МАРШАЛИНГ ДАННЫХ В РАСПРЕДЕЛЕННЫХ СИСТЕМАХ: СРАВНЕНИЕ ДВУХ ПОДХОДОВ

К.В.Дышлевой, В.Е.Каменский, Л.Б.Соловская

При организации вычислений в распределенных системах передача данных может происходить между компонентами, использующими различные правила представления данных. При этом возникают “накладные расходы” на преобразование (маршалинг) данных. В статье обсуждаются проблемы и приводятся конкретные примеры, связанные с реализацией процедур маршалинга.

1. Введение

Большинство современных программных продуктов основано на технологии распределенных вычислений. Такие распределенные системы реализуются в виде компьютерных сетей, совместно использующих ресурсы и обеспечивающих передачу информации между входящими в них компонентами. Для описания распределенных вычислений на абстрактном уровне может служить модель клиент-сервер. В терминах этой модели, клиентское приложение обращается к серверному с запросом, включающим все необходимые параметры. Сервер обрабатывает полученный запрос и возвращает обратно результаты. При этом, передача параметров и результатов может осуществляться между различными платформами, использующими свои собственные правила представления данных. Понятие платформы включает в себя не только аппаратуру и операционную систему, но и язык программирования, и конкретный компилятор этого языка.

Клиент может обращаться к серверу напрямую. В этом случае, все действия, связанные с передачей сообщений, клиент и сервер должны осуществлять сами по своему собственному протоколу. Для упрощения и унификации взаимодействия между клиентом и сервером можно воспользоваться услугами различных систем-посредников, например, ONC RPC, DCE RPC, OMG CORBA.

Такие посредники для передачи данных по сети используют определенные правила кодировки и протоколы взаимодействия. Существуют различные стандарты кодировок данных, например, CDR, BER, NDR, XDR, и соответствующих протоколов, GIOP, OSI RPC, ONC RPC, DCE RPC.

Таким образом, посредник, управляющий взаимодействием между клиентом и сервером, должен уметь преобразовывать данные из одного представления в другое. Процедура трансформации данных для последующей передачи по сети получила название маршалинга *(marshalling)* данных. Говорить о передаче по сети можно не ограничивая общности задачи. Заметим только, что проблема преобразования разнородных данных в битовый поток возникла намного раньше и имеет самостоятельное значение.

При распределенных вычислениях неизбежно возникают “накладные расходы” на организацию маршалинга данных. Особенно это заметно при работе в локальных сетях, где время передачи данных непосредственно по сети может быть значительно меньше, чем время, затраченное на маршалинг.

Для повышения эффективности процедур маршалинга существуют различные подходы. Например, можно улучшить стандартные правила кодирования с целью сокращения размера передаваемых по сети данных и уменьшения времени маршалинга [3], или реализовать наиболее эффективно базовые процедуры маршалинга на аппаратном уровне, или по возможности максимально оптимизировать процедуры маршалинга средствами программного обеспечения.

Наиболее общим подходом к реализации процедур маршалинга на программном уровне является выбор между возможностями интерпретации и компиляции. И интерпретация, и компиляция имеют свои хорошо известные достоинства и недостатки. Интерпретируемый код компактен, но сама процедура интерпретации достаточно медленна. Выполнение скомпилированного кода намного быстрее, но при этом размер кода может существенно увеличиться.

Рассмотрим простейший модельный язык *L*. Он состоит из двух базовых типов *T1* и *T2* и одного составного типа *struct,* аналогичного типу *struct* языка C. Пользователь может определять любые структуры произвольной вложенности. Примеры таких

структур - *struct S1 { T1 f1; T2 f2 }* и *struct S2 { S1 d1; T1 d2 }*. Значения специального типа *TypeInfo* описывают внутреннюю структуру типов, кодируя последовательно их элементы. Значения типа *T1* кодируются 1, значения типа *T2* - 2. Код значений составного типа *struct* начинается с 3, заканчивается 0 и содержит внутри себя последовательность кодов своих полей. Например, код для типа *S1* - “3120”, для типа *S2* - “3312010”.

Простейший интерпретатор маршалинга, умеющий работать со значениями любого типа языка L, может выгдядеть так:

```
typedef char * TypeInfo;
void interpr(TypeInfo *t, void *data) {
    while(**t) {
        switch(**t) {
        case ‘1’:
            put_T1(data);
            data = (char *) data + sizeof(T1);
            (*t)++;
            break;
        case ‘2’:
            put_T2(data);
            data = (char *) data + sizeof(T2);
            (*t)++;
            break;
        case ‘3’:
            (*t)++;
            interpr(t, data);
            break;
        case ‘0’;
            return;
        }
    }
}
```

Чтобы использовать скомпилированный код для маршалинга, необходимо порождать специальные процедуры для каждого типа, определенного пользователем. Например, для типов *S1* и *S2* процедуры маршалинга будут иметь вид:

```
put_S1(S1 * data) {
    put_T1(&(data->f1));
    put_T2(&(data->f2));
}

put_S2(S2 * data) {
    put_S1(&(data->d1));
    put_T1(&(data->d2));
}
```

В статье предлагаются конкретные примеры реализации обоих методов и результаты сравнения их возможностей на некоторых тестовых данных. Процедуры маршалинга рассматриваются в контексте разработки брокера объектных запросов *(Object Request Broker, ORB)*.

Второй раздел более детально вводит понятие маршалинга. Основные понятия контекста реализации сфомулированы в третьем разделе. Четвертый и пятый разделы посвящены некоторым деталям реализации маршалинга на основе интерпретации и компиляции соответственно. Экспериментальные данные для обоих методов обсуждаются в шестом разделе.

2. Понятие маршалинга

Маршалинг - это преобразование данных из одного формата в другой по определенным правилам, обычно, для передачи по сети. При этом преобразовании должны учитываться следующие аспекты представления данных.

Различные платформы могут использовать свои собственные соглашения о кодировке символов (ASCII, EBCDIC кодировка), формате целых чисел и чисел с плавающей точкой (IEEE, VAX, Cray, IBM). Например, возможна следующая схема: для представление целых типов *(short, long)* используется дополнительный код, представление типов *float* и *double* удовлетворяет стандарту ANSI/IEEE, а символьный тип *char* имеет множеством значений ISO Latin/1 символы (расширенное множество символов ASCII кодировки для Западной Европы).

Порядок байтов, как правило, зависит от процессора (*Big Endian* или *Little Endian)*. Существуют процессоры поддерживающие обе возможности (MIPS, UltraSPARC, PowerPC) в зависимости от требований различного программного обеспечения. При передаче данных необходимо учитывать возможное несовпадение порядка байтов.

Существуют различные стратегии выравнивания данных в памяти. Например, компилятор может требовать, чтобы значения базовых типов размещались с адреса, кратного их действительному размеру в байтах (1, 2, 4, 8, 16). Возможно выравнивание границ данных по наибольшей длине, требуемой для размещения значений базовых типов (например, 16). Таким образом, в памяти могут появляться “дырки”, предшествующие некоторой полезной информации. Естественно, что такие “дырки” должны иметь минимальный размер, необходимый для корректного выравнивания последующих данных.

Размещая в памяти сложные структуры данных, некоторые компиляторы могут оптимизировать расположение их полей. При необходимости, процедуры маршалинга должны согласовывать различные стратегии выравнивания и размещения данных.

Маршалинг также включает линеаризацию сложных структур данных, элементы которых могут располагаться не в смежных областях памяти, например, динамических деревьев.

Процедуры маршалинга должны иметь представление о формате данных для конкретной платформы, уметь преобразовывать его в некоторый определенный формат для передачи по сети, получать данные из сети и преобразовывать их обратно в требуемый платформой формат. Таким образом, маршалинг включает две взаимно обратные процедуры: кодирования и декодирования. Алгоритмы этих процедур достаточно симметричны с точностью до базовых операций “положить” *(put)* / “взять” *(get)* данное из некоторого буфера. Хотя процедура декодирования может быть несколько сложнее за счет дополнительных действий (например, отведения памяти).

3. Контекст применения: основные понятия

3.1. Брокеры объектных запросов

Проблемы реализации процедур маршалинга обсуждаются на примере разработки брокера объектных запросов. Распределенные системы рассматриваются в рамках объектно-ориентированного подхода.

Для того, чтобы объект мог прозрачным образом (независимо от платформы) обращаться с запросом к другому объекту, необходимо вмешательство некоторого посредника. Таким посредником, управляющим взаимодействием между клиентским и серверным объектами, является брокер объектных запросов.

В функции брокера объектных запросов входит:

- нахождение серверного объекта;
- упаковка запроса и параметров и их передача;
- ожидание результатов, распаковка и передача этих результатов клиентскому объекту.

Дополнительно к этим основным функциям, брокер объектных запросов занимается вопросами защиты, хранения данных, синхронизации взаимодействия и другими проблемами, связанными с распределенной средой.

Как правило, в таком взаимодействии принимают участие два брокера объектных запросов: один на стороне клиента, другой на стороне сервера. Клиентский брокер передает запрос, полученный от объекта-клиента, на сторону сервера и принимает результаты. Брокер на стороне сервера, в свою очередь, принимает запрос, передает его серверному объекту и возвращает результаты. Если объект является и клиентом, и сервером, то используемый им брокер объектных запросов должен уметь передавать и принимать сообщения.

Предлагаемый к рассмотрению брокер объектных запросов и объекты, работающие с этим брокером, реализованы на языке C++. Брокер поддерживает стандарт брокеров объектных запросов OMG CORBA.

3.2. Стандарт OMG CORBA: IDL, GIOP, CDR

CORBA - стандарт консорциума OMG для создания брокеров объектных запросов [1]. Целью организации консорциума OMG (Object Managment Group) было создание открытых стандартов, обеспечивающих совместную работу объектно-ориентированного программного обеспечения в распределенной среде. Стандарт CORBA, самая известная разработка OMG, определяет распределенные объектные средства, позволяющие приложениям взаимодействовать прозрачным образом (независимо от платформы) на основе совместимых с CORBA брокеров объектных запросов.

Спецификация CORBA построена на объектной модели OMG. Это классическая модель взаимодействия клиент-сервер на основе механизма передачи сообщений. Важно заметить, что в этой модели интерфейсы отделены от реализаций, и сама модель определяет только интерфейсы в терминах языка определения интерфейсов IDL (Interface Definition Language).

IDL - это средство формального описания спецификаций в рамках объектной модели OMG. Основным назначением IDL является представление интерфейсов, независимое от конкретных языков и включающее полные сигнатуры методов (имя объекта, имя метода, типы и способы передачи параметров и тип возвращаемого результата). Спецификации на IDL отображаются в язык реализации клиентских и серверных объектов. Описание интерфейсов на IDL напоминает описание классов на C++. IDL - это чисто декларативный язык, лишенный конструкций, ориентированных на описание алгоритмов.

В IDL реализована строгая концепция типов. Все типы, допускаемые синтаксисом, можно разделить на две группы: базовые и составные. К базовым типам относятся: целочисленный тип (*signed* и *unsigned integer*), 32 и 64-разрядные числа с плавающей точкой IEEE, символы ISO Latin/1, логический тип *boolean* и некоторые другие. *Octet* - специальный 8-разрядный базовый тип данных, не требующий преобразования при переносе с одной платформы на другую. Составные типы - это типы более высокого уровня, задаваемые пользователем. К составным типам относится тип *interface* - основной тип IDL. Среди остальных составных типов можно назвать *struct*, *union*, *sequence* и *array*. Тип *struct* определяет

структуры данных, аналогичные структурам языка C, *sequence* и *array* - массивы из элементов одного типа переменной и фиксированной длины соответственно. Тип *union* семантически напоминает *union* C, дополненный дескрипторами вариантов.

На более низком уровне CORBA определяет стандарт взаимодействия и представления данных посредством GIOP (General Inter-ORB Protocol), протокола взаимодействия между брокерами объектных запросов. Используя, например, наиболее распространенный и гибкий протокол TCP/IP, GIOP определяет минимум дополнительных требований, необходимых для передачи сообщений между брокерами объектных заявок. Спецификация GIOP включает следущие элементы:

- CDR (Common Data Representation) - формат представления данных;
- непосредственно формат сообщений GIOP ;
- требования к транспортному уровню.

IIOP (Internet Inter-ORB Protocol) - дополнительный протокол, описывающий механизм использования протокола TCP/IP для передачи сообщений GIOP. IIOP - это частный случай отображения GIOP на конкретный транспортный протокол. А сам GIOP является аналогом IDL, но уже на транспортном уровне.

CDR задает некоторый набор правил представления данных, включающий использование специального флага в сообщениях GIOP для указания используемого порядка байтов и соглашение о выравнивании базовых типов на их действительную границу. Правила CDR охватывают все множество типов, допустимых IDL.

Спецификация CDR включает понятие потока октетов. Поток октетов - это некоторая абстракция буфера в памяти, который будет передан в другой процесс или на другую машину.

4. Маршалинг на основе интерпретации TypeCode

4.1. Понятие TypeCode

Маршалинг значений базовых типов осуществляется набором базовых примитивов, которые используются процедурами маршалинга более сложных типов. Реализация этих базовых примитивов предполагает следующий формат представления данных:

• дополнительный код для представление целых чисел;

• числа с плавающей точкой стандарта ANSI/IEEE;

• символы ISO Latin/1;

• схема выравнивание значений базовых типов не зависит от того, являются ли они компонентами составных типов или самостоятельными данными; все известные нам компиляторы поддерживают именно эту манеру выравнивания.

Также поддерживаются дополнительные базовые типы IDL, еще не вошедшие в стандарт CORBA (64-разрядный целый тип, *signed* и *unsigned*, плавающий тип двойной точности и символы UNICODE).

Процедуры маршалинга типов, определяемых пользователем, реализованы на основе интерпретации TypeCode. TypeInfo из примера, описанного выше, является простейшим аналогом TypeCode.

TypeCode существует для каждого конкретного типа языка IDL, как базового, так и составного, и содержит полную информацию о внутренней структуре типа. Понятие TypeCode формально определяется в спецификации CORBA как некоторый интерфейс с методами доступа к хранимой информации. Любой TypeCode содержит поле со значением типа TCKind и дополнительные параметры, соответствующие этому значению. Перечислимый тип TCKind определяет множество видов TypeCode для базовых типов (например, *tk_long, tk_float*) и вид конструирования составных типов (например, *tk_struct, tk_sequence*).

Для каждого типа данных, определенного пользователем, константы типа TypeCode, как правило, порождаются компилятором по соответствующей IDL спецификации.

На уровне реализации, TypeCode - это класс C++:

```
class TypeCode {
public:
    // вид TypeCode
    TCKind kind(Environment&) const;
    // сравнение двух TypeCode;
    // значения типа TC_ptr - указатели на константы TypeCode
```

```
        Boolean equal(TC_ptr, Environment&) const;
        ...
        enum traverse_status {traverse_stop, traverse_continue};
        traverse_status traverse(
                const void    *value;   // данные
                void          *context; // поток
        ...);
        TypeCode(TCKind, unsigned char *, ...);
private:
        Octet    *_buffer;
        TCKind *_kind;
}
```

Коструктор класса инициализируется значением типа TCKind и буфером с описанием (TypeCode) компонент, для составных типов. Например, для типа *struct Foo* {*long l; string s;*} необходимо указать значение *tk_struct* и буфер следующего вида:

```
const unsigned char tc_Foo[] = {
        0, 0, 0, 1,
        0, 0, 0, 12, 'I', 'D', 'L', ':', 'F', 'o', 'o', ':', '1', '.', '0', 0,
        0, 0, 0, 4, 'F', 'o', 'o', 0,      // идентификатор типа
        0, 0, 0, 2,                        // количество полей
        0, 0, 0, 2, 'l', 0, 0, 0,
        0, 0, 0, 3,                        // TypeCode поля l (tk_long)
        0, 0, 0, 2, 's', 0, 0, 0,
        0, 0, 0, 18,                       // TypeCode поля s (tk_string)
        0, 0, 0, 0
}
```

Метод *traverse()* позволяет рекурсивно обойти все компоненты типа, описанного TypeCode.

4.2. Процедура интерпретации

Понятия спецификации CDR, включающие поток октетов и набор методов для работы с ним, реализованы как класс C++:

```
class CDR {
public:
        // базовые примитивы маршалинга
        CORBA2::Boolean put_short(CORBA2::Short s);
        inline CORBA2::Boolean put_ushort(CORBA2::Ushort s) {
                return put_short((CORBA2::Short) s); }
        ...
        CORBA2::Boolean get_short(CORBA2::Short &s);
        inline CORBA2::Boolean get_ushort(CORBA2::Ushort &s) {
                return get_short((CORBA2::Short) s); }
        ...
        // интерпретатор маршалинга
        static CORBA2::TypeCode::traverse_status encoder(
                        CORBA2::TC_ptr tc,      // TypeCode
                        const void      *data,  // данные
                        const CDR       *context, // поток
                        ...);

        static CORBA2::TypeCode::traverse_status decoder(...);
private:
        unsigned char   *buffer;
}
```

Маршалинг данных осуществляется методами *encoder()* и *decoder()*, в основе которых лежит процедура интерпретации TypeCode.

Параметрами метода *encoder()* являются сами данные, предназначенные для маршалинга, соответствующий TypeCode и указатель на буфер. Алгоритм работы такой же, как и интерпретатора, описанного в введении к статье: если исходные данные являются значением базового типа (это определяется по виду TypeCode), то сразу вызывается соответсвующий базовый примитив, записывающий информацию в буфер с нужным выравниванием. Иначе происходит вызов интерпретатора TypeCode, реализованный как метод TypeCode::*traverse()*. Интерпретатор

рекурсивно кодирует значения полей сложного типа, вычисляя размер и границу выравнивания для данных, описанных TypeCode.

Размер и граница выравнивания определяются с помощью некоторой таблицы, проиндексированной на основе значений типа TCKind. Значения этого перечислимого типа зафиксированы в стандарте CORBA. Поля таблицы содержат либо непосредственно значения размера и границы выравнивания, либо указатель на функцию, вычисляющую эти величины (как правило, для составных типов).

Размер и граница выравнивания для базовых типов зависят от конкретной платформы. Поэтому часть таблицы, относящаяся к базовым типам, порождается во время инициализации брокера объектных запросов.

Метод *decoder( )*, выполняет функции, обратные методу *encoder( )*, и имеет аналогичную структуру. В качестве дополнительных действий, например, *decoder( )* осуществляет необходимое отведение памяти для данных переменной длины (*sequence* или *string*).

4.3. Достоинства и недостатки метода

К достоинствам рассматриваемого метода, как и вообще технологии интерпретации, можно отнести компактность исполняемого кода и унифицированность использования. Интерпретатор TypeCode единообразно работает с данными любого типа, допускаемого IDL, и произвольной сложности. При этом, интерпретатор поддерживает маршалинг данных, типы которых не известны в момент компиляции. TypeCode таких типов могут быть получены во время выполнения, например, по сети.

Также необходимо заметить, что TypeCode - это уже использующаяся в брокере информационная структура, а не введенная специально для реализации маршалинга. TypeCode, в частности, необходим для описания типов фактических параметров при динамическом вызове.

Базовые примитивы маршалинга реализованы как *inline*-функции, что позволяет уменьшить время выполнения. Но увеличение кода за счет *inline*-подстановок может быть нежелательным, например, для машин с небольшим объемом КЭШ-

памяти. В общем случае, возможно разумное ограничение использования *inline*-подстановок с учетом доступной памяти [2].

К недостаткам интерпретатора маршалинга по TypeCode можно отнести то, что TypeCode жестко фиксирует внутреннее представление типа. Если оптимизирующий компилятор поменяет местами поля в структуре, то такое значение не сможет быть правильно проинтерпретировано с помощью TypeCode.

Основной недостаток, присущий всем методам, основанным на интерпретации, - небольшая скорость выполнения. Потеря производительности (за счет рекурсивного механизма разбора TypeCode) особенно видна при работе с данными составных типов глубокой вложенности. Использование уже скомпилированного кода процедур маршалинга вместо интерпретатора TypeCode может частично решить эту проблему.

5. Использование типизированных процедур маршалинга

5.1. Процедуры маршалинга

Не трудно заметить, что маршалинг значений составных типов осуществляется по некоторой схеме, зависящей от конкретного типа. Например, маршалинг данных типа *struct* включает последовательный маршалинг всех полей, в порядке, указанном в IDL спецификации.

Следовательно, для маршалинга сложных данных можно использовать уже скомпилированные процедуры маршалинга. Процесс порождения таких процедур для типов, описанных в IDL спецификации, достаточно регулярен и может быть автоматизирован с помощью IDL компилятора.

Как уже было сказано выше, спецификации на IDL отображаются в язык приложения (C++ в нашем случае). Для этой цели используется IDL компилятор, в задачи которого входит синтаксический разбор IDL спецификации, построение синтаксическго дерева, генерация на основе этого дерева кода на C++. Порождаемые компилятором файлы включают отображение конструкций IDL в соответствующие конструкции C++ и набор вспомогательных процедур и данных, необходимых для общения с C++ ORB. В число последних входят процедуры маршалинга для данных каждого типа из IDL спецификации.

Процедуры маршалинга базовых типов, содержащие информацию о границе выравнивания и размере соответствующих данных, совпадают с методами класса CDR, реализующими базовые примитивы маршалинга.

Процедуры маршалинга составных типов строятся на основе базовых процедур и имеют вложенную структуру. При этом возможен как просто вызов необходимой процедуры маршалинга, так и ее *inline*-подстановка (причем, не только для базовых типов). В нашем случае, *inline*-подстановка применяется только к достаточно компактным процедурам (несколько операторов на C++). К более громоздким процедурам, например, для данных типа *array*, обращение из других процедур маршалинга происходит посредством обычного вызова.

На C++ процедуры маршалинга для типа *struct Foo* {*long l; string s;*} имеют вид:

```
// процедура кодирования
inline CORBA2::Boolean operator <<= (CDR &trg, const Foo &src) {
        return (trg <<= src.l) && (trg <<= src.s); }

// процедура декодирования
inline CORBA2::Boolean operator >>= (const CDR &src, Foo &trg) {
        return (src >>= trg.l) && (src >>= trg.s); }
```

Операторы <<= и >>= перегружаются для каждого типа, заданного пользователем. В примере, реализация оператора <<= для типа *Foo* сначала использует оператор <<=, определенный для типа *long*, а затем -оператор <<=, определенный для типа *string*.

Заметим, что как и процедуры интерпретации *encoder( )* и *decoder( )*, типизированные процедуры маршалинга для кодирования и декодирования данных соответственно имеют симметричные структуры. С точностью до базовых процедур и некоторых специфических действий при декодировании.

5.2. Достоинства и недостатки метода

В разделе 6 приведены конкретные результаты, полученные при использовании технологий компиляции и интерпретации в

маршалинге. Пока же заметим в пользу компиляции, что при работе с данными составных типов производительность существенно улучшается по сравнению с интерпретацией.

Как и следовало ожидать, за улучшение производительности пришлось платить увеличением объема исполняемого кода. Например, IDL компилятор для каждого типа, определенного пользователем, порождает процедуры маршалинга. Для работы с брокером объектных запросов эти процедуры должны быть скомпилированы вместе с другими вспомогательными процедурами и некоторыми библиотеками, содержащими функции самого брокера.

При реализации процедур маршалинга для уменьшения времени выполнения использовалась технология *inline*-подстановок. Как правило, процедуры маршалинга, в том числе и для составных типов, реализованы именно как *inline*-процедуры. *Inline*-подстановки возникают вместо вызовов соответствующих процедур маршалинга. Очевидно, что размер выполняемого кода будет существенно зависеть от количества методов и их параметров в IDL спецификации.

Интерпретатор маршалинга работает с нетипизированными данными, структура которых описана соответствующими TypeCode. При такой жесткой фиксации внутренней структуры данных могут возникать проблемы с оптимизирующими компиляторами. Интерпретатор маршалинга не может корректно работать с данными, элементы которых размещены в памяти в порядке, отличном от предполагаемого TypeCode. Использование типизированных процедур маршалинга позволяет решить эту проблему.

Однако нужно заметить, что скомпилированные процедуры невозможно использовать для маршалинга данных, типы которых не известны во время компиляции. Для маршалинга таких данных все равно необходимо использовать интерпретацию.

6. Сравнение результатов тестирования

Сравнение возможностей методов маршалинга, ориентированных на интерпретацию и компиляцию, соответственно,

было проведено на примере, работающем со следующей IDL спецификацией:

```
struct S {
        long l;
        string s;
};

typedef sequence<S> Seq;

struct SC {
        S s[10];
        any a;
        Seq seq;
        string s1;
};

typedef SC Arr[10][10];

interface Testing {
        void test_long (in long par);
        void test_struct (in S par);
        void test_seq (in Seq par);
        void test_arr (in Arr par);
};
```

Пример состоит в последовательном вызове функций, декларируемых интерфейсом Testing. В качестве типов входных параметров рассматривались типы IDL различной сложности.

В Таблице 1 указаны времена, необходимые для выполнения маршалинга параметров соответствующих типов как на стороне клиента, так и на стороне сервера. Эксперимент проводился на Sun SPARCstation 4 /microSPARC 100Mhz с использованием компилятора gcc 2.7.0.

Полученные результаты демонстрируют, что производительность при передаче значений базовых типов практически не зависит от метода реализации маршалинга. Это объясняется тем, что интерпретатор и скомпилированные

процедуры маршалинга для базовых типов состоят практически только из одних базовых примитивов.

Таблица 1.

Типы данных
 	
Вид маршалинга
	long	struct S {
		long l;
		string s;
		}
			typedef
			sequence<S>
			Seq
				typedef
				struct SC {
				S s[10];
				any a;
				Seq seq;
				string str;
				} Arr[10][10]

Скомпилированные процедуры маршалинга	0.250 мсек.	0.303 мсек.	6.011 мсек.	46.550 мсек.
Интерпретатор маршалинга	0.275 мсек.	0.470 мсек.	15.500 мсек.	199.970 мсек.
Скомпилированные процедуры маршалинга (с оптимизацией компилятора C++)	0.155 мсек.	0.192 мсек.	2.344 мсек.	15.680 мсек.
Интерпретатор маршалинга (с оптимизацией компилятора C++)	0.157 мсек.	0.255 мсек.	6.921 мсек.	92.170 мсек.

Использование скомпилированных процедур маршалинга дает наибольший выигрыш по времени при работе с данными глубокой вложенности. При маршалинге таких данных интерпретация превращается в достаточно длинную последовательность рекурсивных вызовов. Скомпилированные процедуры маршалинга,

с учетом использования *inline*-подстановок, позволяют практически избежать накладных расходов на дополнительные вызовы процедур. Например, при работе с данными типа *array Arr* из примера (см. Таблицу 1) производительность по времени увеличивается примерно в шесть раз, с использованием оптимизирующей опции компилятора C++, и почти в четыре с половиной раза, не используя эту возможность.

Заметим, что для некоторых типов (*sequence Seq* и *array Arr* из примера) использование скомпилированных процедур маршалинга дает даже лучший результат, чем применение стандартной оптимизации при компиляции методов маршалинга, ориентированных на интерпретацию.

Таблица 2.

	Интерпретатор маршалинга	Скомпилированные процедуры маршалинга
Размер объектного кода	38408 B	60232 B

Размер объектного кода увеличивается за счет типизированных процедур маршалинга, и особенно, за счет их *inline*-подстановок, в среднем, на 30%. Для некоторого примера, работающего с реализацией стандартного сервиса именования спецификации CORBA (Naming Service), объем кода при использовании технологии компиляции для маршалинга превышает объем кода, ориентированного на интерпретацию на 45% (см. Таблицу 2). Более гибкое использование *inline*-подстановок, например, только при реализации процедур маршалинга наиболее часто используемых типов, может позволить сократить расход памяти.

7. Заключение

Маршалинг (преобразование) данных необходимо возникает при организации вычислений в распределенной среде. В статье были рассмотрены два принципиально разных метода реализации процедур маршалинга в контексте разработки брокера объектных запросов.

Экспериментальные данные, полученные в результате исследования, подтвердили известную дилемму “время выполнения - размер кода”, присущую сравнению методов, ориентированных на интерпретацию и компиляцию соответственно.

Так как для разработки брокера объектных запросов основным показателем является время выполнения, то в конечном итоге, предпочтение было отдано методу, ориентированному на компиляцию. Хотя этот выбор не окончателен. Как показывают исследования в этой области [2], наилучший результат дает сочетание всех возможных стратегий (интерпретации, компиляции, *inline*-подстановок) на основе некоторых эвристик. Компилятор, реализующий такую “гибридную” стратегию генерации процедур маршалинга, для каждого узла синтаксического дерева решает, какую из возможных альтернатив выбрать. Например, для узла, являющегося определением типа, происходит выбор между методом, ориентированным на интерпретацию, и методом, ориентированным на компиляцию. А для узла, представляющего поле некоторого составного типа, определяется необходимость *inline*-подстановки.

Эвристики, руководящие этим выбором, рекомендуют применять выигрышные по времени (и дорогие по памяти) стратегии только для реализации маршалинга типов данных, наиболее часто передаваемых между приложениями в рамках некоторого взаимодействия.

По принципу “локальности рабочего множества”, большая часть времени выполнения приходится на достаточно небольшую часть программного кода (Кнутт, 1971). Аналогичные исследования в области сетевых взаимодействий показали, что это утверждение верно и для данных, передаваемых между компонентами распределенных систем. Таким образом, оптимизация маршалинга может быть осуществлена только для подмножества типов данных, используемых в приложении, без существенной потери выигрыша во времени.

Исходная IDL спецификация сама по себе уже содержит некоторые указания о возможной частоте использования типов. Например, интуитивно понятно, что тип элемента массива следует считать часто используемым. То есть, частота использования типа может быть вычислена на основе того, сколько и в качестве

элемента каких именно составных типов он используется. Такой статический анализ IDL спецификации может давать достаточно хорошие критерии выбора подмножества типов для оптимизации маршалинга [2]. Этот подход не зависит от конкретного приложения и может быть осуществлен самим IDL компилятором.

Для каждого конкретного приложения более точные оценки для выбора "рабочего множества" IDL спецификации можно получить с помощью средств профилирования программ. Рабочий профиль программы, учитывая особенности выполнения, такие как циклы, ветвления, позволяет определить, переменные каких типов наиболее часто передаются по сети во время прогона. Используя эти статистические данные, IDL компилятор принимает решение о необходимости оптимизации маршалинга для каждого типа из IDL спецификации.

Оптимизируя процедуры маршалинга только для подмножества типов из IDL спецификации, можно добиться необходимого баланса между временем выполнения и размером кода. Алгоритм выбора такого подмножества, реализация соответствующего IDL компилятора - возможные темы для дальнейших исследований.

Литература

1. The Common Object Request Broker: Architecture and Specification. Revision 2.0. July 1995. Object Management Group.

2. Philipp Hoschka. Automating Performance Optimisation by Heuristic Analysis of A Formal Specification. IFIP TC 6/6.1 International Conference on Formal Description Techniques IX (Theory, application and tools), 8-11 October 1996, c. 77-92.

3. Hiroki Horiuchi, Tetsuya Kuroki, Sadao Obana, Kenji Suzuki. EPER: Efficient Packed Encoding Rules for ASN.1. IFIP TC 6/6.1 International Conference on Formal Description Techniques IX (Theory, application and tools), 8-11 October 1996, c. 179-194