Платформы межкомпонентного взаимодействия в поисково-аналитических системах: состояние и перспективы[1]

Д.В. Зубарев, И.А. Тихомиров

Аннотация. В статье сделан обзор наиболее популярных и современных платформ организации межкомпонентного взаимодействия. Проведено сравнение этих платформ. Описаны перспективы развития способов межкомпонентного взаимодействия.

Ключевые слова: поисково-аналитические системы, межкомпонентное взаимодействие, распределенные системы, высоконагруженные системы.

Введение

Современные поисково-аналитические системы обладают модульной архитектурой и состоят из нескольких десятков модулей. Каждый модуль отвечает за свою задачу: краулер загружает документы из интернета, база данных хранит документы, индексатор строит из собранных документов индексы и т.д. При развертывании на одном или нескольких серверах подобная модульная система требует организации взаимодействия между компонентами. Такое взаимодействие можно организовать с помощью средств языка программирования или специализированных пакетов и библиотек для распараллеливания работы программ.

В настоящее время существуют узкоспециализированные языки программирования, обладающие встроенными средствами для создания распределенных приложений. Одним из таких языков является Erlang. Одна из главных особенностей Erlang состоит в использовании процессов, а не потоков для достижения параллельности. Эти процессы очень легковесны, поэтому работающие программы могут запускать большое количество таких процессов без серьезных накладных расходов [1]. Процессы исполняются в виртуальной машине, а взаимодействие между процессами не зависит от их местоположения. Однако Erlang, как и другие аналогичные языки, проигрывает в выразительной мощи распространенным языкам общего назначения. На нем нельзя реализовать сложные системы, хотя он и используется такими крупными компаниями, как Facebook и Amazon. Среди проектов, созданных помощью Erlang, краулер [2], распределенная база данных riak [3] и промежуточное программное обеспечение, ориентированное на обработку сообщений RabbitMQ [4].

Применительно к поисково-аналитическим машинам, обрабатывающим сотни миллионов документов, хорошо подходят средства распа-

[1] Работа выполнена при финансовой поддержке Минобрнауки России по государственному контракту от 08.06.2012 г. № 07.514.11.4134 в рамках ФЦП «Исследования и разработки по приоритетным направлениям развития научно-технологического комплекса России на 2007-2013 годы»

раллеливания по данным, так как наборы документов могут быть разделены естественным образом и обрабатываться независимо друг от друга на узлах кластера. Системы, использующие параллелизм по данным, легко масштабируются, потому что для увеличения производительности системы в целом требуется всего лишь добавить новые узлы в кластерную установку, которые будут обрабатывать свои порции данных независимо от других узлов. В данной статье будут рассмотрены наиболее популярные в настоящее время платформы, подходящие для распараллеливания по данным. В первых четырех разделах приведен обзор таких платформ, а в пятом разделе производится их сравнение.

1. Protocol buffers

Protocol buffers используется компанией Google в качестве инструмента сериализации структур данных, передаваемых между компонентами распределенной системы. Protocol buffers не включает реализацию RPC (Remote procedure call, удаленный вызов процедур), однако содержит необходимые средства для подключения любой RPC реализации [5]. В Google существует собственная реализация RPC, которая используется для связи всех внутренних компонентов поисковой системы [6]. В отличие от Protocol Buffers, исходные коды этой библиотеки не были открыты. Оригинальная версия Protocol Buffers от Google поддерживала языки C++, Java, Python, впоследствии появились сторонние генераторы для многих языков программирования [5].

Стандартная схема работы с библиотекой включает три шага:

- создается описание структур данных и интерфейсов модулей на языке IDL (interface description language); синтаксис IDL очень похож на синтаксис языка C, а получаемое описание намного меньше и понятнее, чем созданное с помощью языков XML или JSON;
- на основе этих описаний программа, поставляемая вместе с библиотекой, генерирует исходный код для поддерживаемых языков программирования; сгенерированный код содержит определенные в IDL структуры; структуры обладают простейшими методами доступа ко всем полям; реализуются также функции сериализации и десериализации;
- сгенерированные файлы подключаются к коду приложения, в котором осуществляется работа с объявленными в IDL структурами.

По состоянию на апрель 2012 года в базе исходных кодов Google описано 48,162 разных структур данных, используемых в их RPC системе [7].

Структуры данных в Protocol Buffer сериализуются в компактный бинарный формат. При этом не создается схемы, описывающей структуру формата. Это делает невозможным использование сериализованных данных вне системы Protocol buffers. Помимо бинарного, поддерживается также и текстовый формат, который полезен при отладке приложений.

Большим плюсом Protocol buffers является механизм обратной совместимости. Он позволяет добавлять новые поля в структуры данных и использовать с ними старые программы, созданные на основе старых структур данных. Старые программы будут игнорировать новые поля при десериализации. Это достигается с помощью тегирования каждого поля уникальным идентификатором. Тэг поля записывается перед его значением в бинарном формате. Это позволяет найти устаревшим серверам или клиентам поддерживаемые ими значения. Таким образом, для сохранения совместимости разработчику не следует менять идентификаторы полей.

2. Apache Thrift

Библиотека Apache Thrift похожа на Protocol buffers: эффективная бинарная сериализация, похожий IDL, поддержка обратной совместимости с помощью тегирования. Помимо этого, Thrift включает реализацию механизма RPC. Thrift был разработан в Facebook и впоследствии передан фонду Apache для поддержки и создания свободного сообщества разработчиков. Thrift активно используется в Facebook и во многих других проектах и компаниях [8].

Схема использования библиотеки, приведенная в предыдущем разделе для Protocol buffers, идентична и для Thrift. С помощью генераторов кода Thrift создает набор файлов, которые затем используются для реализации клиентов или серверов.

Thrift включает в себя RPC стек для организации взаимодействия между клиентом и сервером. Он представлен на Рис.1.

В верхней части находится код, написанный на языке IDL, описывающий интерфейс сервера и все структуры данных, которые он принимает или возвращает. Далее следует сгенерированный из этих описаний код для конкретного языка программирования. Для клиентов создаются клиентские заглушки, которые принимают все вызовы серверных методов и передают их дальше вниз по стеку. Для сервера создается обработчик запросов, в который разработчик должен передать экземпляр класса, реализующего определенный в IDL интерфейс. При поступлении вызова от клиента обязанностью обработчика является выбор и вызов соответствующего метода у этого экземпляра. Для структур данных, которые используются в качестве входных или выходных параметров (за исключением встроенных типов), генерируются функции read/write. Они обеспечивают сериализацию и десериализацию этих структур. Протокол и транспорт являются частью библиотеки времени выполнения Thrift. Поэтому допустимо изменять протокол и транспорт во время написания программы без повторной генерации кода. Допустимые виды протокола и транспорта рассмотрены ниже. В основе самой нижней части стека, отвечающей за передачу данных, лежат методы, предоставляемые конкретным языком программирования. Например, для ввода/вывода в Java и Python используются средства из стандартных библиотек этих языков. В то время как привязка для C++ использует свою собственную реализацию.

Thrift позволяет независимо друг от друга выбирать используемый протокол, транспорт и тип серверного обработчика запросов. Thrift поддерживает несколько бинарных протоколов, которые различаются по эффективности и размеру получаемого байтового массива. Существует возможность сериализовать структуры данных в формат JSON, а также в текстовый формат.

В то время как приведенные выше протоколы определяют формат передаваемых данных, транспорт Thrift определяет, каким образом сообщения передаются от клиента к серверу.

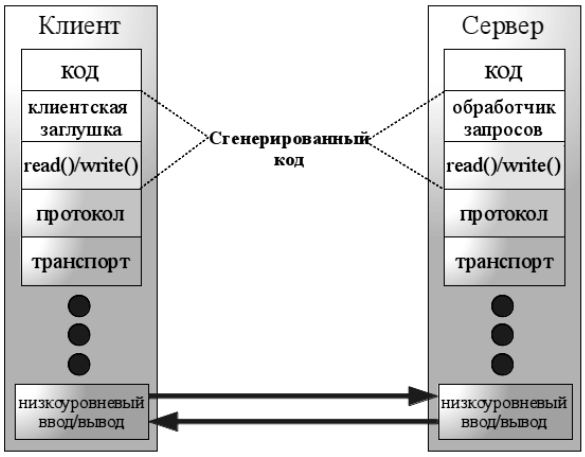

Рис. 1. RPC Стэк Thrift

Помимо TCP, являющегося стандартным для RPC видом транспорта, Thrift предоставляет другие виды транспорта. Среди них надстройка над транспортом TCP, которая делит данные перед отправкой на пакеты. Каждому пакету предшествует его длина. Этот транспорт необходим при использовании неблокирующего сервера. Thrift поддерживает также вывод сериализованных данных в файл или в буфер. Это может быть использовано для долговременного хранения структур данных. Функция компрессии передаваемых данных в Thrift оформлена в виде транспорта. Для того чтобы сжать данные, нужно создать двойной транспорт, на одном конце которого будет компрессор, а на другом любой из выше перечисленных транспортов.

Thrift также включает в себя классы, предоставляющие гибкость при разработке серверной части приложения. Базовый сервер является однопоточным и блокируется при операциях ввода/вывода. Для построения масштабируемых приложений Thrift поддерживает создание многопоточных серверов с неблокирующими операциями ввода/вывода [9].

Серьезным ограничением Thrift является то, что один сервер может одновременно принимать вызовы только для одного сервиса (сервисом в данном случае называется класс, реализующий IDL интерфейс). Это ограничение может быть устранено посредством создания одного общего интерфейса, наследующего от всех других интерфейсов. Таким образом, один сервер может предоставлять несколько серви-

сов, при этом всегда можно создать несколько серверов. Основной недостаток подхода – чрезмерное потребление системных ресурсов (порты, память и т.д.) [10].

3. Zeroc ICE

Zeroc ICE (Internet Communications Engine) – это объектно-ориентированное промежуточное ПО [11], которое разрабатывается с 2000-х годов. На Zeroc ICE оказала большое влияние технология CORBA (Common Object Request Broker Architecture). CORBA – стандарт написания распределённых приложений, развиваемый с 1989 года. Он продвигается консорциумом OMG (Object Management Group), объединившим крупнейшие фирмы, производящие ПО. Создатели Zeroc ICE участвовали в разработке этого стандарта и стали свидетелями всех трудностей, связанных со стандартизацией такой сложной технологии. Спецификации CORBA оказались очень сложными и в ряде случаев оторванными от реальности. Это привело к тому, что некоторая функциональность, предусмотренная в спецификациях, осталась нереализованной, а созданные разными производителями реализации зачастую оказывались несовместимыми. Несмотря на обилие спецификаций, в стандарте CORBA не были учтены некоторые аспекты, необходимые распределенным приложениям. Многопоточные приложения были в стандарте проигнорированы, что делало их непереносимыми и зависимыми от конкретной реализации. Стандарт не предоставлял механизма обратной совместимости между версиями приложения, что делало невозможным обновление серверов независимо от клиентов [12].

Разработчики Zeroc ICE решили создать ПО, которое было бы сравнимо по функциональности с реализациями CORBA, но не обладало описанными выше недостатками. Zeroc ICE является более мощным инструментом, чем Protocol Buffers и Thrift. Помимо реализации RPC, ICE предоставляет ряд сервисов, которые значительно облегчают построение распределенных систем. Рассмотрим сначала основные характеристики ICE.

Для описания интерфейсов модулей и структур данных используется язык Slice (Specification Language for Ice), похожий на IDL Thrift и Protocol Buffers. Разработчик Slice вводит абстракцию, которая называется ICE объект. ICE объект – это сущность в локальном или удаленном адресном пространстве, которая может принимать вызовы клиентов. ICE объект характеризуется уникальным для всей системы идентификатором. Он предоставляет один или несколько интерфейсов, описанных на языке Slice. Интерфейс представляет собой набор именованных операций. Каждый объект имеет главный интерфейс. Кроме того, ICE объект может обладать альтернативными интерфейсами, называемыми фасетами. При вызове удаленного метода клиент должен выбрать либо один из фасетов, либо главный интерфейс. До версии 3.5 фасеты были единственным способом обеспечить обратную совместимость в ICE приложениях. При любых изменениях в ICE объект добавляли новый фасет, с которым могли взаимодействовать новые обновленные клиенты. Старые же клиенты продолжали использовать главный интерфейс.

Сущность на стороне сервера, которая реализует один или несколько ICE объектов, называется сервантом. На практике, сервант – это экземпляр класса, который поставлен в соответствие ICE объекту и реализует его интерфейсы. Сопоставление ICE объекту осуществляется с помощью регистрации серванта на сервере. Несколько сервантов может быть одновременно зарегистрировано на стороне сервера, что делает возможным использовать один сервер для нескольких целей.

В отличие от Thrift, чтобы сделать сервер многопоточным, не нужно изменять его код. ICE предоставляет пул потоков, который при должной настройке будет создавать новый поток для вновь поступившего вызова, если сервер в это время занят.

Для вызова удаленных методов на стороне клиента используется прокси ICE объекта. Прокси является локальным объектом для адресного пространства клиента. Он предоставляет методы, с такими же названиями что и в интерфейсе сервера. С помощью прокси настраивается транспорт, используемый при вызове (TCP, udp), а также синхронность/асинхронность вызовов. Прокси бывают двух видов: прямые и косвенные.

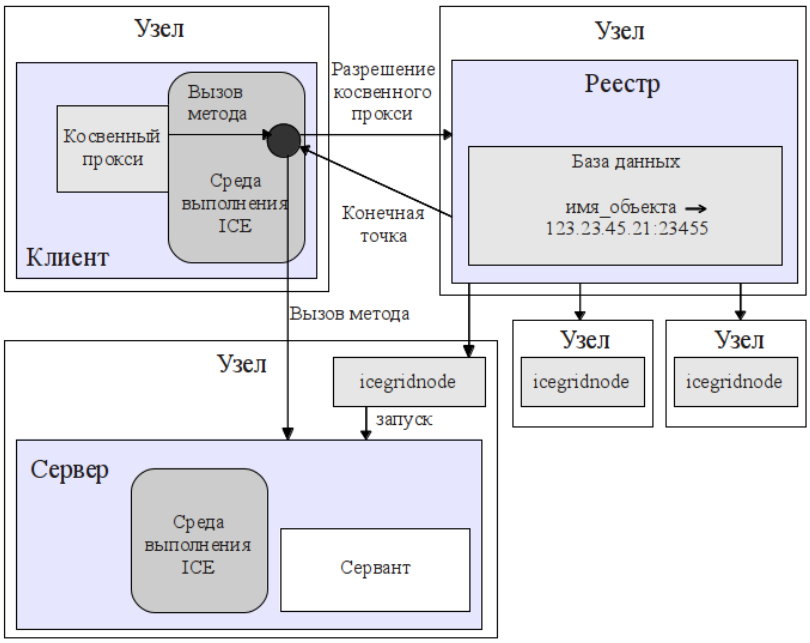

Рис. 2. Схематичное изображение типичного ICEGrid приложения

Прямые прокси содержат адрес сервера, т.е., имя хоста и номер порта. Такой подход не является масштабируемым и делает невозможным изменение местоположения сервера без изменения клиента. Косвенные прокси решают эти проблемы. Вместо адреса сервера такие прокси содержат идентификатор ICE объекта. Для сопоставления идентификатора ICE объекта используется служба имен. Каждый сервер во время запуска регистрирует адреса своих ICE объектов в службе имен. Косвенная адресация добавляет небольшую задержку при первом использовании клиентом прокси из-за необходимости запросить у службы имен адрес сервера. Однако все последующие обращения происходят напрямую между клиентом и сервером, так что накладные расходы незначительны. Использование косвенных прокси в сочетании со службой имен может обеспечить большую функциональность без каких-либо специальных действий со стороны клиентов: репликацию, балансировку нагрузки.

ICE предоставляет реализацию службы имен, называемую ICEGrid [13]. ICEGrid состоит из реестра и произвольного количества узлов. Обычно каждой машине из кластера соответствует один ICEGrid узел (icegridnode). В обязанности узлов входит запуск и мониторинг ICE серверов, запущенных на машине. Необходимую информацию для запуска серверов узлы получают из реестра. Туда она загружается администратором системы либо программно, либо через командную строку. Реестр ведет учет этой информации и сохраняет ее на диск.

Так как реестр является реализацией службы имен, то он отвечает за разрешение косвенных прокси. На Рис. 2 изображена операция разрешения косвенного прокси с помощью ICEGrid реестра.

Реестр не является простой хэш-таблицей для поиска текущих адресов ICE серверов. В случае запроса о местоположении сервера, который является неактивным в данный момент времени, реестр может инициировать запуск этого сервера на узле. С использованием косвенных прокси также организована репликация серверов и балансировка нагрузки между ними. Репликация серверов в ICEGrid реализуется на уровне объектных адаптеров, в которых регистрируется серванты ICE объектов. Объектные адаптеры нескольких серверов могут быть объединены в группу реплик. Когда в реестр приходит запрос на разрешение косвенного прокси, содержащего в качестве имени идентификатор группы реплик, реестр возвращает клиенту адрес одного или нескольких серверов, входящих в группу. Способ выбора этих адресов описывается в конфигурации группы.

Таким образом, с использованием ICEGrid можно создавать масштабируемые системы, в которых клиенты обнаруживают новые добавленные серверы с помощью реестра. В следующем разделе представлен подход, который позволяет создавать слабосвязанные системы и платформу для их построения.

4. WCF

WCF (Windows Communication Foundation) – это платформа для построения распределенных приложений для фреймворка .NET. WCF позиционируется как платформа для создания распределенных приложений, основанных на сервис-ориентированной архитектуре (Service-Oriented Architecture, SOA). Основная идея SOA состоит в использовании распределенных слабосвязанных между собой сервисов. Для реализации принципов SOA WCF использует протокол SOAP (Simple Object Access Protocol), основанный на XML. Такой подход обеспечивает интероперабельность и позволяет создавать крупномасштабные межорганизационные распределенные системы. Однако накладные расходы, связанные с парсингом xml, делают такой подход неприменимым при организации эффективного межкомпонентного взаимодействия. Для этих целей WCF предоставляет бинарный формат сериализации данных и TCP-транспорт.

Функциональность, которую предоставляют сервисы, описывается с помощью языка WSDL (Web Services Description Language), основанного на языке XML. На основании этого описания генерируется код на C# или Visual Basic. Однако среди разработчиков практикуется создание сначала кода сервиса и последующая генерация из него WSDL описания. Частично это связано со сложностью WSDL и его концептуальной отдаленности от привычных программисту интерфейсов классов и структур данных. Сравнение обычного IDL описания с WSDL структурой представлено в [14]. Созданный WSDL файл публикуется обычно на том же сервере, где размещен сервис.

Для взаимодействия клиента с сервисом в WCF используются конечные точки, публикуемые в WSDL файле. На основе них клиенты генерируют прокси, через который можно отправлять сообщения сервису. Конечные точки состоят из трех частей. Первая часть определяет адрес сервиса. Адрес задается с помощью унифицированного идентификатор ресурса. Вторая часть конечной точки называется привязкой. Привязка определяет протокол, транспорт и способ сериализации сообщения. WCF включает заранее определенные привязки для наиболее популярных комбинаций: SOAP сообщения через HTTP, REST, SOAP сообщение через TCP, бинарный протокол через TCP. Также существуют привязки для межпроцессорного взаимодействия через именованные каналы и для службы очереди сообщений. Последняя привязка полезна при коммуникации через сеть, подверженную разрывам соединения. Третья часть называется контрактом. Она определяет интерфейс, предоставляемый сервисом [15].

Большую гибкость добавляет возможность сервиса иметь много конечных точек, доступных в одно и то же время. Это делает возможным предоставить сервис внешним клиентам с помощью привязки HTTP/SOAP для достижения интероперабельности и использовать бинарный протокол через TCP для достижения максимальной производительности.

На основе привязки клиент создает стек каналов, который используется для отправки сообщения. Стек каналов скрывает детали посылки каждого сообщения. Как видно на Рис. 3, он состоит из одного транспортного канала, который всегда является последним каналом в стеке и ответственен за работу с транспортом, определенном в привязке. Предшествовать транспортному каналу могут несколько протокольных каналов, которые отвечают за безопасность и гарантируют надежную доставку сообщения. Обычно это достигается с помощью шифрования проходящих сообщений. Помимо этого, протокольный канал может посылать свои собственные сообщения, например для подтверждения приема [16].

Функциональность сервисов и клиентов может быть расширена с помощью, так называемых поведений. Поведения учитываются в процессе построения стека каналов и могут изменить канал, основанный на определенных пользователем конечных точках. Однако они

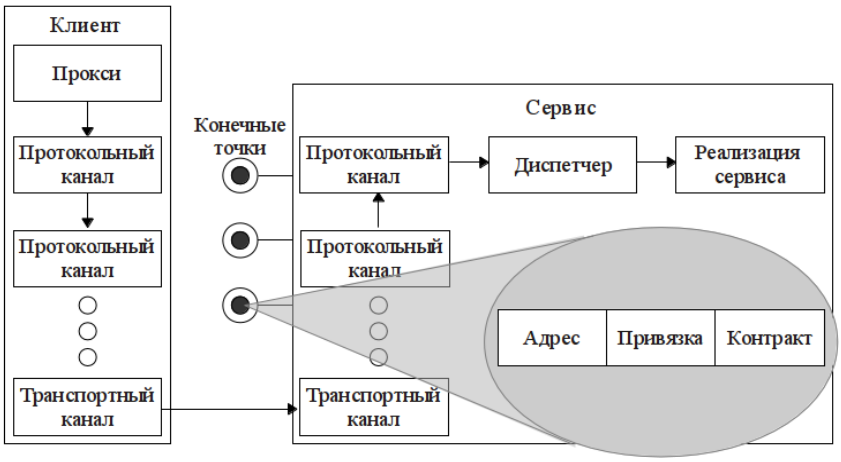

Рис. 3. Взаимодействие клиента и сервиса

привносят только локальные изменения для клиента или сервиса, то есть они не выносятся в WSDL файл. С помощью поведений сервис можно сделать многопоточным. В случае если класс, реализующий сервис, не является потокобезопасным, существует поведение, создающее отдельный экземпляр класса для каждого входящего вызова. Кроме сервисов присутствует возможность настраивать поведение для контрактов, конкретных операций и конечных точек. Поведения можно задавать программно или при конфигурации сервиса. Гибкости WCF сервисам, добавляет возможность реализовать собственное поведение, протокольный канал, для произвольного расширения функциональности.

WCF поддерживает обнаружение сервисов с помощью протокола WS-Discovery. Механизм очень напоминает работу по разрешению косвенных прокси для ICEGrid. Сервисы, доступные для обнаружения, отправляют сообщения для оповещения о своем появлении в сети и об уходе из сети. Для нахождения сервиса клиенты отправляют запрос, содержащий заданные критерии, например тип контракта, или сетевой домен. Службы получают этот запрос и определяют, соответствуют ли они критериям. В случае успеха сервис посылает клиенту конечные точки, необходимые для установки связи [17].

5. Сравнение ProtoBufs, Thrift, ICE и WCF

Из Табл. 1 видно, что Protocol Buffers обладает менее богатыми возможностями для описания структур данных на IDL, чем его конкуренты. В частности, он не поддерживает контейнеры, позволяя только объявить поле структуры данных как “repeated”. Для полей, помеченных этим ключевым словом, будут сгенерированы дополнительные методы для заполнения списка. Такой подход не позволяет создавать методы, принимающие или возвращающие контейнеры.

Также стоит отметить, что до версии 3.5 [18], бета-релиз которой состоялся в декабре 2012 года, в Zeroc ICE нельзя было модернизировать структуры данных, не разрушив совместимость со старыми версиями. В то время как конкуренты поддерживали объявление новых полей как необязательных. Это позволяло в процессе десериализации среде выполнения отбросить незнакомые поля, помеченные как необязательные, сохраняя совместимость. Тогда как разработчику ICE приложений приходилось заботиться о совместимости самостоятельно, создавая новые интерфейсы (фасеты) для взаимодействия с обновленными клиентами. В случае большого количества изменений, код клиентов становился запутанным, а на стороне сервера простаивало большое количество сервантов, реализующих одну и ту же логику, только с разными версиями структур данных.

Как видно из Табл. 2, Thrift обладает меньшей функциональностью по сравнению с двумя другими платформами. В Zeroc ICE эта дополнительная функциональность реализована в виде отдельных сервисов, реализованных с помощью основного ядра платформы. Не

исключено, что с развитием Thrift, появятся сторонние реализации, значительно расширяющие его возможности. Реализация зашифрованных сообщений была добавлена после выхода Thrift усилиями сообщества [19]. Необходимо отметить, что создание многопоточного сервера в Thrift происходит программно. Тогда как в Zeroc Ice или WCF многопоточность сервера можно задать во время конфигурирования приложения. Хуже ситуация у Thrift и с односторонними вызовами, так как разработчик уже на этапе создания IDL должен отметить односторонние методы специальным ключевым словом. В случае с ICE и WCF односторонность вызова задается на уровне прокси и не задействует серверную часть.

Табл. 1. Сравнение платформ

	Protocol Buffers	Apache Thrift	Zeroc Ice	WCF
Поддерживаемые языки	C++, Java, Python	C++, Java, JavaScript, Python, PHP, Ruby, C#, Perl, Objective C, Erlang, Smalltalk, OCaml, Haskell	C++, .NET, Java, Python, Objective-C, Ruby, PHP, ActionScript	C#, visual basic
Форматы передаваемых данных	бинарный, текстовый	бинарный, JSON, текстовый	бинарный	бинарный, xml
Поддерживаемые типы	bool, 32/64-bit, integers, float, double, string, byte sequence	bool, byte, 16/32/64-bit integers, double, string, byte sequence	bool, byte, short, int,long, float, string,	byte, short, int, long, float, double, boolean, char, string, dateTime
Контейнеры	поле “repeated” эмулирует списки	map<t1,t2>, list<t>, set<t>	sequence, dictionary	array, list
Поддержка наследования структур данных	нет	есть	есть	есть
	Protocol Buffers	Apache Thrift	Zeroc Ice	WCF
Исключения	нет	есть	есть	SOAP-сообщение об ошибке
Поддержка компрессии данных	нет	есть	есть	нет
Обратная совместимость	Поддерживается за счет тегирования	Поддерживается за счет тегирования	В версии 3.5 появились опциональные параметры	Поддержка необязательных параметров
Вывод в поток	есть	есть	нет	нет
RPC интерфейсы	есть	есть	есть	есть
RPC реализация	нет	есть	есть	есть
Документация	хорошая	плохая	хорошая	хорошая
Лицензия	BSD	Apache	GPL и проприетарная	проприетарная

Табл. 1. Сравнение RPC реализаций

	Apache Thrift	Zeroc Ice	WCF
Транспорт	TCP	udp, TCP	TCP, http, именованные каналы
Односторонние вызовы	есть	есть	есть
Асинхронные вызовы	есть	есть	есть
Поддержка многопоточных серверов	есть	есть	есть
Безопасная передача данных	SSL	SSL	SSl, WS-SecureConversation
Сервис обнаружения серверов	нет	ICEGrid реестр	WS-Discovery
Обход межсетевого экрана	нет	Glacier2	http

Заключение

В современных поисково-аналитических системах рассмотренные платформы становятся компонентом инфраструктуры средств анализа или хранения больших объемов данных. Protocol buffers – легковесное решение, которое может быть интегрировано в уже существующую систему RPC или систему отложенного обмена сообщениями. Thrift предоставляет свой стек RPC и обладает базовым набором функциональности для построения клиент-серверных распределенных приложений. Он не предоставляет дополнительных сервисов для управления группами компонентов, развернутых на кластере. Такой дополнительной функциональностью обладает Zeroc ICE, платформа поддерживает косвенную адресацию распределенных объектов, что делает возможным группирование объектов, балансировку нагрузки между ними, прозрачное добавление произвольного количества объектов в группу. Эта функциональность делает ICE подходящим инструментом, для организации параллельных вычислений на кластере. WCF хотя и позиционируется как платформа для создания web-сервисов, пригодна и для организации любого вида межпроцессорного взаимодействия: обмен сообщениями между процессами на одной машине через именованный канал, удаленный вызов процедур между модулями развернутыми на узлах кластера с использованием эффективного бинарного протокола, создание web-сервисов для построения межорганизационных распределенных систем.

Интересным развитием RPC платформ может стать Apache Avro, использующийся в проекте Hadoop [20]. Главным его отличием от рассмотренных платформ является генерация на основе IDL описания схемы в формате JSON. Эта схема используется при сериализации и десериализации структур данных в бинарный формат. Наличие машиночитаемой схемы позволяет обойтись без генерации кода для языков программирования с динамической типизацией. Также схема позволяет обойтись без тегирования полей структур данных и параметров методов. Обратная совместимость обеспечивается за счет пометок в схеме. При кажущемся усложнении процесса сериализации, производительность при таком подходе ухудшается незначительно, а по размеру сериализованных данных и обходит конкурентов [0]. Заявлена, но пока не реализована, поддержка http в качестве транспорта, что позволит использовать готовый инструментарий web-серверов для балансировки нагрузки. Подобный подход может оказаться очень гибким и простым в использовании при разработке с применением динамических языков, что может обеспечить высокую пропускную способность и эффективность не только при внутреннем межкомпонентном взаимодействии, но и при взаимодействии через сеть интернет.

В заключении стоит отметить, что различные компании используют разные средства для обеспечения межкомпонентного взаимодействия. Например, Google - Protocol buffers в своей реализации MapReduce, а популярный открытый проект Hadoop, используемый в поисково-аналитических системах yahoo и AOL, использует свое собственное RPC и т.д. Таким образом, выбор платформы в большей степени зависит от целей и задач поисково-аналитической системы, объемов обрабатываемых данных, уровня и опыта разработчиков и ряда других факторов.

Литература

1. Ghosh, D., Sheehy, J., Thorup, K.K., Vinoski, S.: Programming language impact on the development of distributed systems. J. Internet Services and Applications(2012) стр. 23-30.
2. Ebot: an erlang web crawler // [Электронный ресурс] – Режим доступа: http://www.redaelli.org/matteo-blog/2010/05/21/577/ свободный. Проверено 27.01.2013.
3. Riak and Erlang/OTP // [Электронный ресурс] – Режим доступа: http://www.aosabook.org/en/riak.html свободный. Проверено 27.01.2013.
4. RabbitMQ: Features // [Электронный ресурс] – Режим доступа: http://www.rabbitmq.com/documentation.html свободный. Проверено 27.01.2013.
5. Third-Party Add-ons for Protocol Buffers // [Электронный ресурс] – Режим доступа: http://code.google.com/p/protobuf/wiki/ThirdPartyAddOns свободный. Проверено 27.01.2013.
6. Protocol Buffers: Leaky RPC // [Электронный ресурс] – Режим доступа: http://steve.vinoski.net/blog/2008/07/13/protocol-buffers-leaky-rpc/#comment-1093 свободный. Проверено 27.01.2013.
7. Developer Guide // [Электронный ресурс] – Режим доступа: https://developers.google.com/protocol-buffers/docs/overview свободный. Проверено 27.01.2013.

8. Powered by Thrift // [Электронный ресурс] – Режим доступа: http://wiki.apache.org/thrift/PoweredBy свободный. Проверено 27.01.2013.
9. Agarwal, A.; Slee, M. & Kwiatkowski, M. (2007), 'Thrift: Scalable Cross-Language Services Implementation', Technical report, 156 University Ave, Palo Alto, Facebook.
10. Apache Thrift // [Электронный ресурс] – Режим доступа: http://jnb.ociweb.com/jnb/jnbJun2009.html свободный. Проверено 27.01.2013.
11. ICE architecture // [Электронный ресурс] – Режим доступа: http://doc.zeroc.com/display/Ice/Ice+Architecture свободный. Проверено 27.01.2013.
12. Henning M. The Rise and Fall of CORBA // ACM Queue. Vol. 4. No. 5 — June 2006: 28–34.
13. ICE for GRID computing // [Электронный ресурс] – Режим доступа: http://www.zeroc.com/icegrid/ свободный. Проверено 27.01.2013.
14. Differences between Ice and SOAP/Web Services // [Электронный ресурс] – Режим доступа: http://www.zeroc.com/iceVsSoap.html свободный. Проверено 27.01.2013.
15. Introduction to Building Windows Communication Foundation Services // [Электронный ресурс] – Режим доступа: http://msdn.microsoft.com/en-us/library/aa480190.aspx свободный. Проверено 27.01.2013.
16. Channel Model Overview // [Электронный ресурс] – Режим доступа: http://msdn.microsoft.com/en-us/library/ms729840.aspx свободный. Проверено 27.01.2013.
17. Alex Mackey Introducing .NET 4.0 Windows Communication Foundation Apress 2010, стр. 159-173.
18. Optional Data Members // [Электронный ресурс] – Режим доступа: http://doc.zeroc.com/display/Ice35/Optional+Data+Members свободный. Проверено 27.01.2013.
19. TSSLServerSocket and TSSLSocket implementation // [Электронный ресурс] – Режим доступа: https://issues.apache.org/jira/browse/THRIFT-151 свободный. Проверено 27.01.2013.
20. Hadoop: The Definitive Guide, 2nd Edition Tom White O'Reilly Media / Yahoo Press 2010 стр. 110 -142.

jvm-serializers // [Электронный ресурс] – Режим доступа: https://github.com/eishay/jvm-serializers/wiki свободный. Проверено 27.01.2013.

Тихомиров Илья Александрович. Старший научный сотрудник ИСА РАН. В 2002 году окончил Рыбинскую государственную авиационную технологическую академию. Кандидат технических наук. Автор 47 научных работ. Область научных интересов: искусственный интеллект, компьютерная лингвистика, поисковые системы, информационная безопасность, интернет-системы. E-mail: tih@isa.ru

Зубарев Денис Владимирович. Инженер ИСА РАН. Окончил Российский университет дружбы народов в 2012 году. Область научных интересов: искусственный интеллект, поисковые системы, распределенные системы.
E-mail: dvz@isa.ru