ПОСТРОЕНИЕ РАСПРЕДЕЛЕННЫХ СИСТЕМ СРЕДСТВАМИ ФУНКЦИОНАЛЬНЫХ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ

В.Н. Брагилевский

В настоящей статье предлагается общий подход к построению распределенных систем средствами функциональных языков программирования. В основе этого подхода лежит вызов удаленных функций как аналог вызова удаленных процедур (RPC) в императивной парадигме или методов удаленных объектов (RMI) в объектно-ориентированной. Предполагается, что удаленные функции являются чистыми, т. е. выполняются без побочных эффектов. Описываются важнейшие детали технической реализации такого подхода в чисто функциональном языке программирования Haskell.

Введение

Функциональные языки программирования долгое время находились на обочине промышленного программирования, оставаясь по большей части предметом рассмотрения академической общественности. Однако в последнее время наблюдается широкое проникновение элементов функциональных языков в промышленные языки программирования, такие как C++ и C#. С другой стороны, достижения в проектировании самих функциональных языков позволяют использовать их в областях, считавшихся ранее исключительной прерогативой низкоуровневых императивных языков (C, Fortran). К таким областям относятся и высокопроизводительные параллельные вычисления. В качестве примеров таких работ можно привести исследования по параллельному программированию на языке Рефал [1], разработанном В.Ф. Турчиным [2], а также исследования, выполняемые в Московском энергетическом университете [3].

В основе построения распределенных систем лежит сетевое взаимодействие как механизм передачи данных между различными узлами (процессами) системы. Выделяют четыре категории основных примитивов сетевого взаимодействия [4]:

- низкоуровневая передача сообщений (сокеты Беркли, интерфейс MPI);
- вызов удаленных процедур (в т. ч. вызов методов удаленных объектов);
- высокоуровневая передача сообщений (в качестве сообщений здесь выступают структуры данных языка программирования);
- непрерывные потоки данных.

Эти примитивы в разной степени соответствуют стилю функционального программирования, поэтому при реализации распределенных систем требуется выбирать такие примитивы, которые могут быть отражены на требуемый язык программирования без привлечения дополнительных синтаксических и семантических сущностей. Выбор примитива оказывает влияние на целый ряд факторов, среди которых такие важные параметры распределенных систем как эффективность, отказоустойчивость, открытость и масштабируемость получаемых приложений. Не менее важное значение имеет и удобство написания программ, влияющее на скорость их разработки.

В работе [5] была предпринята попытка анализа существующих примитивов сетевого взаимодействия на предмет соответствия функциональной парадигме. В результате анализа оказалось, что наиболее предпочтительным примитивом для реализации распределенных систем средствами функциональных языков является вызов удаленных процедур (функций в терминах избранной парадигмы). В данной статье предлагается обобщенная архитектура распределенной системы, построенная с использованием выбранных примитивов, и обсуждаются основные детали технической реализации этой архитектуры на примере библиотеки, разработанной для языка Haskell [6].

Архитектура распределенной системы на базе вызова удаленных функций

Будем рассматривать клиент-серверную архитектуру распределенных систем, в которых в рамках единой системы действует множество процессов-серверов и процессов-клиентов. Для функционирования такой системы необходимы службы именования с реестрами серверов, службы координации с поддержкой синхронизации времени, службы поддержания отказоустойчивости и защиты информации [4]. Все эти вспомогательные службы составляют инфраструктуру системы, они обычно строятся поверх примитивов сетевого взаимодействия.

Для выполнения удаленного вызова функции клиент должен обладать следующей информацией:

- местонахождение сервера (например, IP-адрес и номер порта в сетях TCP/IP);
- набор функций, реализованных на сервере (интерфейс сервера);
- параметры удаленных функций и возвращаемые ими значения;
- описания типов данных, соответствующих параметрам и возвращаемым значениям удаленных функций.

Ясно, что местонахождение сервера может определяться во время выполнения программы-клиента, например, оно может быть результатом запроса к службе именования. В рамках настоящей работы предполагается, что набор функций и информация об их параметрах определяются на этапе компиляции. Для их задания достаточно предоставить клиенту *интерфейсный файл*, т. е. список функций с указанием типов их параметров и возвращаемых значений. При сетевом взаимодействии неизбежны ошибки передачи данных, связанные с нарушениями в функционировании как компонентов распределенной системы (серверов, клиентов, вспомогательных служб), так и непосредственно сетевой инфраструктуры, поэтому клиентам также необходимо контролировать соответствующие исключительные ситуации. Сервер может не знать о потенциальных клиентах, которые могут обратиться к нему с запросом, и их местоположении, однако уже на этапе компиляции он должен иметь реализации всех функций из интерфейсного файла.

Собственно передача сообщений (с параметрами и результатами) и организация выполнения вызываемых функций является задачей инфраструктуры системы. При этом может иметь место следующая последовательность действий:

1. Клиент выполняет удаленный вызов.
2. Информация о вызываемой функции упаковывается (сериализуется) в одно сообщение вместе со всеми параметрами вызова.
3. Упакованное сообщение передаётся на сервер.
4. Сервер распаковывает (десериализует) сообщение, определяет и запускает вызываемую функцию, передавая ей в качестве параметров извлеченные из полученного по сети сообщения данные.
5. Результат функции упаковывается вместе с необходимой служебной информацией и передается клиенту.
6. Клиент распаковывает результат и выполнение удаленного вызова завершается.

Следует отметить некоторые ограничения предлагаемого механизма. Во-первых, на всех его этапах возможно возникновение исключений, обработка которых может предусматривать как повторное выполнение запроса (в рассматриваемом случае чистых функций, которые по определению являются идемпотентными, это не может привести к ошибочным результатам), так и отказ от выполнения запроса с информированием клиента посредством сгенерированного исключения. Во-вторых, предполагается, что все параметры и результаты имеют конечный размер, что вообще говоря не гарантируется, поскольку такие языки программирования как Haskell позволяют работать с бесконечными структурами данных. Это предположение необходимо для корректной упаковки параметров и результатов в сообщения, передаваемые по сети. В принципе, ограничение предложенного механизма на конечность параметров и результатов может быть снято, если задействовать для моделирования бесконечных структур данных примитив непрерывных потоков данных, однако в настоящей работе этот случай не рассматривается. В-третьих, поскольку сетевое взаимодействие неизбежно приводит к возникновению побочных эффектов, так как фактически оно является одной из форм ввода-вывода, с точки зрения клиента функция, вызываемая удаленно, не может сохранить свойство чистоты, присущее той же функции, рассматриваемой как функции, выполняемой на сервере. Тем не менее, здесь существует возможность гарантировать отсутствие нежелательных побочных эффектов в адресном пространстве процесса-клиента, ограничивая их эффектами сетевого взаимодействия, которые будут наблюдаться клиентом исключительно в виде возбуждения исключений.

Для реализации этого механизма необходимо наличие следующих вспомогательных компонентов на клиенте и сервере:

- набор клиентских функций-заглушек, имеющих те же параметры, что и соответствующие функции из интерфейсного файла;
- клиентская функция, осуществляющая упаковку параметров и выполняющая передачу сообщения на сервер и получение сообщения с результатом; эта же функция должна отвечать за информирование клиента о возникших исключениях сетевого взаимодействия;
- набор функций-реализаций на сервере, удовлетворяющий интерфейсному файлу;
- серверная функция-распределитель, осуществляющая получение сообщений от клиентов, их распаковку и передачу требуемой функции-реализации, получение результата вычислений и отправку соответствующего сообщения клиенту.

Общая схема взаимодействия клиента и сервера представлена на рис. 1.

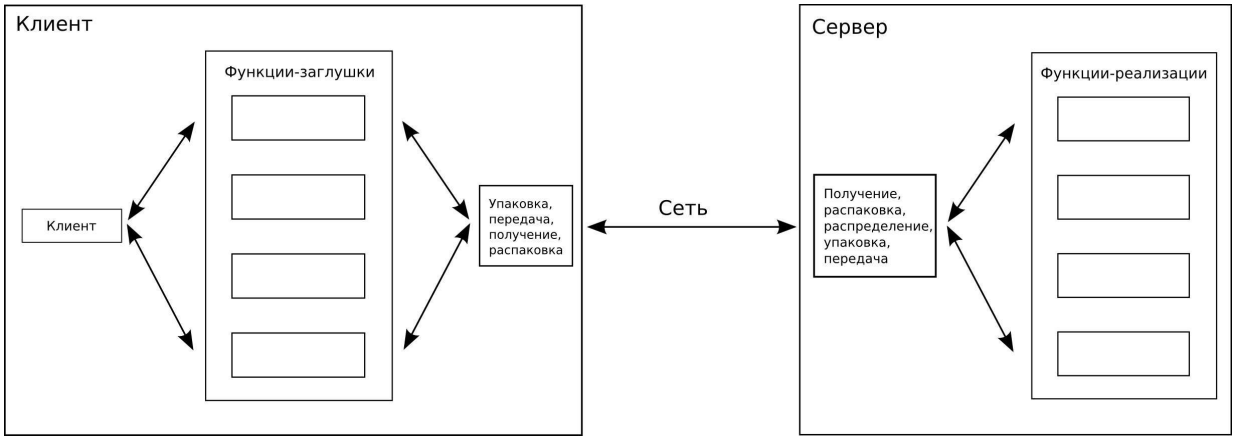

Рис. 1. Схема взаимодействия клиента и сервера

При наличии базового механизма функционального сетевого взаимодействия нетрудно представить реализацию высокоуровневых служб распределенной системы. Так, служба именования может моделироваться парой функций, одна из которых преобразует семантические имена серверов в структуры данных с полной адресной информацией, а вторая выполняет обратное преобразование, позволяя искать по части адресной информации соответствующие семантические имена. Эти функции могут использоваться клиентом при определении местоположения сервера, реализующего необходимые функции. Служба организации отказоустойчивости вообще не требует участия со стороны клиента, её реализация может размещаться во вспомогательных функциях клиента и сервера, реализующих сетевое взаимодействие. Простейший сценарий обеспечения отказоустойчивости — это повторный вызов, в более сложных ситуациях возможен выбор другого действующего в данный момент сервера и перенаправление вызова именно ему. Аналогичным образом реализуется служба защиты информации: для этого достаточно организовать шифрование передаваемых по сети сообщений и их дешифрование при получении. Необходимые при этом атрибуты криптографических протоколов (например, ключи шифрования) могут определяться прозрачным для клиента образом в конфигурационных файлах.

Наиболее интересным моментом реализации высокоуровневых служб распределенных систем является реализация распределенной координации. Дело в том, что соответствующие службы фактически отображаются на функции высших порядков. Например, выбор главного узла представляет собой распределенный алгоритм поиска максимального элемента при заданном предикате — критерии выбора [4]. Параллельное выполнение одного вызова на нескольких серверах для разных данных и сбор полученных результатов — это пара функциональных примитивов *map/reduce*

Детали реализации библиотеки вызовов удаленных функций в языке Haskell

Язык программирования Haskell является чисто функциональным языком программирования. В отличие от большинства других распространенных функциональных языков, таких как OCaml, F#, Lisp, Erlang, которые оставляют программистам возможность использовать при необходимости императивный стиль, в языке Haskell такая возможность строго контролируется средствами монадических вычислений. Язык препятствует смешению частей программ, связанных с вводом-выводом, с чисто функциональными компонентами. В предлагаемой реализации именно монады позволяют ограничить негативный результат побочных эффектов сетевого взаимодействия, ограничив их генерацией соответствующих исключений.

Наличие интерфейсных файлов с описанием функций и их типов делает возможным автоматическое построение всех необходимых клиентских функций-заглушек. Эта задача состоит из двух подзадач. Первая из них заключается в разборе интерфейсного файла средствами библиотеки — парсера исходного кода на языке Haskell, а вторая состоит в непосредственной генерации функций-заглушек на этапе компиляции. Вторая подзадача решается стандартными средствами метапрограммирования на платформе Haskell, а именно пакетом Template Haskell [7].

Задачи сериализации и десериализации (упаковки и распаковки) сообщений также могут быть решены средствами Template Haskell, для этих целей предусматривается автоматическое порождение функций, преобразующих структуры данных, с которыми работают функции, в байтовые строки (ByteString), которые с свою очередь передаются по сети с использованием стандартных функций сетевого взаимодействия.

Рассмотрим простейший пример сетевого взаимодействия на основе вызова удаленных функций. Для простоты будем считать, что сервер работает по тому же адресу, что и клиент (localhost), но использует собственный номер порта (скажем, для определенности, 1500).

Итак, перечень удаленных функций определяется интерфейсным файлом (RFunctions.hs):

```
module RFunctions where

f :: Integer -> Integer
g :: Pair -> String
h :: [Integer]-> (Integer, Integer)
```

Первая удаленная функция (f) вычисляет по одному заданному целому числу другое целое число. Вторая функция (g) принимает на вход пользовательский тип данных (Pair) и возвращает строковое значение. Наконец, третья функция преобразует список целых чисел в пару целых.

Структура данных Pair определяется в модуле описания типов данных:

```
{-# LANGUAGE TemplateHaskell #-}
module DataTypes where
import StubGenerator
import Language.Haskell.TH

data Pair = Pair Integer Integer
```

```
    deriving Show
$(derivingBinary ''Pair)
```

Функция *derivingBinary* здесь — это функция TemplateHaskell, которая генерирует вспомогательные функции для сериализации данных типа Pair на этапе компиляции.

Наконец, сам код клиента, который вызывает все эти функции и возвращает результат вызова:

```
{-# LANGUAGE TemplateHaskell #-}
import ClientUtils
import StubGenerator
import DataTypes

$(genClientStubs "RFunctions.hs")

evalSmth n = do
    n1 <- f n
    n2 <- g (Pair n n1)
    (n3, n4) <- h [1..n]
    return (n1, n2, n3, n4)
```

Файл RFunctions.hs является интерфейсным, на основе его содержимого генерируются клиентские функции-заглушки (функция *genClientStubs*). Функция *evalSmth* представляет собой монадическое вычисление, оно не выполняется само по себе. Для его запуска необходимо указать сервер, на котором реализованы соответствующие функции. В следующей основной программе монадическое вычисление выполняется на некотором сервере, а его результат выводится на консоль:

```
main = eval (evalSmth 13) (ServerAddress "localhost" 1500)
         >>= putStrLn.show
```

Таким образом всё, что необходимо сделать на клиенте, — это описать соответствующее монадическое вычисление (которое в реальности, разумеется, будет гораздо более сложным) и запустить его на выполнение.

На стороне сервера вместо интерфейсного файла может использоваться собственно модуль языка Haskell с реализованными функциями и объявлениями используемых типов данных. Для удобства написания кода сервера все функции этого модуля автоматически (средствами TemplateHaskell) регистрируются на сервере на этапе компиляции. Распределение пришедшего запроса требуемой функции вырождается в поиск в ассоциативном массиве, ставящем в соответствие имени функции её реализацию. Такой массив также генерируется автоматически.

Приведем для иллюстрации код основной программы-сервера, в котором запускается функция-распределитель, а на вход ей подаётся номер порта сервера и «реестр» зарегистрированных на сервере функций-реализаций:

```
main = serve 1500 $(genServerFunctions "RFunctions.hs")
```

Все приведенные фрагменты кода (за исключением кода функций-реализаций) полностью исчерпывают код, который должны писать разработчики распределенной системы на основе предлагаемого подхода.

Сравнение предлагаемого подхода с идеями Glasgow distributed Haskell

Наиболее близко по заявленным целям (построение распределенных систем средствами функциональных языков программирования) к настоящей работе подходит проект Glasgow distributed Haskell [8], выполнявшийся в Эдинбургском университете (Шотландия) до 2003 года. Поэтому необходимо указать на основные отличия между предлагаемым здесь подходом и идеями, заложенными в основу реализации GdH. Проект GdH определяет язык, который является расширением Glasgow Parallel Haskell и Concurrent Haskell, входящих в стандартную поставку самого широко распространенного компилятора для языка Haskell — Glasgow Haskell Compiler (GHC) [9]. В рамках этого расширения вводится понятие удаленного узла, а понятие потока исполнения обобщается до удаленного процесса. Кроме того, обеспечиваются базовые средства синхронизации (непереносимые переменные, размещаемые в адресном пространстве одного из процессов системы). Таким образом, пользователю предоставляется возможность запуска потока исполнения на удаленной машине, поэтому с архитектурной точки зрения речь идет о об особом виде многопоточности, при котором потоки распределяются по нескольким независимым машинам.

В отличие от этого, основная задача настоящей работы состоит в построении такой архитектуры приложения, которая позволила бы выполнять удаленные вычисления как можно более прозрачно, без отвлечения на детали низкоуровневого распределения задач по процессам.

Другим отличием (технического характера) является тот факт, что реализация GdH предусматривает внесение исправлений в исходный код самого компилятора GHC, точнее его подсистемы времени исполнения. Представляется, что именно по этой причине проект GdH был остановлен и его реализации не работают в современных версиях компилятора GHC. Предлагаемый в данной работе подход реализуется на уровне библиотеки, которая может быть использована со любыми достаточно актуальными версиями компилятора GHC.

Тем не менее, следует заметить, что сама идея разработки чисто функционального построения распределенных систем на базе языка Haskell впервые получила свое развитие именно в рамках проекта GdH во второй половине 90-х годов XX века.

Дальнейшие направления работы

Среди проблем, требующих своего решения в рамках построения распределенных систем, стоит отметить реализацию «ленивой» передачи параметров и «переноса кода».

Реализация ленивой передачи параметров предполагает отказ от явной упаковки параметров в сообщение с запросом и возможность запроса недостающих данных со стороны сервера. Такая возможность позволила бы в некоторых случаях повысить эффективность взаимодействия, поскольку есть ненулевая вероятность того, что данные, передаваемые на сервер, вовсе не понадобятся. В случае ленивой передачи параметров в такой ситуации параметры по сети передаваться не будут. Одним из возможных решений может являться использование непрерывных потоков данных, как еще одного примитива сетевого взаимодействия.

Перенос кода означает наличие возможности передачи программного кода, который требуется исполнить на удаленной машине. Разумеется, эта возможность привносит дополнительные сложности, связанные с безопасностью приложений, однако вместе с тем придает им большую гибкость. В самом простом случае код может передаваться в виде текста программы, которая на сервере сначала компилируется, а затем исполняется. В более сложных реализациях можно передавать частично скомпилированный промежуточный код. Ясно при этом, что нет большой пользы в передаче непосредственно исполняемого кода, поскольку это возможно только в случае использования одной операционной системы на клиенте и сервере, что в случае распределенных систем редко имеет место.

ЛИТЕРАТУРА:

1. Ю.А. Климов, А.Ю. Орлов. Параллельное программирование на языке Рефал // Труды Всероссийской научной конференции «Научный сервис в сети Интернет: решение больших задач» (22-27 сентября 2008 г., г. Новороссийск). М.: Изд-во МГУ, 2008. С. 241–243.
2. В.Ф. Турчин. Метаязык для формального описания алгоритмических языков // В сб.: Цифровая вычислительная техника и программирование. М.: Сов. Радио, 1966. С. 116–124.
3. С.Е. Бажанов, В.П. Кутепов, Д.А. Шестаков. Язык функционального параллельного программирования и его реализация на кластерных системах // Программирование. 2005. № 5. С. 18–51.
4. Э. Таненбаум, М. ван Стеен. Распределенные системы. Принципы и парадигмы. СПб.: Питер, 2003. 877 с.
5. В.Н. Брагилевский. Примитивы распределённого взаимодействия в языке Haskell: сравнение и анализ // Материалы X международной научно-практической конференции «Информатика: проблемы, методологии, технологии». Воронеж. 2010. С. 125–129.
6. S.P. Jones, J. Hughes. Haskell 98: A Non-strict, Purely Functional Language. 1999.
7. T. Sheard, S.P. Jones. Template metaprogramming for Haskell. Proc. Haskell Workshop. Pittsburgh, 2002. P. 1–16.
8. R.F. Pointon, P.W. Trinder, H-W. Loidl. The Design and Implementation of Glasgow distributed Haskell // IFL 1999. Springer: LNCS 2011, 2000. P. 101–116.
9. Glasgow Haskell Compiler, http://www.haskell.org/ghc/.