенных объектов. Изд-во “Мир”, Москва, 2002.
6. Галактионов В.В. Java-технология в распределенных системах с CORBA-архитектурой. Сообщение ОИЯИ, Р10-2002-63, Дубна, 2002.
7. Карев А.А., Галактионов В.В., Добрянский В.М. Методы и средства программирования интероперабельных объектов для задач АСУ ТП. Сообщение ОИЯИ Р10-2003-37, Дубна, 2003.

Карев А. А., Галактионов В. В., Добрянский В. М. Р10-2003-38
Специфика взаимодействия компонентов в задачах АСУ ТП
и методика расширенного применения технологий middleware

В работе исследованы специфические для АСУ ТП виды взаимодействия в распределенной системе (нестандартные типы заявок и потоков данных) и предложены методы их реализации с применением примитивов объектных технологий промежуточного слоя Java/RMI и CORBA.

Работа выполнена в Научном центре прикладных исследований ОИЯИ.

Препринт Объединенного института ядерных исследований. Дубна, 2003

Перевод авторов

Karev A. A., Galaktionov V. V., Dobrianski V. M. P10-2003-38
Particularity of the Distributed Component Interconnection
in Control Engineering Tasks and Method for Wider Usage
of the Middleware Technologies

In this paper the investigation of specific types of the component interconnection for distributed system in control engineering (nonstandard queries and dataflow types) was performed. Implementation methods, using primitive objects Java/RMI and CORBA middleware technologies, were prepared.

The investigation has been performed at the Scientific Center of Application Research, JINR.

Preprint of the Joint Institute for Nuclear Research. Dubna, 2003

Редактор *М. И. Зарубина*
Макет *Н. А. Киселевой*

Подписано в печать 31.03.2003.
Формат 60 × 90/16. Бумага офсетная. Печать офсетная.
Усл. печ. л. 1,87. Уч.-изд. л. 2,48. Тираж 290 экз. Заказ № 53835.

Издательский отдел Объединенного института ядерных исследований
141980, г. Дубна, Московская обл., ул. Жолио-Кюри, 6.
E-mail: publish@pds.jinr.ru
www.jinr.ru/publish/