Удалённый вызов процедур (или Вызов удалённых процедур) (от англ. Remote Procedure Call (RPC)) — класс технологий, позволяющих компьютерным программам вызывать функции или процедуры в другом адресном пространстве (как правило, на удалённых компьютерах). Обычно, реализация RPC технологии включает в себя два компонента: сетевой протокол для обмена в режиме клиент-сервер и язык сериализации объектов (или структур, для необъектных RPC). Различные реализации RPC имеют очень отличающуюся друг от друга архитектуру и разнятся в своих возможностях: одни реализуют архитектуру SOA, другие CORBA или DCOM. На транспортном уровне RPC используют в основном протоколы TCP и UDP, однако, некоторые построены на основе HTTP (что нарушает архитектуру ISO/OSI, так как HTTP изначально не транспортный протокол).

Принцип

Идея вызова удалённых процедур (Remote Procedure Call — RPC) состоит в расширении хорошо известного и понятного механизма передачи управления и данных внутри программы, выполняющейся на одной машине, на передачу управления и данных через сеть. Средства удалённого вызова процедур предназначены для облегчения организации распределённых вычислений и создания распределенных клиент-серверных информационных систем. Наибольшая эффективность использования RPC достигается в тех приложениях, в которых существует интерактивная связь между удалёнными компонентами с небольшим временем ответов и относительно малым количеством передаваемых данных. Такие приложения называются RPC-ориентированными.

Характерными чертами вызова локальных процедур являются:

Асимметричность, то есть одна из взаимодействующих сторон является инициатором;

Синхронность, то есть выполнение вызывающей процедуры приостанавливается с момента выдачи запроса и возобновляется только после возврата из вызываемой процедуры.

Реализация удалённых вызовов существенно сложнее реализации вызовов локальных процедур. Можно обозначить следующие проблемы и задачи, которые необходимо решить при реализации RPC:

Так как вызывающая и вызываемая процедуры выполняются на разных машинах, то они имеют разные адресные пространства, и это создает проблемы при передаче параметров и результатов, особенно если машины находятся под управлением различных операционных систем или имеют различную архитектуру (например, используется прямой или обратный порядок байтов). Так как RPC не может рассчитывать на разделяемую память, то это означает, что параметры RPC не должны содержать указателей на ячейки нестековой памяти и что значения параметров должны копироваться с одного компьютера на другой. Для копирования параметров процедуры и результата выполнения через сеть выполняется их сериализация.

В отличие от локального вызова удалённый вызов процедур обязательно использует транспортный уровень сетевой архитектуры (например TCP), однако это остается скрытым от разработчика.

Выполнение вызывающей программы и вызываемой локальной процедуры в одной машине реализуется в рамках единого процесса. Но в реализации RPC участвуют как минимум два процесса — по одному в каждой машине. В случае, если один из них аварийно завершится, могут возникнуть следующие ситуации: при аварии вызывающей процедуры удалённо вызванные процедуры станут «осиротевшими», а при аварийном завершении удалённых процедур станут «обездоленными родителями» вызывающие процедуры, которые будут безрезультатно ожидать ответа от удалённых процедур.

Существует ряд проблем, связанных с неоднородностью языков программирования и операционных сред: структуры данных и структуры вызова процедур, поддерживаемые в каком-либо одном языке программирования, не поддерживаются точно так же во всех других языках. Таким образом имеется проблема совместимости, до сих пор не решённая ни с помощью введения одного общепринятого стандарта, ни с помощью реализации нескольких конкурирующих стандартов на всех архитектурах и во всех языках.

**Вызов удаленных процедур (RPC)**

**Концепция удаленного вызова процедур**

Идея вызова удаленных процедур *(Remote Procedure Call - RPC)* состоит в расширении хорошо известного и понятного механизма передачи управления и данных внутри программы, выполняющейся на одной машине, на передачу управления и данных через сеть. Средства удаленного вызова процедур предназначены для облегчения организации распределенных вычислений. Наибольшая эффективность использования RPC достигается в тех приложениях, в которых существует интерактивная связь между удаленными компонентами с небольшим временем ответов и относительно малым количеством передаваемых данных. Такие приложения называются RPC-ориентированными.

Характерными чертами вызова локальных процедур являются:

* Асимметричность, то есть одна из взаимодействующих сторон является инициатором;
* Синхронность, то есть выполнение вызывающей процедуры при останавливается с момента выдачи запроса и возобновляется только после возврата из вызываемой процедуры.

Реализация удаленных вызовов существенно сложнее реализации вызовов локальных процедур. Начнем с того, что поскольку вызывающая и вызываемая процедуры выполняются на разных машинах, то они имеют разные адресные пространства, и это создает проблемы при передаче параметров и результатов, особенно если машины не идентичны. Так как RPC не может рассчитывать на разделяемую память, то это означает, что параметры RPC не должны содержать указателей на ячейки нестековой памяти и что значения параметров должны копироваться с одного компьютера на другой. Следующим отличием RPC от локального вызова является то, что он обязательно использует нижележащую систему связи, однако это не должно быть явно видно ни в определении процедур, ни в самих процедурах. Удаленность вносит дополнительные проблемы. Выполнение вызывающей программы и вызываемой локальной процедуры в одной машине реализуется в рамках единого процесса. Но в реализации RPC участвуют как минимум два процесса - по одному в каждой машине. В случае, если один из них аварийно завершится, могут возникнуть следующие ситуации: при аварии вызывающей процедуры удаленно вызванные процедуры станут "осиротевшими", а при аварийном завершении удаленных процедур станут "обездоленными родителями" вызывающие процедуры, которые будут безрезультатно ожидать ответа от удаленных процедур.

Кроме того, существует ряд проблем, связанных с неоднородностью языков программирования и операционных сред: структуры данных и структуры вызова процедур, поддерживаемые в каком-либо одном языке программирования, не поддерживаются точно так же во всех других языках.

Эти и некоторые другие проблемы решает широко распространенная технология RPC, лежащая в основе многих распределенных операционных систем.

**Базовые операции RPC**

Чтобы понять работу RPC, рассмотрим вначале выполнение вызова локальной процедуры в обычной машине, работающей автономно. Пусть это, например, будет системный вызов

count=read (fd,buf,nbytes);

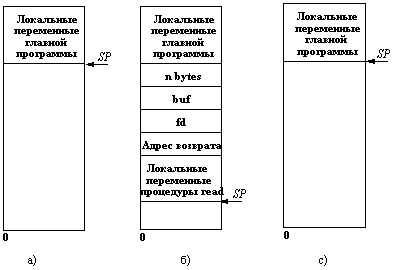
где fd - целое число,   
buf - массив символов,   
nbytes - целое число.

Чтобы осуществить вызов, вызывающая процедура заталкивает параметры в стек в обратном порядке (рисунок 3.1). После того, как вызов read выполнен, он помещает возвращаемое значение в регистр, перемещает адрес возврата и возвращает управление вызывающей процедуре, которая выбирает параметры из стека, возвращая его в исходное состояние. Заметим, что в языке С параметры могут вызываться или по ссылке (by name), или по значению (by value). По отношению к вызываемой процедуре параметры-значения являются инициализируемыми локальными переменными. Вызываемая процедура может изменить их, и это не повлияет на значение оригиналов этих переменных в вызывающей процедуре.

Если в вызываемую процедуру передается указатель на переменную, то изменение значения этой переменной вызываемой процедурой влечет изменение значения этой переменной и для вызывающей процедуры. Этот факт весьма существенен для RPC.

Существует также другой механизм передачи параметров, который не используется в языке С. Он называется call-by-copy/restore и состоит в необходимости копирования вызывающей программой переменных в стек в виде значений, а затем копирования назад после выполнения вызова поверх оригинальных значений вызывающей процедуры.

Решение о том, какой механизм передачи параметров использовать, принимается разработчиками языка. Иногда это зависит от типа передаваемых данных. В языке С, например, целые и другие скалярные данные всегда передаются по значению, а массивы - по ссылке.



*Рис. 3.1. а) Стек до выполнения вызова read;   
б) Стек во время выполнения процедуры;   
в) Стек после возврата в вызывающую программу*

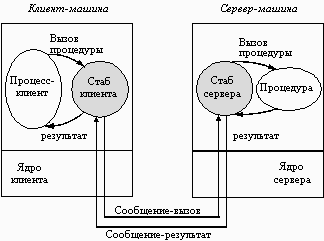
Идея, положенная в основу RPC, состоит в том, чтобы сделать вызов удаленной процедуры выглядящим по возможности также, как и вызов локальной процедуры. Другими словами - сделать RPC прозрачным: вызывающей процедуре не требуется знать, что вызываемая процедура находится на другой машине, и наоборот.

RPC достигает прозрачности следующим путем. Когда вызываемая процедура действительно является удаленной, в библиотеку помещается вместо локальной процедуры другая версия процедуры, называемая клиентским стабом (stub - заглушка). Подобно оригинальной процедуре, стаб вызывается с использованием вызывающей последовательности (как на рисунке 3.1), так же происходит прерывание при обращении к ядру. Только в отличие от оригинальной процедуры он не помещает параметры в регистры и не запрашивает у ядра данные, вместо этого он формирует сообщение для отправки ядру удаленной машины.

**Этапы выполнения RPC**

Взаимодействие программных компонентов при выполнении удаленного вызова процедуры иллюстрируется рисунком 3.2. После того, как клиентский стаб был вызван программой-клиентом, его первой задачей является заполнение буфера отправляемым сообщением. В некоторых системах клиентский стаб имеет единственный буфер фиксированной длины, заполняемый каждый раз с самого начала при поступлении каждого нового запроса. В других системах буфер сообщения представляет собой пул буферов для отдельных полей сообщения, причем некоторые из этих буферов уже заполнены. Этот метод особенно подходит для тех случаев, когда пакет имеет формат, состоящий из большого числа полей, но значения многих из этих полей не меняются от вызова к вызову.

Затем параметры должны быть преобразованы в соответствующий формат и вставлены в буфер сообщения. К этому моменту сообщение готово к передаче, поэтому выполняется прерывание по вызову ядра.



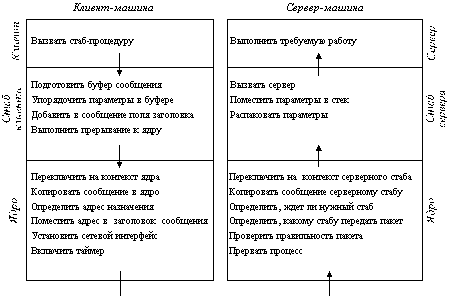
*Рис. 3.2. Remote Procedure Call*

Когда ядро получает управление, оно переключает контексты, сохраняет регистры процессора и карту памяти (дескрипторы страниц), устанавливает новую карту памяти, которая будет использоваться для работы в режиме ядра. Поскольку контексты ядра и пользователя различаются, ядро должно точно скопировать сообщение в свое собственное адресное пространство, так, чтобы иметь к нему доступ, запомнить адрес назначения (а, возможно, и другие поля заголовка), а также оно должно передать его сетевому интерфейсу. На этом завершается работа на клиентской стороне. Включается таймер передачи, и ядро может либо выполнять циклический опрос наличия ответа, либо передать управление планировщику, который выберет какой-либо другой процесс на выполнение. В первом случае ускоряется выполнение запроса, но отсутствует мультипрограммирование.

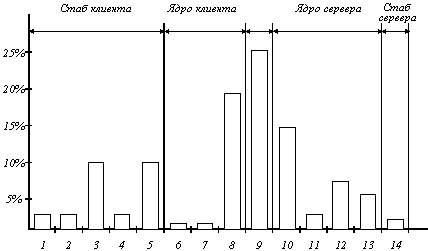
На стороне сервера поступающие биты помещаются принимающей аппаратурой либо во встроенный буфер, либо в оперативную память. Когда вся информация будет получена, генерируется прерывание. Обработчик прерывания проверяет правильность данных пакета и определяет, какому стабу следует их передать. Если ни один из стабов не ожидает этот пакет, обработчик должен либо поместить его в буфер, либо вообще отказаться от него. Если имеется ожидающий стаб, то сообщение копируется ему. Наконец, выполняется переключение контекстов, в результате чего восстанавливаются регистры и карта памяти, принимая те значения, которые они имели в момент, когда стаб сделал вызов receive.

Теперь начинает работу серверный стаб. Он распаковывает параметры и помещает их соответствующим образом в стек. Когда все готово, выполняется вызов сервера. После выполнения процедуры сервер передает результаты клиенту. Для этого выполняются все описанные выше этапы, только в обратном порядке.

Рисунок 3.3 показывает последовательность команд, которую необходимо выполнить для каждого RPC-вызова, а рисунок 3.4 - какая доля общего времени выполнения RPC тратится на выполнение каждого их описанных 14 этапов. Исследования были проведены на мультипроцессорной рабочей станции DEC Firefly, и, хотя наличие пяти процессоров обязательно повлияло на результаты измерений, приведенная на рисунке гистограмма дает общее представление о процессе выполнения RPC.



*Рис. 3.3. Этапы выполнения процедуры RPC*



*Рис. 3.4. Распределение времени между 14 этапами выполнения RPC*

***1. Вызов стаба***

***2. Подготовить буфер***

***3. Упаковать параметры***

***4. Заполнить поле заголовка***

***5. Вычислить контрольную сумму в сообщении***

***6. Прерывание к ядру***

***7. Очередь пакета на выполнение***

***8. Передача сообщения контроллеру по шине QBUS***

***9. Время передачи по сети Ethernet***

***10. Получить пакет от контроллера***

***11. Процедура обработки прерывания***

***12. Вычисление контрольной суммы***

***13. Переключение контекста в пространство пользователя***

***14. Выполнение серверного стаба***

**Динамическое связывание**

Рассмотрим вопрос о том, как клиент задает месторасположение сервера. Одним из методов решения этой проблемы является непосредственное использование сетевого адреса сервера в клиентской программе. Недостаток такого подхода - его чрезвычайная негибкость: при перемещении сервера, или при увеличении числа серверов, или при изменении интерфейса во всех этих и многих других случаях необходимо перекомпилировать все программы, которые использовали жесткое задание адреса сервера. Для того, чтобы избежать всех этих проблем, в некоторых распределенных системах используется так называемое динамическое связывание.

Начальным моментом для динамического связывания является формальное определение (спецификация) сервера. Спецификация содержит имя файл-сервера, номер версии и список процедур-услуг, предоставляемых данным сервером для клиентов (рисунок 3.5). Для каждой процедуры дается описание ее параметров с указанием того, является ли данный параметр входным или выходным относительно сервера. Некоторые параметры могут быть одновременно входными и выходными - например, некоторый массив, который посылается клиентом на сервер, модифицируется там, а затем возвращается обратно клиенту (операция copy/ restore).

*Рис. 3.5. Спецификация сервера RPC*

Формальная спецификация сервера используется в качестве исходных данных для программы-генератора стабов, которая создает как клиентские, так и серверные стабы. Затем они помещаются в соответствующие библиотеки. Когда пользовательская (клиентская) программа вызывает любую процедуру, определенную в спецификации сервера, соответствующая стаб-процедура связывается с двоичным кодом программы. Аналогично, когда компилируется сервер, с ним связываются серверные стабы.

При запуске сервера самым первым его действием является передача своего серверного интерфейса специальной программе, называемой binder'ом. Этот процесс, известный как процесс регистрации сервера, включает передачу сервером своего имени, номера версии, уникального идентификатора и описателя местонахождения сервера. Описатель системно независим и может представлять собой IP, Ethernet, X.500 или еще какой-либо адрес. Кроме того, он может содержать и другую информацию, например, относящуюся к аутентификации.

Когда клиент вызывает одну из удаленных процедур первый раз, например, read, клиентский стаб видит, что он еще не подсоединен к серверу, и посылает сообщение binder-программе с просьбой об импорте интерфейса нужной версии нужного сервера. Если такой сервер существует, то binder передает описатель и уникальный идентификатор клиентскому стабу.

Клиентский стаб при посылке сообщения с запросом использует в качестве адреса описатель. В сообщении содержатся параметры и уникальный идентификатор, который ядро сервера использует для того, чтобы направить поступившее сообщение в нужный сервер в случае, если их несколько на этой машине.

Этот метод, заключающийся в импорте/экспорте интерфейсов, обладает высокой гибкостью. Например, может быть несколько серверов, поддерживающих один и тот же интерфейс, и клиенты распределяются по серверам случайным образом. В рамках этого метода становится возможным периодический опрос серверов, анализ их работоспособности и, в случае отказа, автоматическое отключение, что повышает общую отказоустойчивость системы. Этот метод может также поддерживать аутентификацию клиента. Например, сервер может определить, что он может быть использован только клиентами из определенного списка.

Однако у динамического связывания имеются недостатки, например, дополнительные накладные расходы (временные затраты) на экспорт и импорт интерфейсов. Величина этих затрат может быть значительна, так как многие клиентские процессы существуют короткое время, а при каждом старте процесса процедура импорта интерфейса должна быть снова выполнена. Кроме того, в больших распределенных системах может стать узким местом программа binder, а создание нескольких программ аналогичного назначения также увеличивает накладные расходы на создание и синхронизацию процессов.

**Семантика RPC в случае отказов**

В идеале RPC должен функционировать правильно и в случае отказов. Рассмотрим следующие классы отказов:

1. Клиент не может определить местонахождения сервера, например, в случае отказа нужного сервера, или из-за того, что программа клиента была скомпилирована давно и использовала старую версию интерфейса сервера. В этом случае в ответ на запрос клиента поступает сообщение, содержащее код ошибки.
2. Потерян запрос от клиента к серверу. Самое простое решение - через определенное время повторить запрос.
3. Потеряно ответное сообщение от сервера клиенту. Этот вариант сложнее предыдущего, так как некоторые процедуры не являются идемпотентными. Идемпотентной называется процедура, запрос на выполнение которой можно повторить несколько раз, и результат при этом не изменится. Примером такой процедуры может служить чтение файла. Но вот процедура снятия некоторой суммы с банковского счета не является идемпотентной, и в случае потери ответа повторный запрос может существенно изменить состояние счета клиента. Одним из возможных решений является приведение всех процедур к идемпотентному виду. Однако на практике это не всегда удается, поэтому может быть использован другой метод - последовательная нумерация всех запросов клиентским ядром. Ядро сервера запоминает номер самого последнего запроса от каждого из клиентов, и при получении каждого запроса выполняет анализ - является ли этот запрос первичным или повторным.
4. Сервер потерпел аварию после получения запроса. Здесь также важно свойство идемпотентности, но к сожалению не может быть применен подход с нумерацией запросов. В данном случае имеет значение, когда произошел отказ - до или после выполнения операции. Но клиентское ядро не может распознать эти ситуации, для него известно только то, что время ответа истекло. Существует три подхода к этой проблеме:

* Ждать до тех пор, пока сервер не перезагрузится и пытаться выполнить операцию снова. Этот подход гарантирует, что RPC был выполнен до конца по крайней мере один раз, а возможно и более.
* Сразу сообщить приложению об ошибке. Этот подход гарантирует, что RPC был выполнен не более одного раза.
* Третий подход не гарантирует ничего. Когда сервер отказывает, клиенту не оказывается никакой поддержки. RPC может быть или не выполнен вообще, или выполнен много раз. Во всяком случае этот способ очень легко реализовать.

Ни один из этих подходов не является очень привлекательным. А идеальный вариант, который бы гарантировал ровно одно выполнение RPC, в общем случае не может быть реализован по принципиальным соображениям. Пусть, например, удаленной операцией является печать некоторого текста, которая включает загрузку буфера принтера и установку одного бита в некотором управляющем регистре принтера, в результате которой принтер стартует. Авария сервера может произойти как за микросекунду до, так и за микросекунду после установки управляющего бита. Момент сбоя целиком определяет процедуру восстановления, но клиент о моменте сбоя узнать не может. Короче говоря, возможность аварии сервера радикально меняет природу RPC и ясно отражает разницу между централизованной и распределенной системой. В первом случае крах сервера ведет к краху клиента, и восстановление невозможно. Во втором случае действия по восстановлению системы выполнить и возможно, и необходимо.

1. Клиент потерпел аварию после отсылки запроса. В этом случае выполняются вычисления результатов, которых никто не ожидает. Такие вычисления называют "сиротами". Наличие сирот может вызвать различные проблемы: непроизводительные затраты процессорного времени, блокирование ресурсов, подмена ответа на текущий запрос ответом на запрос, который был выдан клиентской машиной еще до перезапуска системы.

Как поступать с сиротами? Рассмотрим 4 возможных решения.

* *Уничтожение.* До того, как клиентский стаб посылает RPC-сообщение, он делает отметку в журнале, оповещая о том, что он будет сейчас делать. Журнал хранится на диске или в другой памяти, устойчивой к сбоям. После аварии система перезагружается, журнал анализируется и сироты ликвидируются. К недостаткам такого подхода относятся, во-первых, повышенные затраты, связанные с записью о каждом RPC на диск, а, во-вторых, возможная неэффективность из-за появления сирот второго поколения, порожденных RPC-вызовами, выданными сиротами первого поколения.
* *Перевоплощение*. В этом случае все проблемы решаются без использования записи на диск. Метод состоит в делении времени на последовательно пронумерованные периоды. Когда клиент перезагружается, он передает широковещательное сообщение всем машинам о начале нового периода. После приема этого сообщения все удаленные вычисления ликвидируются. Конечно, если сеть сегментированная, то некоторые сироты могут и уцелеть.
* *Мягкое перевоплощение* аналогично предыдущему случаю, за исключением того, что отыскиваются и уничтожаются не все удаленные вычисления, а только вычисления перезагружающегося клиента.
* *Истечение срока*. Каждому запросу отводится стандартный отрезок времени Т, в течение которого он должен быть выполнен. Если запрос не выполняется за отведенное время, то выделяется дополнительный квант. Хотя это и требует дополнительной работы, но если после аварии клиента сервер ждет в течение интервала Т до перезагрузки клиента, то все сироты обязательно уничтожаются.

На практике ни один из этих подходов не желателен, более того, уничтожение сирот может усугубить ситуацию. Например, пусть сирота заблокировал один или более файлов базы данных. Если сирота будет вдруг уничтожен, то эти блокировки останутся, кроме того уничтоженные сироты могут остаться стоять в различных системных очередях, в будущем они могут вызвать выполнение новых процессов и т.п.

XML-RPC

XML-RPC (сокр. от англ. Extensible Markup Language Remote Procedure Call — XML-вызов удалённых процедур) — стандарт/протокол вызова удалённых процедур, основанный на XML, является прародителем SOAP, отличается исключительной простотой применения. XML-RPC, как и любой другой интерфейс RPC, определяет набор стандартных типов данных и команд, которые программист может использовать для доступа к функциональности другой программы, находящейся на другом компьютере в сети.

Протокол XML-RPC был изначально разработан Дэйвом Винером из компании «UserLand Software» в сотрудничестве с [Майкрософт](http://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D0%B9%D0%BA%D1%80%D0%BE%D1%81%D0%BE%D1%84%D1%82) в [1998 году](http://ru.wikipedia.org/wiki/1998_%D0%B3%D0%BE%D0%B4). Однако [корпорация](http://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D1%80%D0%BF%D0%BE%D1%80%D0%B0%D1%86%D0%B8%D1%8F) Майкрософт вскоре сочла этот протокол слишком упрощённым, и начала расширять его функциональность. После нескольких циклов по расширению функциональности, появилась система, ныне известная как SOAP. Позднее Майкрософт начала широко рекламировать и внедрять [SOAP](http://ru.wikipedia.org/wiki/SOAP), а изначальный XML-RPC был отвергнут. Но, несмотря на отвержение Майкрософт, стандарт XML-RPC очаровал многих программистов своей необычайной простотой и, за счёт этого, существует по сей день и даже постепенно набирает популярность.

## Типы данных

|  |  |  |
| --- | --- | --- |
| **Имя типа** | **Пример тега** | **Описание типа** |
| array | <array>  <data>  <value><i4>1404</i4></value>  <value><string>Что-нибудь здесь</string></value>  <value><i4>1</i4></value>  </data>  </array> | Массив величин, без ключей |
| base64 | <base64>eW91IGNhbid0IHJlYWQgdGhpcyE=</base64> | Кодированные в [Base64](http://ru.wikipedia.org/wiki/Base64) двоичные данные |
| boolean | <boolean>1</boolean> | Логическая (булева) величина (0 или 1) |
| date/time | <dateTime.iso8601>19980717T14:08:55</dateTime.iso8601> | Дата и время |
| double | <double>-12.53</double> | Дробная величина двойной точности |
| integer | <i4>42</i4> | Целое число |
| string | <string>Здравствуй, Мир!</string> | Строка символов (в той же кодировке, что и весь [XML](http://ru.wikipedia.org/wiki/XML)-документ) |
| struct | <struct>  <member>  <name>Что-то</name>  <value><i4>1</i4></value>  </member>  <member>  <name>Ещё что-то</name>  <value><i4>2</i4></value>  </member>  </struct> | Массив величин, с ключами |
| nil | <nil/> | Нулевая (пустая) величина — это [расширение](http://ontosys.com/xml-rpc/extensions.php) XML-RPC |

## [[править](http://ru.wikipedia.org/w/index.php?title=XML-RPC&action=edit&section=3)] Примеры

Типичный пример запроса XML-RPC:

<?xml version="1.0"?>

<methodCall>

<methodName>examples.getStateName</methodName>

<params>

<param>

<value><i4>41</i4></value>

</param>

</params>

</methodCall>

Типичный пример ответа на запрос XML-RPC:

<?xml version="1.0"?>

<methodResponse>

<params>

<param>

<value><string>South Dakota</string></value>

</param>

</params>

</methodResponse>

Типичный пример отчёта об ошибке в запросе XML-RPC:

<?xml version="1.0"?>

<methodResponse>

<fault>

<value>

<struct>

<member>

<name>faultCode</name>

<value><int>4</int></value>

</member>

<member>

<name>faultString</name>

<value><string>Too many parameters.</string></value>

</member>

</struct>

</value>

</fault>

</methodResponse>

#### На сцене - XML-RPC

RPC - удаленный вызов процедур с помощью XML. Сама методика удаленного вызова процедуры известна давно и используется в таких технологиях, как DCOM, SOAP, CORBA. RPC предназначен для построения распределенных клиент-серверных приложений. Это дает возможность строить приложения, которые работают в гетерогенных сетях, например на компьютерах различных систем, производить удаленную обработку данных и управление удаленными приложениями.

Приведем сильно упрощенный пример. Приложение, выполняя обработку некоторых данных на локальной машине, обращается к некоторой процедуре. Если ее реализация присутствует в программе, то процедура (функция) принимает параметры, выполняет действие и возвращает некоторые данные. Если это удаленный вызов, мы должны знать, где будет исполняться наша процедура. Запрос на выполнение процедуры вместе с параметрами записывается в виде XML-документа и посредством HTTP передается по сети на другой компьютер, где из XML-документа извлекается имя процедуры, параметры и прочая нужная информация. После завершения работы процедуры формируется ответ (например, возвращаемые данные) - и он передается компьютеру, пославшему запрос. Заметим, что для прикладной программы все действия совершенно прозрачны.

По этому принципу функционируют все системы, и различия в реализации и процедуре обмена не оказывают существенного влияния на его суть.

Хорошо, предположим, у нас есть возможность удаленно вызывать процедуры и функции - чего же нам еще? А вот чего. Формат обмена данными при классической модели RPC (DCOM, CORBA) остается бинарным - а значит, работать с ним сложнее, он не слишком подходит, если надо организовать работу распределенной системы, где между отдельными участками сети стоят firewall/прокси-серверы. Технология DCOM, например, реализована для Windows-систем, CORBA функционирует на разных платформах, но наиболее полноценна ее реализация на J2EE. Значит, всегда найдется (и действительно находится) такая конфигурация сети/платформ, чтобы для реализации распределенной системы в ней ни одна технология не подходила. Так что же делать?

Задавшись этим вопросом, компания [UserLand Software Inc.](http://www.xmlrpc.com/) создала технологию XML-RPC. Основным транспортом в ней является протокол HTTP; формат данных - XML. Это снимает ограничения, налагаемые как на конфигурацию сети, так и на маршрут следования пакетов,- вызовы XML-RPC представляют собой простой тип данных text/xml и свободно проходят сквозь шлюзы везде, где допускается ретрансляция http-трафика.

У новой технологии есть и другие преимущества. Применение XML для описания данных позволило упростить программные средства создания распределенных приложений, снизились требования к клиенту и серверу. Например, теперь есть возможность связать веб-планшет с сервером на работе и с домашним компьютером. Программы разбора (парсинга) XML сейчас существуют практически для всех операционных систем и на всех языках программирования - следовательно, препятствий для внедрения технологии вроде бы нет.

#### Что же это такое?

Рассмотрение XML-RPC проведем на упрощенном тестовом примере. Для снижения затрат мы разворачиваем систему, где на один компьютер (сервер) ставится мощное ПО для перевода, проверка синтаксиса и грамматики, а все клиенты обращаются к нему посредством XML-RPC. (Конечно, этот пример выдуман, чтобы легче было познакомить читателя с технологией - но, господа программисты, кто мешает реально сделать такую систему?)

Сообщение XML-RPC передается методом POST-протокола HTTP. Сообщения бывают трех типов: запрос, ответ и сообщение об ошибке.

|  |  |
| --- | --- |
| **Запрос** | |
| **XML-RPC запрос** | **Описание** |
| POST /RPC2 HTTP/1.0 User-Agent: MyAPP-Word/5.1.2 (WinNT) Host: server.localnet.com Content-Type: text/xml Content-length: 172 <? xml version="1.0"?> <methodCall> <methodName>CheckWord</methodName> <params> <param> <value><string>проверка</string></value> </param> </params> </methodCall> | Сначала идет стандартный заголовок http-запроса. MIME-тип данных должен быть text/xml, длина также обязательно должна присутствовать и иметь корректное значение, равное длине передаваемого сообщения. Стандартный заголовок любого корректного XML-документа. Корневой узел. Не допускается вложенности тегов <methodCall> - значит, одним запросом мы можем вызвать только один метод. Тег <methodName> указывает на объект и название метода, который вызывается. Можно указывать так, как принято в языках программирования вызывать свойства класса: имя метода - через точку после имени класса. Можно также передавать пути и имя программы. Мы вызываем метод CheckWord объекта OrfoCheck. В секции <params> задаются параметры, которые передаются в метод. Секция может содержать произвольное число подэлементов <param>, содержащих параметр, который описывается тегом <value>. Параметры и типы данных мы рассмотрим чуть дальше. В нашем варианте методу передается один параметр, слово (оно заключено в тег <string>), которое надо проверить на правильность написания. Все теги, согласно спецификации XML, должны иметь соответствующие закрывающие элементы - в XML-RPC нет одиночных тегов. |

#### Типы данных

В протоколе XML-RPC предусмотрено семь простых типов данных и два сложных, для передачи параметров методу и возвращаемых значений. Эти типы отображают основные типы данных реальных языков программирования. Более сложные типы, такие, например, как объекты, нужно передавать в двоичном виде или заменять структурами.

Целые числа - задаются тегом <i4> или <int> и представляются 4-байтовыми целыми числами со знаком. Для задания отрицательных чисел ставится знак "-", например 34, 344, -15.

Логический тип данных представляется тегом <boolean> и может иметь значения 0 (false) или 1 (true). Можно использовать как 1/0, так и символьные константы true/false.

ASCII-строка - тип данных, принимаемый по умолчанию. Представляет собой просто строку символов, заключенную в теги <string></string>. В качестве символов нельзя использовать служебные знаки "<" и "&" - их следует передавать кодами &lt; и &amp; соответственно.

Числа с плавающей точкой. Задаются тегом <double> и представляют собой числа с плавающей точкой двойной точности. Как разделитель целой и дробной части используется знак ",". Пробелы недопустимы. Отрицательные числа задаются знаком "-" перед числом.

Дата/время. Для передачи времени/даты служит тег <dateTime.iso8601>. Пример времени - 19980717T14:08:55 (в спецификации написано, что сервер сам должен определять, как посылать время/дату. Использовать этот тип данных, пользоваться структурой или же просто передавать дату как строку не рекомендуется).

Двоичные данные передаются в закодированном (base64) виде и описываются тегом <base64>.

Структуры. Для передачи структурированных данных можно конструировать свои структуры. Структура определяется корневым элементом <struct>, который может содержать произвольное количество элементов <member>, определяющих каждый член структуры. Член структуры описывается двумя тегами: первый, <name>, описывает имя члена, второй, <value> содержит значение члена (вместе с тегом, описывающим тип данных). Например, так описывается структура с двух строковых элементов:

<struct>

<member>

<name>FirstWord</name>

<value><string>Hell</string></value>

</member>

<member>

<name>SecondWord</name>

<value><string>World!</string></value>

</member>

</struct>

Массивы. Массивы не имеют названия и описываются тегом <array>. Он содержит один элемент <data> и один или несколько дочерних элементов <value>, где задаются данные. В качестве элементов массива могут выступать любые другие типы в произвольном порядке, а также другие массивы - что позволяет описывать многомерные массивы. Так же можно описывать массив структур. Пример 4-элементного массива:

<array>

<data>

<value><i4>34</i4></value>

<value><string>Привет, Мир!</string></value>

<value><boolean>0</boolean></value>

<value><i4>-34</i4></value>

</data>

</array>

|  |  |
| --- | --- |
| **Ответ сервера** | |
| **XML-RPC ответ** | **Описание** |
| HTTP/1.1 200 OK Connection: close Content-Length: 166 Content-Type: text/xml Date: Fri, 17 Jul 1998 19:55:08 GMT Server: MyWordCheckSerwer/5.1.2-WinNT <? xml version="1.0"?> <methodResponse> <params> <param> <value><boolean>true</boolean></value> </param> </params> </methodResponse> Тело ответа при ошибке приложения <fault> <value> <struct> <member> <name>faultCode</name> <value><int>4</int></value> </member> <member> <name>faultString</name> <value> <string>Too many рarameters.</string> </value> </member> </struct> </value> </fault> | Сначала идет стандартный заголовок http-ответа сервера. MIME-тип данных должен быть text/xml, длина также должна обязательно присутствовать и иметь корректное значение, равное длине передаваемого сообщения. Стандартный заголовок любого корректного XML-документа. Корневой узел. Не допускается вложенности тегов <methodResponse>. Теги <params> и <param> аналогичны запросу и включают один или более элементов <value>, которые содержат значение, возвращенное методом. Если сервер отвечает HTTP-кодом 200 ОК - это значит, что запрос успешно обработан. Он уведомляет лишь о том, что данные по сети переданы правильно и сервер сумел их корректно обработать. Но метод также может вернуть ошибку - и это уже будет ошибка не протокола, а логики приложения. В таком случае передается сообщение и структура, которая описывает код ошибки и текстовое объяснение. В нашем примере передается структура из двух элементов: первый элемент содержит целочисленный код ошибки (4), второй элемент - текстовая строка, описывающая ошибку (Too many рarameters - неправильное число параметров). |

#### Окончательный вариант

Теперь можно окончательно описать работу нашего тестового примера. Итак, приложение MyAppWord (текстовый редактор) хочет перевести на английский, например, слово "world". Программа формирует запрос к серверу, вызывая процедуру перевода TranslateWord. Процедуре передается структура, содержащая слово, которое следует перевести, и направление перевода, которое задается символьной строкой - "en-ru".

**MyAppWord  
Запрос:**

POST /RPC2 HTTP/1.0   
User-Agent: MyAppWord/5.1.2 (WinNT)   
Host: server.localnet.com   
Content-Type: text/xml   
Content-length: 172   
<? xml version="1.0"?>   
<methodCall>   
<methodName>TranslateWord</methodName>   
<params>   
<param>   
<value>   
<struct>   
<member>   
<name>Word</name>   
<value><string>world</string></value>   
</member>   
<member>   
<name>typetranslate</name>   
<value><string>en-ru</string></value>   
</member>   
</struct>   
</param>   
</params>   
</methodCall>

Сервер, приняв наш запрос, передает его программе-демону, которая производит парсинг запроса, выделяет из него нужные данные и, найдя (например, по таблице) ссылку на нужный метод, вызывает его с переданными параметрами. Если тип и количество параметров правильные, то по окончании работы метода программа-демон принимает возвращенное значение, преобразует его в XML-описание и формирует ответ.

**MyAppWord  
Ответ:**

HTTP/1.1 200 OK   
Connection: close   
Content-Length: 166   
Content-Type: text/xml   
Date: Fri, 17 Jul 1998 19:55:08 GMT   
Server: MyWordCheckSerwer/5.1.2-WinNT   
<? xml version="1.0"?>   
<methodResponse>   
<params>   
<param>   
<struct>   
<member>   
<name>WordtoTranslate</name>   
<value><string>world</string></value>   
</member>   
<member>   
<name>translatesword</name>   
<value><string>мир</string></value>   
</member>   
<member>   
<name>typetranslate</name>   
<value><string>en-ru</string></value>   
</member>   
</struct>   
</param>   
</params>   
</methodResponse>

**MyAppWord  
Сообщение об ошибке:**

HTTP/1.1 200 OK   
Connection: close   
Content-Length: 166   
Content-Type: text/xml   
Date: Fri, 17 Jul 1998 19:55:08 GMT   
Server: MyWordCheckSerwer/5.1.2-WinNT   
<? xml version="1.0"?>   
<methodResponse>   
<fault>   
<value>   
<struct>   
<member>   
<name>faultCode</name>   
<value><int>10</int></value>   
</member>   
<member>   
<name>faultString</name>   
<value>   
<string>Перевод невозможен. Слово отсутствует в словаре.</string>   
</value>   
</member>   
</struct>   
</value></fault> </methodResponse>

Приложение получит такое сообщение, когда запрос на перевод не может быть удовлетворен, поскольку слова нет в словаре.

Хотя наш пример, на первый взгляд, кажется надуманным и простым, тем не менее, на нем показано, как можно уже сегодня использовать XML-RPC для решения конкретных задач. Конечно, его возможности намного шире, и можно, например, представить себе распределенную ОС, построенную на XML-RPC, или системы визуализации данных, построенные по архитектуре X Window, но с применением все того же XML-RPC.

#### XML-RPC vs SOAP

Если для реализации удаленного вызова вы используете XML, то у вас есть выбор: использовать XML-RPC или же SOAP (Simple Object Access Protocol). О последней уже написано множество статей, поэтому предлагаем только сравнить обе технологии.

Вот некоторые характеристики, которые определяют различия XML-RPC или же SOAP:

|  |  |  |
| --- | --- | --- |
| **Характеристика** | **XML-RPC** | **SOAP** |
| Скалярные типы данных | + | + |
| Структуры | + | + |
| Массивы | + | + |
| Именованные массивы и структуры | - | + |
| Определяемые разработчиком кодировки | - | + |
| Определяемые разработчиком типы данных | - | + |
| Детализация ошибок | + | + |
| Легкость освоения и практического применения | + | - |

Конечно, на первый взгляд "минус" в столбце SOAP встречается только единожды. Это создает иллюзию "всереализуемости всего" в нем. Но давайте присмотримся внимательнее. Основные типы данных у обоих конкурентов одинаковые. Но в XML-RPC отсутствует возможность задавать имена для массивов и структур (все структуры и массивы являются "анонимными"). Возможно, это упущение разработчиков, но решить эту проблему можно и самому, например вводя еще одну строковую переменную с именем массива или структуры (в случае, если таких объектов много, можно завести специальный массив "имен массивов").

С "определяемыми разработчиком кодировками" ситуация уже серьезнее. Сам механизм подобного ограничения не совсем ясен - ни стандарт XML, ни, тем более, транспортный уровень (протокол HTTP) таких ограничений не имеют. Да и стремление сделать клиент/сервер XML-RPC как можно более простым тоже не привело бы к возникновению подобного ограничения. Хотя, с другой стороны, SOAP тоже не блещет поддержкой кодировок (US-ASCII, UTF-8, UTF-16). Правда, в обеих технологиях есть возможность обойти все эти недостатки сразу - тип данных base64. Но выход ли это?

Посмотрим теперь на пункт "легкость в освоении и применении". В свете сегодняшних темпов развития технологий и стандартов, особенно Web, этот пункт приобретает большую важность. Реальна ситуация, когда крупный проект начинает разрабатываться на самой передовой основе - а в конце работы новый стандарт не только "уже не новый", но и "уже не стандарт вообще". Недавно W3C опубликовала черновой вариант SOAP Version 1.2 - поверьте, и объем, и сложность документации впечатляют. Трудности возникают даже на этапе ознакомительного чтения, не говоря уже о разработке. А вот спецификация XML-RPC занимает около трех страниц А4 и предельно проста.

Да, ни одна из этих технологий не является панацеей от всех бед и не претендует на полноту. Большинство программистов и разработчиков спецификаций сходятся на том, что:

* если вам нужна система для работы со сложной логикой, если вы передаете большие комплексные структуры данных, если вам нужна полная информация о клиенте, если вы хотите, чтобы запрос содержал в себе инструкции по его обработке, и, наконец, если для вас важно, чтобы за стандартом стояли гранды индустрии (Microsoft, IBM, Sun) - вам следует остановить свой выбор на SOAP;
* если же данные являются относительно простыми, а приложения должны работать на множестве платформ и на разных языках, если важна скорость работы и логика системы не нуждается в сложных командах - используйте XML-RPC.

Выбрав SOAP, сразу смиритесь с тем, что у вашей команды разработчиков (или у вас самих) впереди много головной боли, которая начнется уже с момента чтения спецификаций. Но ведь зато вы будете иметь возможность в конце концов построить систему практически любой сложности и функциональности.

Если выберете XML-RPC - написать программу клиента/сервера не составит труда даже начинающему программисту. Да и о выборе ПО можете не задумываться - хоть Borland Delphi/Kylix, хоть Phyton. Но не все задачи будут решаться сразу, а некоторые не будут решаться вообще.

**Заключение**

Не трудно увидеть, что стандарт XML-RPС очень прост - и в то же время применение XML как основного инструмента для описания данных позволяет сделать его очень гибким. Протокол можно модифицировать под каждую конкретную задачу, а использование хорошо зарекомендовавших себя стандартов на передачу данных (HTTP/HTTPS) позволяет успешно применять его на любых платформах, где имеется его поддержка.

**Ссылки для разработчиков**

Реализации XML-RPC существуют для множества платформ и языков программирования. Вот ссылки на наиболее интересные компоненты для:

* [Borland Delphi/Kylix](http://www.codepunk.com/)
* PHP - есть несколько вариантов:
  + [XMLRPC-EPI](http://xmlrpc-epi.sourceforge.net/)
  + [phpRPC](http://sourceforge.net/projects/phprpc)
  + [phpxmlrpc](http://phpxmlrpc.sourceforge.net/)
  + [Keith Devens' XML-RPC](http://www.keithdevens.com/software/xmlrpc) (самый простой)
* [XML-RPC for C/C++](http://xmlrpc-c.sourceforge.net/" \t "_blank)
* [XML-RPC for Python](http://www.pythonware.com/products/xmlrpc)
* [XML-RPC.NET for MS.NET environment](http://www.xml-rpc.net/" \t "_blank)

[Большой список](http://www.xmlrpc.com/directory/1568/implementations) реализаций XML-RPC.   
Кроме того, постоянно пополняется [список публичных XML-RPC-сервисов](http://www.xmlrpc.com/directory/1568/services).   
Одними из наиболее известных проектов, использующих XML-RPC, являются [Mozilla](http://www.mozilla.org/projects/xmlrpc) и [Apache](http://ws.apache.org/xmlrpc).