ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ РФ
ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
УРАЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
им. А. М. Горького

МАТЕМАТИКО–МЕХАНИЧЕСКИЙ ФАКУЛЬТЕТ
Кафедра алгебры и дискретной математики

ОБ ОБЪЕДИНЕНИИ МОДУЛЕЙ НА ЯЗЫКАХ C++, DELPHI, C#, JAVA В ЕДИНУЮ ПРОГРАММУ

Допустить к защите:
зав. кафедрой Алгебры и
дискретной математики
д.ф.–м.н., проф.
Волков М. В.

Дипломная работа
студента 6 курса
Полетаева Дмитрия Геннадьевича

Научный руководитель
к.ф.–м.н., ассистент кафедры
Алгебры и дискретной математики
Клепинин Александр Владимирович

Екатеринбург
2008

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

К настоящему моменту в области компьютерных наук создано огромное количество языков программирования[1]. Хотя большая их часть представляет лишь исторический интерес и не нашла обширного практического применения, на данный момент не существует языка, который для всех проектов подходит лучше остальных. Более того, иногда оказывается удобным или даже необходимым использование в одном проекте различных языков программирования, например, по следующим причинам:

- Различные модули проекта в силу их специфики удобнее создать на том или ином языке программирования.
- В проекте участвуют люди с различным знанием языков программирования, и мы хотим предоставить им возможность создавать модули на наиболее подходящем им языке.

В табл. 1 перечислены некоторые особенности языков, участвующих в рассмотрении.

Таблица 1. Сравнение языков программирования

Java	➢ Переносимость бинарного кода между различными аппаратными платформами. ➢ Обширная библиотека примитивов (JavaAPI). ➢ Для некоторых закрытых платформ разработка на других языках невозможна.
C++	➢ Позволяет создавать высокопроизводительные приложения, являясь в то же время объектно–ориентированным. ➢ Наличие компиляторов под множество платформ. ➢ Удобен для создания приложений системного уровня.

[1] Список из около 2500 компьютерных языков в [1]

Продолжение табл.1

C#	➢ Основной язык для платформы .Net. ➢ Возможность использования как .Net Framework API, так и прямых вызовов Windows API.
Delphi	➢ Наличие конструктора GUI и коллекций готовых компонент позволяет значительно ускорить разработку некоторого класса программ.

Общей задачей, в рамках которой проходило выполнение данной работы, было создание платформы, позволяющей упростить объединение модулей, созданных на разных языках программирования. Одним из основных свойств, которые должна предоставить такая платформа, является возможность запуска методов (функций, процедур) написанных на одном языке из программы, написанной на другом.

Хотя возможности применения создаваемой платформы обширны, основным назначением, по крайней мере, на начальном этапе является упрощение проведения турнира программ. Турнир программ – это мероприятие, на котором разработчикам предлагается написать некоторую модель поведения в игре, задаваемой создателями турнира. После этого программы–игроки начинают сражаться друг с другом в этой игре. Побеждает тот разработчик, чья программа окажется более успешной. Как правило, разработчики, участвующие в турнире программ, знакомы с разными языками программирования и задача объединения модулей написанных на разных языках появляется из необходимости проведения раундов, в которых друг с другом соревнуются разные решения.

При проведении данной работы сделано исследование существующих способов объединения модулей, принято участие в проектировании собственной системы вызовов удалённых методов, разработана архитектура диспетчера, реализован диспетчер для языка C++.

ОБ ОБЪЕДИНЕНИИ ПРОГРАММНЫХ МОДУЛЕЙ

Объединение модулей в единую систему подразумевает возможность их взаимодействия, реализуемого, как правило, за счёт вызова из одного модуля кода расположенного в другом модуле. Рассмотрим несколько способов объединения модулей написанных на разных языках.

1. **Статическая линковка**
   - На этапе компиляции модули, написанные на разных языках, раздельно компилируются соответствующими компиляторами в объектные файлы.
   - На этапе линковки объектные файлы, полученные на предыдущем шаге, объединяются в единый исполняемый файл.
   - На этапе исполнения никакой дополнительной работы по объединению проводить не нужно.

Преимуществом этого способа является высокая скорость работы на этапе выполнения (т.к. отсутствуют накладные расходы), например, этот способ является обычным для сборки проектов состоящих из нескольких исходных файлов, написанных на языках программирования C++ или Delphi.

Отметим и недостатки:

- Ограниченные возможности по применению, вызванные различиями в реализации языков:
  - Компиляторы языков Delphi и C++ как правило[2] генерируют native код, а Java и C# – код для виртуальной машины (Java машины и .Net Framework соответственно).

[2] Компилятор C++ из Visual Studio может генерировать код для платформы .Net Framework

- ❖ Различные стили вызовов функций (например, порядок загрузки аргументов в стек в C++ и Delphi), хотя, как правило, их удаётся согласовать за счёт специальных директив компилятора.
- ❖ Из–за того, что стандарт языка может не фиксировать способ реализации тех или иных механизмов языка, возможно получить ситуацию, когда модули написанные даже на одном языке, но скомпилированные разными компиляторами, окажутся несовместимы при линковке. Например, в языке C++ не зафиксирован[3] способ реализации таблицы виртуальных методов.

- ➢ Невозможность множественной реализации функции, т.е. создать в разных модулях функции с одинаковым именем и сигнатурой.
- ➢ При замене одного модуля придётся осуществлять линковку всего проекта целиков.
- ➢ Невозможность запустить модули на различных вычислительных узлах (распределить вычислительную нагрузку).

2. Динамические библиотеки

Отличается от предыдущего способа тем, что модули компилируются в динамические библиотеки и вместо линковки используется динамическое подключение библиотек. Это снимает часть перечисленных недостатков статической линковки, но оставляет некоторые проблемы по совместимости. Основное преимущество данного метода заключается в возможности использования одной библиотеки из нескольких программ, что способствует уменьшению размеров программ. Этот способ объединения активно используется при вызове системных методов

[3] Существует Intel C++ ABI (Application Binary Interface) фиксирующий некоторые особенности реализации. Его поддерживается Intel C++ Compiler и GCC (начиная с версии 3.2), что обеспечивает их совместимость при линковке.

операционной системы. К сожалению, этот метод не позволяет напрямую распределять модули по вычислительным узлам в сети.

3. Передача параметров при запуске

В этом случае модули, подлежащие объединению, оформляются в отдельные приложения, а необходимые для работы параметры передаются при запуске. Этот способ оказывается удобен простотой использования и применяется, например, в технологии CGI. В качестве недостатков можно отметить:

- Отсутствие памяти сохраняемой между вызовами. Например, в ситуации, когда приложению необходимо выполнять работу над большим объёмом данных, который при последующих вызовах меняется лишь незначительно, необходимо при каждом запуске передавать все данные целиком. Наличие же сохраняемой между вызовами памяти позволяет передавать лишь изменения с момента последнего запуска, что может оказаться эффективнее.
- Затруднена обратная связь между модулями.
- Запуск процесса сопровождается накладными расходами со стороны операционной системы, иногда, например, в случае семейства операционных систем Windows эти расходы весьма значительны. Для запуска часто используемых модулей этот метод может оказаться непригодным к использованию, т.к. большая часть процессорного времени будет уходить на обеспечение запуска процессов.
- Так же как и в других способах объединения, здесь приходится решать задачу согласования типов данных и способов их хранения. Как правило, при запуске нового процесса параметры передаются в текстовом виде, что хотя и обеспечивает универсальность, но вводит

дополнительные накладные расходы на преобразования в/из текстового вида.

4. Удалённый вызов методов

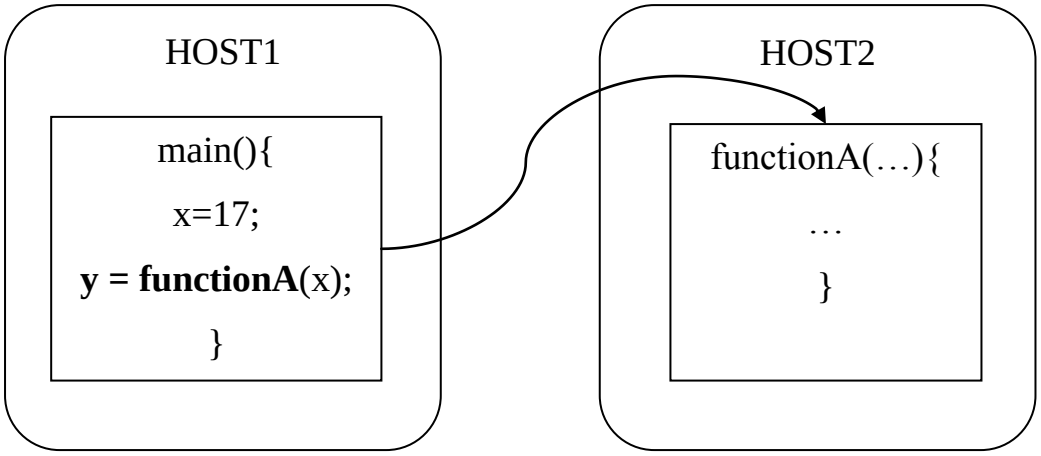
Хотя этот способ имеет бóльшие накладные расходы на вызов, чем использование динамических библиотек, он позволяет объединять модули вне зависимости от того, запускаются ли они на одном компьютере или на разных. В языках Java и C# задача удалённого вызова методов частично решена их создателями (а именно технологии RMI и .NET Remoting соответственно), но, к сожалению, эти решения ограничены платформой, для которой они разрабатывались.

На основании проведённого сравнения было принято решение разработать свою систему объединения модулей с использованием удалённого вызова методов, но способную объединить модули, написанные на языках Java, Delphi, C++, C#.

УДАЛЁННЫЙ ВЫЗОВ МЕТОДОВ

Рассмотрим задачу удалённого вызова методов. Пусть, для примера, на компьютере HOST2 находится метод functionA, который необходимо вызвать с компьютера HOST1.

Рис. 1. Пример удалённого вызова метода



При удалённом вызове возникают следующие задачи:

- Так как соединение между компьютерами позволяет передавать лишь поток байтов, то вызов функции и её аргументы необходимо преобразовать в массив байт.
- Для имитации вызова локального метода необходимо заблокировать поток до момента получения результата от удалённой стороны.
- Для того чтобы вызывать удалённый метод с тем же синтаксисом, что и локальный, должен существовать локальный метод–заглушка, осуществляющий передачу запроса удалённой стороне.

Каждая их этих задач может быть решена отдельной подсистемой. Перечислим компоненты, необходимые для осуществления удалённого вызова метода:

1. Маршалинг

Языки, участвующие в рассмотрении, отличаются набором поддерживаемых типов данных и способом хранения данных для

некоторых типов. Например, в Java отсутствуют примитивные типы беззнаковых целых чисел, которые есть в C++, кроме того различаются способы хранения строк в языках C++, Delphi, Java. Существуют и более тонкие отличия языков, а именно:

- Различные способы управления памятью. Языки Java и C# используют автоматическое управление памятью в отличие от C++ и Delphi.
- В языке Java существуют контролируемые исключения, требующие своего объявления в сигнатуре методов. В языке C++ подобных требований на сигнатуры не накладывается.

Для того чтобы организовать взаимодействие различных модулей необходимо согласовать типы данных, используемых на вызывающей и вызываемой сторонах. При этом могут использовать следующие приёмы, или некоторая их комбинация:

- Ограничить набор допустимых типов, т.е. выделить некоторый набор типов, являющийся подмножеством множества типов, допустимых для каждого из рассматриваемых языков.
- Ввести иерархию своих собственных типов, и реализовать поддержку этих типов для каждого языка.
- Осуществлять преобразование в наиболее подходящий в данном языке тип. Например, строковые объекты из Java (String) преобразовывать в строки с нулевым окончанием в C++ (char*)

Подсистема маршалинга должна обеспечить упаковку фактических параметров метода в некоторый независимый от языка контейнер и, напротив, система демаршалинга, получив на вход этот контейнер, должна по нему воссоздать передаваемые данные и преобразовать их к подходящим типам.

2. Диспетчер

В данной задаче возникает две проблемы диспетчеризации:

- Диспетчеризация пакетов. В конечном счете, вызов удалённой функции влечёт передачу на удалённый хост информации идентифицирующей метод, а также аргументы метода. Если выполняемое приложение многопотоковое, то возникает задача синхронизации множественных вызовов при обращении к общему транспорту.
- Диспетчеризация потоков. При вызове локальных функций, в тот момент, когда управление возвращается к вызвавшему функцию, значение функции уже вычислено и доступно. Если мы хотим повторить то же поведение для удалённых методов, необходимо заблокировать вызвавший поток до тех пор, пока с удалённого хоста не будет получен результат вычислений.

Эти проблемы можно решить единым способом для всего приложения целиком, что позволяет выделить код, отвечающий за диспетчеризацию, в отдельную подсистему и при необходимости обращаться к ней.

3. Генератор прокси и сервисов.

Для запуска удаленного метода необходимо сначала с помощью маршалинга упаковать фактические параметры запуска, а затем передать упакованные данные в диспетчер. Писать соответствующий код каждый раз, когда необходимо вызвать метод, довольно накладно. Для упрощения использования удалённых методов вводятся специальные методы – прокси. Являясь локальными методами прокси позволяют запускать удалённые методы, используя тот же синтаксис, что и при запуске локальных. Так как для каждой пары удалённый метод – язык необходимо создать отдельный прокси метод, их количество может оказаться довольно большим и написание каждого прокси вручную отнимет много времени у

разработчика. В этом случае может оказаться более эффективным подход, когда от разработчика потребуется формально описать сигнатуру метода на некотором псевдо языке, а конкретные прокси для каждого языка будут создаваться по этому описанию автоматически. Также генератор прокси позволит разработчику не задумываться о вызове маршалинга и диспетчера, вызов удалённого метода будет идентичен вызову локального.

Сервис – это метод, который вызывается диспетчером при получении запроса на выполнение метода, находящегося на данном хосте. Он должен обеспечить преобразования, обратные тем, что выполнил прокси, а именно осуществить демаршалинг и вызвать метод, опубликованный для удалённого выполнения.

Хотя для языка Java имеются механизмы для генерации кода на этапе выполнения, для языков C++ и Delphi это было бы весьма затруднительно. Поэтому предполагается, что генератор кода до этапа компиляции проекта создаст необходимые исходные файлы, реализующие требуемую функциональность, а затем исходный код разработчика, объединённый с результатом работы генератора, будет передан компилятору.

Перечислим некоторые этапы создания прокси и сервис методов при использовании генератора кода:

- Разработчик описывает пользовательские типы данных. Это описание необходимо для последующего корректного осуществления маршалинга/демаршалинга.
- Разработчик создаёт описание методов, подлежащих удалённому вызову в некотором конфигурационном файле. Необходимой для генерации информацией является количество аргументов метода, их типы и порядок следования, а также тип возвращаемого значения. Используемые типы данных должны либо поддерживаться маршалингом непосредственно, либо определяться в файле описаний типов данных.

- Генератор кода создаёт для каждого целевого языка:
  - Определения пользовательских типов данных.
  - Код способный осуществить маршалинг/демаршалинг всех используемых типов данных.
  - Прокси и сервисы удалённых методов, использующие сгенерированные функции маршалинга/демаршалинга.

4. Канал передачи

На канал передачи накладываются довольно слабые требования, а именно обеспечить передачу данных между процессами. Для этого подходят, например, именованные каналы, потоки ввода–вывода, сокеты. Нами использованы сокеты как наиболее естественный способ передачи данных по сети.

Рис 2. Схема вызова удаленного метода

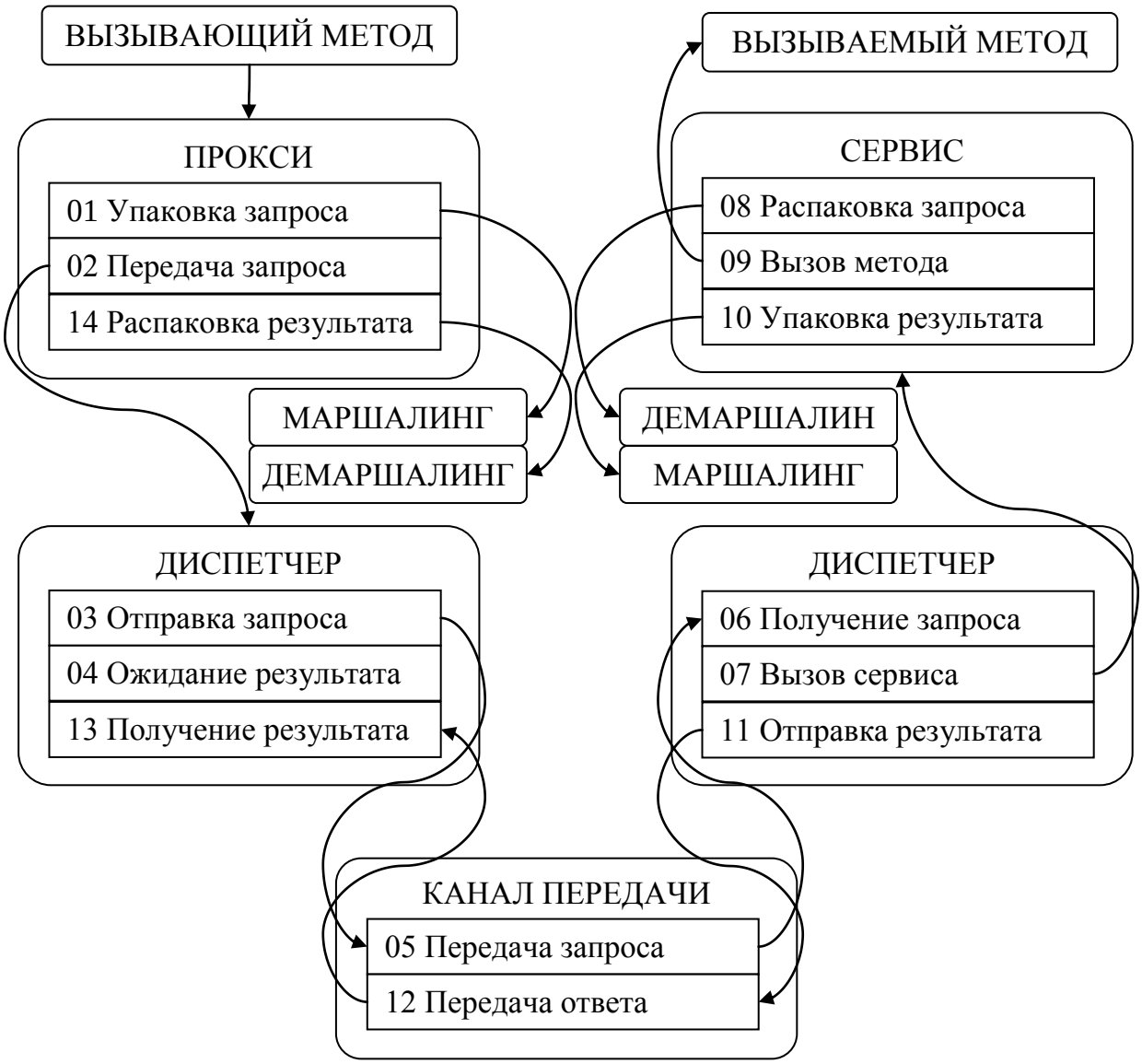


Диаграмма взаимодействия основных компонент показана стрелками, порядок вызовов – цифрами.

ПОИСК УДАЛЁННЫХ МЕТОДОВ

В предыдущем разделе был рассмотрен механизм вызова удалённых методов при уже установленном соединении. Для создания соединения необходимо знать идентификатор удалённой стороны и идентификатор метода. Если количество объединяемых модулей и удалённых методов крайне мало, то можно в каждом модуле локально хранить информацию о всех доступных методах и хостах их поддерживающих. При увеличении количества методов и модулей или при частом изменении конфигурации поддержка такой информации в актуальном состоянии становится весьма накладной. Одним из возможных решений является введение некоторого координирующего сервера, поддерживающего информацию о доступных методах. Это позволяет получить следующие преимущества:

- Исчезает проблема рассогласования информации о доступных методах в разных модулях.
- Часть информации (например, о сетевых адресах) можно получать автоматически в момент подключения модуля к координирующему серверу. Таким образом, модуль должен до момента запуска обладать лишь информацией об адресе сервера, а адреса остальных модулей получить с сервера.
- Информация о доступных узлах и методах может обновляться динамически. Т.е. модуль может просмотреть ресурсы, доступные в данный момент времени.

Примером набора исполняемых функций для такого сервера могут служить:

1. Публикация/отзыв интерфейса[4].
2. Просмотр списка существующих интерфейсов.

[4] Термин интерфейс используется в типичном для ООП значении, а именно как контейнер определений методов.

3. Публикация/отзыв реализации для определённого интерфейса.
4. Просмотр списка существующих реализаций для определённого интерфейса.
5. Определение сведений необходимых для подключения к определённой реализации.

Эту модель можно немного изменить, добавив в сервер функциональность по маршрутизации всех передаваемых пакетов. В таком случае сетевое взаимодействие каждого модуля ограничивается подключением к серверу. Вместо отправки модулю информации об адресе удалённой машины серверу достаточно сообщить модулю имя виртуального канала, связывающего прокси и сервис. Такое изменение вносит следующие отличия:

- Упрощение диспетчеров за счёт того, что нужно устанавливать и поддерживать лишь одно сетевое соединение.
- Т.к. все передаваемые пакеты передаются через один сервер, он может оказаться узким местом и ограничить производительность.

Одним из требований к проектируемой системе было обеспечить наряду с общедоступными методами, методы, доступ к которым возможен только после осуществления некоторых действий (например, авторизации). Возможные способы осуществления данной функциональности:

1. Разрешить вызов всех методов, но внутри закрытых методов проверять разрешения на вызов. Если у вызывающего имеются необходимые разрешения, то выполнять метод, иначе уведомлять вызывающего о недопустимости вызова.
2. При публикации метода указывать дескриптор защиты, описывающий видимость метода для определённых клиентов. Сервер при опросе клиентом доступных методов формирует список на основе этих дескрипторов.

3. Введение дополнительного уровня. В этом случае диспетчер модуля регистрирует на сервере не интерфейс, а контейнер интерфейсов или, иначе говоря, некоторый виртуальный канал внутри которого диспетчер уже сам будет ответственен за работу с интерфейсами.

Третий способ был выбран из–за того, что его использование, во–первых, в меньшей степени меняет логику работы интеграционного сервера, а во–вторых, позволит не усложнять логику работы в интерфейсах, не нуждающихся в скрытых методах. Например, в интерфейсе интеграционного сервера, который не нуждается в скрытых методах, можно выделить специальные виртуальные каналы для каждой функции, предоставляемой интеграционным сервером, что позволяет избавиться от дополнительного пакетирования.

Перечислим объекты, с которыми происходит работа интеграционного сервера и диспетчеров:

Имя конечной точки – определяет соглашение о взаимодействии удалённых сторон (одна из которых будет выступать в роли сервера, а другая в роли клиента). Если в простом случае это соглашение может фиксировать один определённый интерфейс в терминах ООП (т.е. позволять вызывать некоторые методы, фиксированные в данном интерфейсе), то в более общем случае оно может определять контейнер интерфейсов, некоторые из которых доступны сразу после соединения, а некоторые интерфейсы появляются при выполнении определённых условий. Поскольку мы предполагаем создание всех необходимых прокси и сервисов из специальных файлов описаний до этапа запуска, то в качестве имени конечной точки можно использовать короткий

идентификатор, известный всем участвующим в объединении диспетчерам до этапа выполнения[5].

Конечная точка – реализация серверной (предоставляющего услугу вызова удалённых методов) части, соответствующей соглашению, определяемому некоторым *именем конечной точки*. Для одного имени могут быть несколько реализаций (как разными модулями, так и множественная реализация одним модулем). Между именем конечной точки и конечными точками существует отношение "один ко многим", т.е. одному имени могут соответствовать несколько конечных точек, а конечной точке соответствует в точности одно имя.

Сессия – соединение (некоторый виртуальный канал) между клиентом и сервером (а именно конечной точкой, определяющей реализацию сервера). Между конечными точками и сессиями существует отношение "один ко многим" (пример отношений в табл. 2).

Таблица 2. Пример возможного соотношения имен конечных точек, самих конечных точек и сессий.

Имя конечной точки 1			Имя конечной точки 2	
Конечная точка 1	Конечная точка 2		Конечная точка 3	
Сессия 1	Сессия 2	Сессия 3	Сессия 4	Сессия 5

Цель этих объектов следующая:

- Имя конечной точки фиксирует правила взаимодействия.
- Конечная точка в дополнение фиксирует серверную часть для определённых правил.
- Сессия в дополнение фиксирует клиентскую часть для определённого сервера и правил.

[5] В веб–сервисах для описания интерфейсов используется специальный, основанный на XML язык WSDL, что позволяет осуществлять поиск подходящих сервисов и формирование аргументов на этапе выполнения программы.

Действия серверной части для публикации:

1. Опубликовать на интеграционном сервере имя конечной точки.
2. Опубликовать на интеграционном сервере конечную точку для опубликованного имени.
3. Ожидать создания сессии.

Действия клиентской части для подключения к серверу:

1. Просмотреть список имён конечных точек, выбрать нужное имя.
2. Просмотреть список конечных точек для выбранного имени, выбрать нужную конечную точку.
3. Открыть сессию с выбранной конечной точкой.

МЕТАДАННЫЕ

При удалённом вызове методов оказывается недостаточным передать лишь данные запроса (упакованные аргументы вызова), передаваемые пакеты данных необходимо снабжать некоторой дополнительной информацией. Например, такой информацией является идентификатор вызываемого сервиса, без которого невозможно определить, какому сервису необходимо передать полученный пакет. Рассмотрим метаданные, вводимые в пакет, а также функциональность, которую они обеспечивают:

- *sessionID* – идентификатор сессии. Это поле определяет пару: конечная точка – клиент. Для диспетчера это поле позволяет разделять пакеты между различными обработчиками (обработчик сервера регистрируется при объявлении конечной точки, а обработчик клиента – при установлении сессии), которыми в простейшем случае являются сервис и прокси, в этом смысле идентификатор сессии схож с номером порта в TCP. Для интеграционного сервера, т.к. сессия связывает два удалённых узла, это поле определяет удалённую сторону, которой необходимо передать пакет, в этом случае идентификатор сессии выступает как имя виртуального канала, определяющего маршрут следования пакета. Идентификаторы сессии назначаются интеграционным сервером при открытии сессии.
- *size* – размер пакета. В настоящий момент в качестве транспорта для передачи запросов используется потоко–ориентированный TCP транспорт, т.е. данные передаются в едином потоке без деления на пакеты. Поле размера позволяет разбить входной поток на пакеты.
- *requestID* – идентификатор запроса. Рассмотрим следующую ситуацию: многопоточный клиент в двух потоках вызывает один и тот же удалённый метод, но, возможно, с разными параметрами. В этом случае оба запроса будут отправлены удалённой стороне, а

потоки заблокированы до получения результата. Через некоторое время от удалённой стороны будут получены два ответа, и встанет задача распределения этих ответов между двумя ожидающими потоками. Этой ситуации можно избежать, если запретить параллельный вызов удалённого метода, но это ограничение может впоследствии стать довольно неудобным. Идентификатор запроса позволяет корректно сопоставлять получаемые результаты и заблокированные потоки. При отправке запроса диспетчер маркирует пакет незанятым в данный момент идентификатором и отправляет удалённой стороне. Удалённая сторона при формировании пакета содержащего результат сохранит идентификатор, что позволит диспетчеру найти именно тот поток, который послал данный запрос.

- ⇨ *count* – счётчик пакетов, зарезервирован в пакете, но в настоящее время не используется. Предположим, во время работы программы происходит кратковременный разрыв сетевого соединения (либо мы самостоятельно закрыли сокет, диагностировав ошибку при передаче), в этом случае разумным поведением является попытка восстановления соединения. К сожалению, мы не можем узнать, какие пакеты отправились успешно, а какие нет[6]. Одним из решений данной проблемы может являться отправка уведомлений об успешно принятых пакетах. На отправляющей стороне тогда достаточно после восстановления соединения послать заново все пакеты, на которые не было получено подтверждений. На принимающей стороне в этом случае можно столкнуться с получением одного пакета дважды (если пакет был отправлен успешно, а потерялось его

[6] Если системная функция отправки рапортует об успешном выполнении, это лишь говорит об успешном помещении пакета в буфер отправки, но не об успешной доставке.

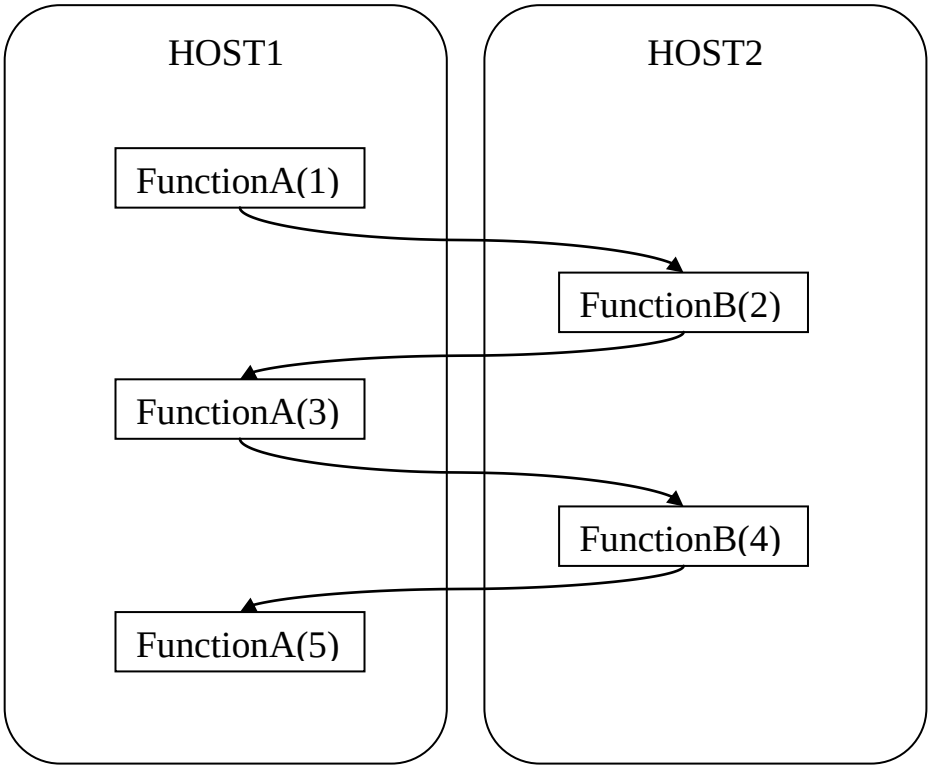
уведомление), но эта проблема решается за счёт учёта полученных пакетов и отбрасывания пакета, если пакет с таким номером был недавно получен. Поле count предназначено для нумерации пакетов, что позволит вести учёт удачно отправленных и принятых пакетов и тем самым восстанавливать соединение после сбоев.

- ⇨ *flags* – поле флагов. На данный момент определены следующие флаги:
  - ➢ *FLAG_DIRECTION* – определяет, является ли данный пакет запросом (и должен быть доставлен сервису) или ответом (и должен быть доставлен потоку, заблокированному в ожидании этого ответа).
  - ➢ *FLAG_ACKNOWLEDGEMENT* – определяет, является ли пакет информационным или пакетом подтверждением (о назначении таких пакетов изложено в описании поля count), зарезервирован в пакете, но в настоящее время не используется.
- ⇨ *CRC* – контрольная сумма заголовка пакета. Используемый нами транспорт TCP обеспечивает надёжность доставки за счёт проверки контрольной суммы. Размер контрольной суммы TCP составляет лишь 16 бит, что при передаче значительных объёмов данных может оказаться недостаточным[7]. В случае повреждения поля длины пакета в большую сторону, диспетчер будет пытаться дочитать пакет до конца, что может вызвать "зависание" диспетчера. Эта ошибка неприятна тем, что вызывает остановку всего сервера. Поэтому было принято решение дополнительно защитить заголовок пакета контрольной суммой. В случае если ошибка находится в теле пакета, а заголовок верен, поддерживается корректное разбиение на пакеты, и ошибка повреждает лишь один пакет, а не всё последующие.

[7] При тестировании на гигабитном соединении по TCP за менее чем сутки было получено несколько ошибок передачи.

⇨ *threadID* – идентификатор потока, зарезервирован в пакете, но в настоящее время не используется. Рассмотрим ситуацию неявной рекурсии. При рекурсии из локальных методов используется один поток, а информация о функциях, не завершивших своё выполнение, помещается в стек. В случае использования удалённых методов при вызове для обработки каждого запроса выделяется поток, таким образом, глубина рекурсии не может быть больше, чем число запущенных потоков. Такое ограничение делает невозможным создание программ с глубоким уровнем рекурсии на удалённых методах.

Рис. 3. Пример неявной рекурсии



Возможным решением задачи об уменьшении количества потоков является использование заблокированных в данный момент времени потоков. Можно назвать две стратегии использования заблокированных потоков:

- До тех пор, пока количество потоков не превысит некоторого порога, выделять потоки для сервисов обычным способом (создавать новые или брать из пула), при превышении порога – использовать случайный поток из заблокированных потоков. Недостатки данного метода:
  - Если приходит пакет для потока, который в данный момент выполняет какой–то сервис, то передать пакет станет возможным лишь после завершения работы сервиса, т.е. в вызов удалённого метода вносятся дополнительные издержки.
  - Возможно возникновение взаимных блокировок. Этой ситуации можно избежать, если запретить вызов удалённых методов из сервисов, но это ограничение является довольно сильным.
- В примере, иллюстрирующем неявную рекурсию, можно заметить, что ответ на FunctionB(2) не будет получен до тех пор, пока не будет получен ответ на FunctionA(3). Это означает, что вызывать функцию FunctionA(3) безопасно из потока, заблокированного во время выполнения FunctionA(1), т.к. во время выполнения FunctionA(3) невозможно разблокирование FunctionA(1). Это замечание можно обобщить: сервис безопасно вызывать в любом заблокированном потоке, который не может быть разблокирован ранее, чем закончится выполнение этого сервиса. Одним из способов отслеживания такого соответствия между запросами к сервисам и заблокированными потоками является передача в пакете информации идентифицирующей поток, в котором происходит вызов удалённого метода. В приведённом выше примере получая запрос на выполнение FunctionA(3) можно по идентификатору потока вычислений определить, что вызов FunctionA(3) происходит в рамках вычисления FunctionA(1) и можно использовать поток, заблокированный на вычислении FunctionA(1). Поле threadID

предназначено для передачи идентификатора вычислений, на основе которого можно на каждом узле ограничить количество необходимых потоков числом потоков, созданных разработчиков во всём проекте. Т.е. в худшем случае один поток созданный разработчиком потребует создания на каждом удалённом узле по одному дополнительному потоку. В приведённом примере неявной рекурсии будет достаточно двух потоков, вне зависимости от глубины рекурсии.

Окончательный вид пакета представлен в табл. 3.

Таблица 3. Строение пакета

Смещение	Размер	Название поля
0	2	sessionID
2	1	flags
3	1	CRC
4	4	count
8	4	size
12	4	requestID
16	4	threadID
20	переменный	data

РЕАЛИЗАЦИЯ ДИСПЕТЧЕРА

При реализации диспетчера на C++ приходилось учитывать, что впоследствии его придётся переносить на другие языки и, возможно, платформы. Поэтому при реализации особое внимание уделялось созданию оболочек над системными функциями, позволяющих свести изменения при портировании к минимуму (о схожести языков см. [2,3]). Для удобства разработки диспетчер был разделён на модули, каждый из которых решает свою подзадачу. Опишем модули, входящие в диспетчер, и функциональность, которую они должны обеспечить.

Определения базовых типов.

Язык C++ не фиксирует размер некоторых типов (см [4]). Например, целочисленный тип int, как правило, имеет размер машинного слова, т.е. при компиляции для 32–х битных платформ и 64–х битных его размер будет различаться. У каждого компилятора, как правило, есть специфичные заголовочные файлы, не описанные в стандарте, определяющие типы фиксированного размера, но при использовании другого компилятора под другой операционной системой подходящего заголовочного файла может не найтись. Также необходимо учитывать, что название типов в других языках программирования будет иным. Собственный заголовочный файл с описанием типов позволяет упростить портирование проекта под другие платформы, а также использовать единый способ наименования типов в других языках программирования.

Примитивы синхронизации.

Были созданы оболочки для двух примитивов: мьютекс и событие.

Мьютекс – взаимоисключающая блокировка. В каждый момент времени мьютекс может быть захвачен лишь одним потоком или находиться в освобождённом состоянии. Доступные методы по работе с мьютексом:

- Захватить (ожидать) мьютекс. Поток будет заблокирован до тех пор, пока не сможет захватить мьютекс или не истечёт указанное время ожидания захвата.
- Освободить ранее захваченный мьютекс.

Событие – примитив синхронизации, отличающийся от мьютекса следующим:

- Событие может быть переведено в сигнальное состояние любым потоком (мьютекс может освободить лишь захвативший его поток).
- Событие поддерживает функции не только по установке в сигнальное состояние, но и в сброшенное (мьютекс можно сбросить лишь при захвате).
- События бывают двух типов: с автосбросом (в этом случае при ожидании события оно сбрасывается, как и мьютекс) и ручным сбросом (при ожидании – состояние события не изменяется).

Событие обладает теми же методами доступа, что и мьютекс, а также дополнительным методом сброса состояния в несигнальное. При создании события необходимо задать его тип (с автосбросом или без).

Создание оболочек для этих двух примитивов позволяет:

- Упростить обработку ошибок. При работе с объектами операционной системы всегда существует вероятность неудачного вызова системных функций (например, из–за нехватки памяти). В диспетчере довольно активно используются примитивы синхронизации, и написание кода обрабатывающего неудачный код возврата при каждом вызове системных функций значительно увеличит объём кода. Игнорировать возможность неудачного выполнения системных функций также нежелательно, т.к. это может потом значительно усложнить отладку приложения. Ошибки выполнения системных функций, как правило, являются критическими, и обработка ошибки заключается в выводе

информации об ошибке и месте её возникновения и остановке приложения. Оболочки позволяют произвести проверку на ошибки внутри себя и скрыть обработку ошибок системных функций от разработчика.

- Автоматическое уничтожение объектов. Примитивы синхронизации операционной системы для уничтожения требуют вызова специальных функций. Если разработчик по каким то причинам не вызовет функцию уничтожения, то объект будет существовать в операционной системе до момента закрытия приложения. У оболочек код по уничтожению системного объекта расположен в деструкторе, что позволяет в большинстве случаев автоматизировать очистку и тем самым избежать некоторых ошибок. В качестве примера, иллюстрирующего возможную ошибку, можно назвать появление не перехватываемого исключения после создания системного объекта, в этом случае типичное выполнение кода прерывается, и предусмотренное разработчиком удаление системного объекта может не выполниться.
- В диспетчере вся работа с синхронизацией производится через оболочки, что позволяет при портировании лишь переписать оболочки с сохранением интерфейса, а многочисленное использование оболочек оставить без изменений.

Формирование сообщений об ошибках

Большинство системных функций при ошибках устанавливают код ошибки. В дополнение существует системная функция, которая по коду может попытаться сформировать текстовое сообщение. Таким образом, для максимальной наглядности отладки после каждого вызова функции, способной приводить к ошибке, необходимо:

- Проверить успешность завершения работы функции, в случае ошибки начать обработку ошибки.
- Вывести сообщение описывающее место возникновения ошибки
- Вывести код ошибки
- Попытаться получить описание ошибки по коду на языке локализации ОС, и вывести его.
- Вывести имя исходного файла и номер строчки, на которой произошла ошибка (по крайней мере, при компиляции в режиме отладки).

Писать соответствующий код при каждом обращении к системным функциям довольно неприятно, поэтому был создан набор макросов, осуществляющих эту работу. Макросы в зависимости от режима сборки проекта несколько меняют своё поведение (например, при сборке отладочной версии довольно удобно пользоваться assert, но при сборке финальной версии его использование не сможет прервать выполнение программы). При выводе текста описывающего ошибку на языке, совпадающем с локализацией операционной системы в которой запускается программа, возникли некоторые трудности, связанные с отсутствием поддержки unicode в консоли windows, которые на данный момент решены за счёт принудительной установки необходимой кодировки в консоли.

Абстракция ввода–вывода.

Как уже отмечалось ранее, для вызова удалённых методов пригоден любой транспорт, способный передать запрос и ответ. Абстракция ввода–вывода фиксирует для транспорта необходимый интерфейс, а именно:

- Передать блок информации фиксированного размера.
- Получить блок информации фиксированного размера.
- Закрыть транспорт.

- Инициировать транспорт, если это возможно (для сокетов это переподключение).

Диспетчер работает лишь с абстракцией транспорта, а как именно реализован транспорт (например, за счёт сокетов или каналов) для диспетчера несущественно. Это должно обеспечить лёгкость замены транспорта.

В соответствии с этим интерфейсом создана реализация на основе winsock. В ней решаются задачи:

- Сопряжение интерфейсов. Функции приёма и отправки данных winsock могут обработать меньше данных, чем запросил пользователь. Для обеспечения интерфейса транспорта необходимо организовать последовательный вызов этих функций до тех пор, пока не будут обработаны все данные.
- Сохранение настроек подключения, что необходимо для обеспечения восстановления канала связи.
- Инициализация Winsock DLL, что необходимо для обеспечения доступа к winsock функциям.
- Обработка ошибок выполнения winsock функций.

Пакет данных

Основное назначение пакета – обеспечить хранение данных. Создание класса описывающего пакет позволило добиться дополнительной функциональности:

- Упрощение работы с пакетом. Созданы функции, выполняющие типичные для пакетов действия:
  - Вычисление CRC
  - Проверка CRC
  - Управление флагами

- Возможность изменения внутреннего устройства пакета, без модификации кода в других частях диспетчера.
- Автоматическое управление памятью. Если в языке Java всё управление памятью происходит автоматически, то в C++ разработчик сам ответственен за освобождение памяти. Пакеты данных могут создаваться и освобождаться в различных частях диспетчера, и отслеживание своевременности удаления пакета становится сложной задачей. Создание автоматического управления памятью для пакетов на основе идеи, используемой в "умных" указателях, позволяет упростить освобождение ресурсов. Это также позволяет избежать ошибок не освобождения занятой памяти при возникновении исключений.

Диспетчер пакетов

В многопоточном приложении возможно одновременное обращение к общему ресурсу из различных потоков. В случае диспетчера одним из общих ресурсов является транспорт. Диспетчер пакетов занимается синхронизацией потоков при их обращении к транспорту. Он обеспечивает:

- Возможность одновременного чтения из транспорта одним потоком и записи другим.
- Невозможность смешения пакетов. Если один поток начал чтение пакета, то транспорт будет заблокирован для других потоков до момента завершения операции для всего пакета. Аналогично для записи.
- В случае обнаружения ошибки в транспорте:
  1. Предотвратить (заблокировать) все последующие запросы к транспорту.

2. Ожидать завершения всех текущих операций с транспортом. Например, если обнаружена ошибка при выполнении чтения, а в это время параллельно выполняется операция записи, то ожидать завершения операции записи (успешной или нет).
3. Информировать остальные части диспетчера о разрушении транспорта, что обеспечит возможность восстановления связи.

Такая последовательность действий позволяет в случае одновременного обнаружения ошибки при чтении и записи корректно сформировать лишь одно сообщение о закрытии транспорта. В тот момент, когда диспетчер пакетов сообщает о разрушении транспорта, все запросы к транспорту завершены и заблокированы, и можно начинать работу по восстановлению канала.

Подсистема идентификаторов запросов

Как уже отмечалось выше, при отправке запроса диспетчер должен заполнить поле идентификатора запроса некоторым числом и заблокировать поток до тех пор, пока не будет получен ответ с тем же самым идентификатором. Подсистема идентификаторов запросов реализует управление идентификаторами. Рассмотрим предоставляемые ею функции:

- Создать новый (неиспользуемый) идентификатор.
- Зарегистрировать идентификатор в списке ожидаемых идентификаторов, и противоположная функция – удалить из списка ожидаемых.
- Найти идентификатор среди ожидаемых идентификаторов.
- Удалить (освободить) идентификатор.

Функции создания и удаления используют пул для ускорения работы. Т.к при работе с идентификаторами используются общие таблицы, то

осуществляется необходимая синхронизация доступа на основе примитивов синхронизации.

Диспетчер потоков

Центральный элемент диспетчера, обеспечивает приём и отправку пакетов (причём пакеты бывают двух типов: запросы и ответы). Рассмотрим поведение диспетчера потоков для всех четырёх вариантов:

- Неблокируемая отправка, т.е. отправка пакета, на который не ожидается возврата. В этом случае диспетчер отправляет пакет и возвращает управление вызывающему потоку.
- Блокируемая отправка, т.е. отправка, ожидающая результата. В этом случае:
  - Генерируется идентификатор запроса и ставится в очередь ожидаемых.
  - Отсылается пакет
  - Поток блокируется до получения ответа на этот запрос.
  - После снятия блокировки (получения ответа) уничтожается запрос.
  - Ответ на запрос возвращается вызывающему потоку.
- Получение пакета, являющегося ответом:
  - Поиск в таблице ожидаемых запросов запроса с указанным идентификатором, если не найден – игнорирование пакета.
  - Удаление идентификатора из списка ожидаемых запросов.
  - Передача пакета потоку, заблокированному в ожидании этого пакета.
  - Разблокировка потока.
- Получение пакета, являющегося запросом. Необходимо осуществить просмотр зарегистрированных в диспетчере сервисов и передать пакет подходящему сервису.

ЗАКЛЮЧЕНИЕ

В рамках данной работы было осуществлено изучение и сравнение некоторых существующих способов объединения модулей, определены недостатки этих способов. Были выдвинуты требования к системе, позволяющей удалённо вызывать методы, и принято участие в её проектировании. Разработан формат сообщений, обеспечивающий необходимую функциональность. Реализован диспетчер для языка C++.

Схемы маршалинга и генераторы прокси и сервисов выходят за рамки данной работы, но в дальнейшем планируется их проектирование и реализация.

Создаваемая система объединения модулей, к тому же позволяющая производить удалённый вызов методов, может оказаться полезной в различных проектах. В настоящий момент предполагается использование этой платформы для упрощения создания турниров программ, проводимых в УрГУ в рамках соревнований спортивного программирования.

ЛИТЕРАТУРА

1. Kinnersley B. Collected Information On About 2500 Computer Languages, Past and Present [Электрон. ресурс]. Режим доступа: http://people.ku.edu/~nkinners/LangList/Extras/langlist.htm
2. Хорстманн К.С., Корнелл Г. Библиотека профессионала. Java 2. Том 1. Основы / Пер. с англ. – М.: Издательский дом "Вильямс", 2004. – 848с.
3. Хорстманн К.С., Корнелл Г. Библиотека профессионала. Java 2. Том 2. Тонкости программирования / Пер. с англ. – М.: Издательский дом "Вильямс", 2004. – 1120с.
4. Страуструп Б. Язык программирования C++. Специальное издание / Пер. с англ. – М.: ООО "Бином–Пресс", 2006. – 1104с.

РЕФЕРАТ

Полетаев Д.Г. ОБ ОБЪЕДИНЕНИИ МОДУЛЕЙ НА ЯЗЫКАХ C++, DELPHI, C#, JAVA В ЕДИНУЮ ПРОГРАММУ, дипломная работа: стр. 35, табл. 3,рис 3, библ. 4 назв.

Ключевые слова: удалённый вызов методов, диспетчер, синхронизация потоков.

Рассматривается задача объединения модулей с помощью удаленного вызова методов. Описываются необходимые компоненты, их зависимости и функциональные возможности. Документируются особенности реализации диспетчера системы вызова удалённых методов для языка C++.