Унифицированный интерфейс доступа к алгоритмическим сервисам в Web *

О. В. Сухорослов

Центр Грид-технологий и распределенных вычислений Института системного анализа РАН

В работе описана модель алгоритмического сервиса, которая позволяет унифицировать удаленный доступ к широкому спектру вычислительных приложений, служащих для решения определенных классов научных и прикладных задач. Рассмотрена применимость различных технологий для реализации удаленного доступа к алгоритмическим сервисам. Разработан унифицированный интерфейс доступа к алгоритмическим сервисам на основе веб-технологий и подхода REST. Создана эталонная программная реализация данного интерфейса, демонстрирующая применимость предлагаемого подхода на практике.

Введение

При решении сложных научных задач исследователь часто сталкивается с отсутствием требующихся ему ресурсов на своем компьютере, будь то информационные ресурсы, вычислительные мощности или программное обеспечение. При этом необходимые ресурсы могут быть найдены на серверах, вычислительных системах или компьютерах коллег, распределенных административно и территориально. Современные сетевые технологии позволяют организовать удаленный доступ к различным видам ресурсов с последующим их объединением в рамках распределенных приложений. Данный подход позволяет избежать переноса всех требуемых ресурсов на один компьютер, а также обладает рядом других важных преимуществ.

В настоящей работе рассматривается вопрос об организации удаленного доступа к проблемно-ориентированным вычислительным ресурсам, служащим для решения определенных классов научных и прикладных за-

* Работа выполнена при поддержке фонда РФФИ (№ 08–07–00430–а), Президиума РАН (программа фундаментальных исследований П1) и Совета по грантам Президента Российской Федерации (№ НШ–5511.2008.9).

дач. В качестве методологической основы используется сервис-ориентированный подход, позволяющий реализовать унифицированный механизм доступа к указанному виду ресурсов и обеспечивающий удобство их интеграции в произвольные приложения. В первом разделе работы вводится понятие алгоритмического сервиса, представляющего доступ к ресурсу рассматриваемого типа в распределенной среде, а также обсуждаются характерные особенности данных сервисов.

Для упрощения повторного использования и композиции алгоритмических сервисов в рамках различных приложений требуется унификация механизма удаленного доступа к сервисам на уровне протоколов и форматов данных. Для этого могут быть использованы различные технологии и платформы распределенного программирования. Во втором разделе проводится анализ применимости различных подобных технологий для реализации унифицированного механизма доступа к алгоритмическим сервисам. В качестве базовой платформы для организации удаленного доступа к алгоритмическим сервисам предлагается использовать протоколы и технологии Web, основанные на архитектурном стиле REST (Representational State Transfer). Данные технологии основаны на открытых стандартах (HTTP, URI, XML), имеют большое количество независимых реализаций и поддерживаются всеми современными языками программирования. Это позволяет максимально расширить как круг потенциальных разработчиков, так и пользователей сервисов, обеспечив при этом совместимость различных реализаций и широкое повторное использование сервисов.

Третий раздел работы посвящен непосредственно описанию унифицированного интерфейса доступа к алгоритмическим сервисам на основе технологий Web. В качестве базового протокола доступа к сервисам используется протокол HTTP, а формата данных — JSON (JavaScript Object Notation). Разработанный интерфейс учитывает характерные особенности алгоритмических сервисов, такие как длительная обработка вычислительно сложных запросов и передача больших объемов данных в виде файлов. Также в соответствии с сервис-ориентированным подходом разработанный интерфейс поддерживает интроспекцию, т. е. получение информации о сервисе. Выбранные базовые технологии допускают независимые реализации описанного интерфейса на различных языках программирования. При этом детали реализации сервиса остаются полностью на усмотрение его разработчика.

В четвертом разделе описывается эталонная программная реализация данного интерфейса на языке Java, демонстрирующая применимость предлагаемого подхода на практике. Сервер EveREST позволяет быстро преобразовывать широкий спектр существующих приложений в удаленно доступные алгоритмические сервисы. В частности, поддерживается интеграция приложений с интерфейсом командной строки, приложений на языке

Java, а также грид-приложений, запускаемых в инфраструктуре EGEE. Важной дополнительной функцией сервера является генерация веб-интерфейсов, позволяющих пользователям обращаться к размещенным сервисам через веб-браузер.

В заключении формулируются основные результаты работы, обсуждаются аналогичные разработки и намечаются пути дальнейших исследований.

1. Модель алгоритмического сервиса

Алгоритмический сервис представляет собой доступный по сети программный компонент, поддерживающий решение определенного класса задач с помощью соответствующих вычислительных алгоритмов. В соответствии с моделью клиент-сервер, сервис обслуживает приходящие к нему запросы клиентов на решение конкретных задач. Запрос клиента содержит параметризованное описание задачи, формулируемое в виде конечного набора входных параметров. После успешной обработки запроса сервис возвращает клиенту результат, оформленный в виде конечного набора выходных параметров.

Для взаимодействия с сервисом клиенту необходимо знать список входных и выходных параметров сервиса. Данная информация является частью описания сервиса, публикуемого в общедоступном месте или предоставляемого сервисом по запросу клиента. Описание параметров сервиса определяет форматы сообщений (контракт), которым должны следовать клиент и сервис. Данное описание должно поддерживать машинную интерпретацию, например для валидации сообщений, генерации клиентского кода или реализации динамических вызовов.

Отметим, что описанная модель является довольно общей и может быть, вообще говоря, применена к большому классу сервисов, не обязательно предоставляющих доступ к вычислительным алгоритмам. В частности, это могут быть сервисы доступа к базам данных или сервисы, осуществляющие обработку и преобразование данных. В общем случае, речь может идти о произвольном программном компоненте, оформленном в виде сервиса. Однако в рамках данной работы рассматриваются в первую очередь алгоритмические сервисы, нацеленные на решение научных и прикладных вычислительных задач. Рассмотрим характерные особенности данных сервисов.

Во-первых, каждый запрос обрабатывается сервисом независимо от других запросов. Иными словами, результат запроса определяется исключительно значениями передаваемых клиентом входных параметров и не зависит от результатов других запросов. Фактически данная особенность играет роль ограничения, накладываемого на рассматриваемые сервисы.

А именно, за рамками описываемой модели сервиса остаются клиент-серверные приложения, в которых требуется поддерживать состояние (сессию) между последовательными запросами клиента к серверу. Примером такого приложения может являться удаленная работа в интерактивном режиме с пакетом MATLAB, когда пользователь передает пакету команды, оперирующие с уже хранящимися на сервере результатами предыдущих команд. Исключение из рассмотрения подобных случаев позволяет существенно упростить интерфейс и реализацию сервисов, а также, что самое главное, повысить масштабируемость и отказоустойчивость приложений в целом. Отметим, что данное ограничение часто накладывается на сервисы в рамках сервис-ориентированной архитектуры.

Вторая важная особенность алгоритмических сервисов заключается в том, что обработка запросов клиентов (решение задач) часто сопряжена с проведением длительных вычислений и может сопровождаться запуском вычислительных заданий на многопроцессорных вычислительных комплексах и в грид. В этом случае сервис не может сразу вернуть клиенту ответ на его запрос, и обработка подобных запросов должна вестись в асинхронном режиме. В ответ на вызов клиенту может возвращаться идентификатор запроса, используя который клиент может опрашивать статус запроса (т. н. режим pull) и получить готовый результат. Клиент также может получать от сервиса уведомления об изменении статуса его запроса (режим push). Стоит отметить, что поддержка асинхронной обработки запросов неизбежно приводит к появлению внутреннего состояния на стороне сервиса вследствие того, что запрос и получение его результата реализуется в виде нескольких вызовов сервиса. Однако данное состояние существует исключительно в контексте одного конкретного запроса к сервису.

В-третьих, доступное клиентам описание сервиса должно содержать информацию не только о параметрах сервиса, но и о классе решаемых сервисом задач, используемых при этом алгоритмах и численных методах, точности получаемых решений и т. п. Формальное описание данной информации в машинно-интерпретируемом виде является сложной задачей, имеющей специфические требования для конкретных прикладных областей. В общем случае данная информация может предоставляться в произвольной форме, доступной для просмотра пользователем.

В-четвертых, существует много алгоритмических сервисов, принимающих на вход или генерирующих большие объемы данных. Для подобных сервисов необходимо организовать эффективную передачу параметров большого размера в виде файлов с использованием соответствующих механизмов передачи данных по сети. В этом случае запрос к сервису или возвращаемый сервисом результат запроса могут содержать не сами значения параметров, а ссылки на соответствующие файлы.

2. Выбор базовых технологий

Описанная модель алгоритмического сервиса может быть реализована с использованием различных платформ и технологий распределенного программирования, в том числе Remote Procedure Call (RPC), объектно-ориентированного промежуточного ПО (ППО), WS-сервисов (веб-сервисов на основе протокола SOAP и спецификаций WS-*) и REST-сервисов (веб-сервисов на основе протокола HTTP и архитектурного стиля REST). Выбор той или иной технологии зависит от ряда факторов, таких как распространенность, удобство разработки и открытость исходного кода. В рамках научной сервис-ориентированной среды, допускающей участие различных лиц и организаций, очень важно наличие открытых стандартов и множества независимых реализаций базовой технологии. Важно также, чтобы используемая технология как можно ближе соответствовала описанной модели сервиса. Рассмотрим данные вопросы подробнее.

Удаленный вызов процедуры в рамках технологии RPC идеологически хорошо укладывается в модель алгоритмического сервиса. Однако у RPC есть ряд недостатков. Традиционная модель RPC, описанная в стандарте OSF-DCE, не поддерживает асинхронные вызовы и обработку запросов. Многие из поддерживаемых сейчас реализаций RPC являются проприетарными и имеют ограниченную совместимость. В целом, RPC является низкоуровневой технологией, применяемой, главным образом, для внутрисистемного взаимодействия и плохо приспособленной для создания внешних сервис-ориентированных интерфейсов.

Объектно-ориентированное ППО, как развитие RPC на основе объектно-ориентированного подхода, представлено такими технологиями, как CORBA,.NET Remoting, Java RMI, ZeroC Ice, Apache Thrift, Cisco Etch. Единственным открытым стандартом, имеющим несколько реализаций, из перечисленного списка является CORBA. Общепризнанным фактом является чрезмерная сложность спецификаций CORBA и, как следствие, освоения данной технологии. В начале 2000-х годов интерес к CORBA пошел на спад, и развитие технологии замедлилось. Ряд производителей отказался от дальнейшей поддержки своих реализаций CORBA. Наиболее развитой из остальных упомянутых технологий является ППО Ice, разрабатываемое компанией ZeroC, Inc. с начала 2000-х годов. Фактически, Ice является попыткой реализовать ППО, сравнимое по своим возможностям с CORBA, но лишенное при этом упомянутых недостатков. Технология Ice поддерживает разработку приложений на многих популярных языках и доступна на всех распространенных разновидностях операционных систем. Ice распространяется вместе с исходным кодом под лицензией GNU General Public License (GPL). Пожалуй, единственным, но весьма важным недостатком Ice является то, что данная технология не основана на обще-

признанном стандарте и является проприетарной разработкой, доступной в единственной реализации. Этот факт затрудняет использование данной технологии в открытой для множества участников сервис-ориентированной среде.

Отметим еще одно обстоятельство, которое не благоприятствует использованию объектно-ориентированного ППО для реализации модели алгоритмических сервисов. В соответствии с данной моделью сервис изначально не имеет внутреннего состояния и операций, т. е. не является объектом. Более подходящими понятиями здесь являются функция или процедура. При вызове алгоритмического сервиса клиент не передает имя операции, а результат запроса не зависит от предыдущих запросов к сервису этого или других клиентов. В то же время объектно-ориентированное ППО поощряет создание удаленных объектов, предоставляющих несколько операций и имеющих внутреннее состояние. Справедливости ради стоит отметить, что поддержка асинхронной обработки запросов неизбежно приводит к появлению внутреннего состояния на стороне сервиса. Однако данное состояние существует исключительно в контексте одного конкретного запроса к сервису.

Нетрудно также заметить, что удаленные объекты, предоставляющие доступ к алгоритмическим сервисам, могут иметь одинаковые интерфейсы, с точностью до форматов входных и выходных параметров сервиса. В этом случае не требуется вся мощь присутствующего в объектно-ориентированном ППО языка описания интерфейсов (IDL), а требуется только средство описания форматов входных и выходных параметров сервиса. Таким образом, объектно-ориентированное ППО можно использовать для реализации алгоритмических сервисов при соблюдении определенных ограничений, приводящих в итоге к унифицированному интерфейсу и схеме описания форматов запросов и ответов.

Веб-сервисы на основе протокола SOAP и спецификаций WS-*, которые будем далее называть WS-сервисами, имеют много общего с объектно-ориентированным ППО. В первую очередь это касается наличия многочисленных спецификаций, сложности освоения и сопряжения реализаций различных компаний-разработчиков. Однако есть ряд важных различий. Спецификации WS-сервисов являются открытыми стандартами, изначально ориентированными на реализацию сервис-ориентированной архитектуры. В основе WS-сервисов лежат общепризнанные стандарты — протокол HTTP и формат XML. Заметим, что это также приводит к тому, что WS-сервисы проигрывают в производительности технологиям объектно-ориентированного ППО, использующим двоичный протокол. Описание форматов запросов и ответов к WS-сервису проводится на языке XML Schema, отдельно от описания интерфейса WS-сервиса, осуществляемого на языке WSDL. Архитектура WS-сервисов поощряет создание

сервисов без внутреннего состояния. Тем не менее, существует способ управления внутренним состоянием (спецификации WS-Resource Framework), в том числе для поддержки асинхронной обработки запросов. Наконец, что немаловажно, данные технологии активно поддерживаются крупными компаниями-разработчиками, что привело к повсеместному их распространению и появлению множества реализаций, в том числе и с открытым кодом.

Распространенной критикой WS-сервисов является их чрезмерная сложность и некорректное использование протокола HTTP, что ставит под сомнение приставку «Web-» в их названии. При этом преимущества от использования WS-сервисов заметны только в определенном классе приложений, ориентированных на поддержку сложных бизнес-процессов, встречающихся в корпоративных и государственных системах и слабо распространенных в технических и научных проектах. Это привело к появлению альтернативных, более простых в использовании подходов к реализации веб-сервисов, основанных на прямом использовании протокола HTTP. Первый подход, фактически реализующий парадигму RPC непосредственно поверх HTTP (например, XML-RPC), хотя и упрощает реализацию веб-сервисов, но так же как WS-сервисы игнорирует архитектурные принципы Web и протокола HTTP. Второй подход основан на применении архитектурного стиля REST, лежащего в основе архитектуры Web.

В 2000 году Рой Филдинг, один из главных разработчиков протокола HTTP и других спецификаций Web, опубликовал диссертацию [1] с описанием эталонной модели архитектуры Web. Данная модель или архитектурный стиль, получивший название Representational State Transfer (REST), содержит ряд ключевых принципов или ограничений. Центральными понятиями REST являются понятия ресурса, идентификатора и представления ресурса. В рамках REST вводятся следующие ограничения на архитектуру системы: клиент-серверная архитектура, отсутствие внутреннего состояния, кэширование ответов на запросы, унифицированный интерфейс доступа к ресурсам, многоуровневая архитектура, динамическая загрузка кода. Данные ограничения позволяют обеспечить желаемые свойства архитектуры Web, такие как масштабируемость, расширяемость и открытость. Отметим, что стиль REST носит общий характер и может быть применен при реализации других распределенных систем.

В качестве базовой платформы для организации удаленного доступа к алгоритмическим сервисам предлагается использовать протоколы и технологии Web, основанные на архитектурном стиле REST. Благодаря унифицированному интерфейсу доступа к ресурсам, использованию открытых стандартов (HTTP, URI) и наличию множества проверенных временем реализаций (веб-серверы, библиотеки для работы с HTTP для всех совре-

менных языков программирования), REST обеспечивает максимальную свободу для независимой разработки веб-сервисов и соответствующих клиентских приложений [2]. Немаловажным обстоятельством является простота разработки REST-сервисов в сравнении с другими рассмотренными технологиями. Все это позволяет максимально расширить как круг потенциальных разработчиков, так и пользователей сервисов, обеспечив при этом совместимость различных реализаций и широкое повторное использование сервисов

Практическая реализуемость модели алгоритмического сервиса на принципах REST рассматривается далее.

3. Интерфейс доступа к алгоритмическим сервисам

В данном разделе описывается унифицированный интерфейс доступа к алгоритмическим ресурсам на основе технологий Web и архитектурного стиля REST. Перечислим основные требования, предъявляемые к данному интерфейсу в соответствии с моделью алгоритмического сервиса:

- поддержка интроспекции, т. е. получения описания сервиса по запросу клиента;
- поддержка асинхронной обработки запросов, требующих длительных вычислений;
- поддержка передачи параметров запроса и результата в виде файлов.

Дополнительными требованиями являются следование спецификации HTTP и принципам REST, а также использование распространенных форматов представления данных, таких как XML и JSON.

3.1. Общая схема интерфейса

В соответствии с принципами REST, интерфейс алгоритмического сервиса образован совокупностью идентифицируемых при помощи URI ресурсов, поддерживающих стандартные методы протокола HTTP (табл. 1). Данными ресурсами являются:

- сервис, идентифицируемый с помощью SERVICE_URI;
- запрос к сервису, идентифицируемый с помощью REQUEST_URI;
- файл результата запроса, идентифицируемый с помощью FILE_URI;
- сервер, идентифицируемый с помощью SERVER_URI (является необязательным ресурсом).

Ресурс-сервис поддерживает два метода HTTP. Метод GET возвращает клиенту представление данного ресурса, содержащее описание сервиса. Метод POST служит для отправки сервису нового запроса. В теле запроса клиент передает набор значений входных параметров задачи. В ответ сер-

Таблица 1

Ресурсы и методы интерфейса алгоритмического сервиса

Ресурс / URI	GET	POST	DELETE
Сервис SERVICE_URI	Получение описания сервиса	Запрос к сервису	—
Запрос REQUEST_URI	Получение статуса и результатов запроса	—	Отмена запроса, удаление результатов запроса
Файл результата запроса FILE_URI	Загрузка файла	—	—
Сервер SERVER_URI	Получение списка сервисов, размещенных на сервере	—	—

вис создает новый ресурс-запрос, являющийся подчиненным по отношению к ресурсу-сервису, и возвращает идентификатор ресурса-запроса и его текущее представление клиенту.

Ресурс-запрос поддерживает методы GET и DELETE. Метод GET возвращает представление данного ресурса, содержащее информацию о текущем статусе запроса. Данная информация должна обязательно включать состояние запроса, которое может принимать следующие значения:

- WAITING — выполнение запроса еще не началось;
- RUNNING — запрос выполняется;
- DONE — запрос выполнен успешно;
- FAILED — выполнение запроса завершилось ошибкой.

Если выполнение запроса завершено успешно (состояние DONE), то в представление ресурса-запроса также включается результат запроса в виде набора значений выходных параметров. Часть значений выходных параметров может содержать идентификаторы ресурсов-файлов. Метод DELETE ресурса-запроса позволяет клиенту отменить выполнение запроса или, если выполнение уже завершено, удалить результаты запроса. После вызова DELETE данный ресурс-запрос, а также подчиненные ему ресурсы-файлы, перестают существовать.

Ресурс-файл представляет собой часть результата запроса и поддерживает метод GET для получения содержимого файла клиентом.

Дополнительный ресурс-сервер предназначен для случаев, когда в рамках одного HTTP-сервера размещено несколько алгоритмических сервисов. В этих случаях может быть полезным получение списка сервисов,

размещенных на сервере. Для этих целей зарезервирован метод GET. Ресурс-сервер в данном случае является родительским по отношению к ресурсам-сервисам.

Отметим, что в рамках описанной схемы интерфейса не предписываются конкретные шаблоны для URI ресурсов, которые могут варьироваться между реализациями. При построении URI рекомендуется соблюдать указанные иерархические отношения между ресурсами.

Описанный интерфейс поддерживает обработку запросов как в синхронном, так и асинхронном режиме. Действительно, если результат запроса может быть сразу возвращен клиенту, то он передается внутри возвращаемого в ответ на запрос представления ресурса-запроса с указанием состояния DONE. Если же для обработки запроса требуется время, то это указывается внутри возвращаемого клиенту представления ресурса-запроса с помощью указания соответствующего состояния запроса (WAITING или RUNNING). В этом случае клиент использует переданный URI ресурса-запроса для дальнейшего опроса состояния запроса и получения результата.

Подобная схема, с одной стороны, обеспечивает максимальную свободу сервиса в выборе стратегии обработки каждого отдельного запроса. С другой стороны, при реализации клиента требуется учесть оба сценария развития событий. Почему одни запросы могут быть обработаны быстро, а другие нет? Это может быть связано как с сильной зависимостью времени выполнения алгоритма от значений входных параметров, так и просто с текущей загрузкой сервиса входящими запросами и их возможной приоритетизацией. В общем случае трудно априори предсказать режим обработки запроса и разделить запросы на синхронные и асинхронные. Исходя из этих соображений и была выбрана данная схема.

Для сервисов, результат которых однозначно определяется входными параметрами (небольшого размера) и вычисляется быстро, можно было бы предусмотреть отправку запроса с помощью метода GET с передачей входных параметров в query-параметрах URI. Использование метода GET для подобных случаев предпочтительно из-за возможности кэширования результатов запросов и использования более агрессивной стратегии обработки отказов. Однако наличие двух способов отправки запроса усложнило бы реализации клиентов и сервисов. Кроме того, как было указано выше, в некоторых случаях трудно гарантировать быструю обработку всех возможных запросов к сервису. Что касается кэширования результатов запросов, то оно может быть реализовано на стороне клиента. Большая мощность множеств возможных запросов и их результатов, характерная для алгоритмических сервисов, приводит к тому, что разные клиенты редко будут отправлять одинаковые запросы, и кэширование их на промежуточных серверах окажется малоэффективным.

3.2. Форматы представления данных

За рамками описанной выше схемы интерфейса алгоритмического сервиса остался вопрос о том, какие форматы представления данных будут использовать во время взаимодействия клиент и сервис. Протокол HTTP поддерживает указание формата запроса с помощью заголовка «Content-Type», а также динамическое согласование формата ответа с помощью заголовка «Accept». Значениями указанных заголовков являются идентификаторы MIME-типов, регистрируемых организацией IANA. Данные возможности протокола HTTP позволяют поддерживать в рамках описанного интерфейса сразу несколько форматов представления данных, используемых в зависимости от типа клиента или других обстоятельств.

Наиболее распространенными форматами представления данных в Web являются HTML, XML и JSON. Первый формат используется, главным образом, для отображения данных пользователю через веб-браузер. Два других формата используются при реализации программных интерфейсов к веб-сервисам. Наибольшее распространение получил XML, как универсальный формат обмена данными между программными системами. В последние несколько лет набирает популярность формат JSON, который является компактной альтернативой XML и хорошо сочетается с языком JavaScript, на котором пишется большинство клиентов популярных веб-сервисов (не путать с WS-сервисами).

В качестве основного формата представления данных в интерфейсе алгоритмического сервиса предлагается использовать JSON, исходя из следующих соображений:

- более компактное представление структур данных, в то время как XML ориентирован на представление произвольных документов;
- число библиотек для работы с JSON на различных языках программирования приближается к таковому для XML;
- удобство работы с JSON-данными на языке JavaScript.

Пожалуй единственным недостатком JSON является отсутствие стандартного средства описания схем, сопоставимого с XML Schema, и готовых реализаций валидаторов. Впрочем, уже существует прототип подобного языка — JSON Schema [3].

В качестве дополнительных форматов представления данных в интерфейсе алгоритмического сервиса могут использоваться XML и HTML. Последний может использоваться для поддержки работы пользователей с сервисом через веб-браузер. Конкретные представления запросов и ответов в данном случае выходят за рамки спецификации программного интерфейса сервиса и определяются разработчиком сервиса. Что касается XML, то представления запросов и ответов для данного формата в насто-

Таблица 2

Возможные форматы представления данных для ресурсов и методов интерфейса алгоритмического сервиса

Ресурс / URI	GET	POST	DELETE
Сервис SERVICE_URI	Accept: aplication/json application/xml text/html	Content-Type: application/json application/xml multipart/form-data Accept: aplication/json application/xml text/html	—
Запрос REQUEST_URI	Accept: aplication/json application/xml text/html	—	—
Файл результата запроса FILE_URI	Content-Type: соотв. типу файла	—	—
Сервер SERVER_URI	Accept: aplication/json application/xml text/html	—	—

ящее время не проработаны. При дальнейшем детальном описании интерфейса будет использоваться только формат JSON.

В табл. 2 указаны возможные форматы представления запроса и ответа для каждого ресурса и метода интерфейса алгоритмического сервиса. Несколько замечаний:

- описания сервисов в форматах JSON и XML должны включать описания схем для входных и выходных параметров сервиса на соответствующих языках (JSON Schema или XML Schema);
- при передаче запроса к сервису через форму в веб-браузере используется стандартный формат «multipart/form-data»;
- формат представления ресурса-файла определяется типом (расширением) файла.

3.3. Детальное описание интерфейса

В настоящем разделе приводится детальное описание запросов и возможных ответов для интерфейса алгоритмического сервиса. В качестве формата представления данных используется JSON. Отметим, что данное описание носит предварительный характер и может изменяться в будущем на основе опыта практического применения.

3.3.1. Получение описания сервиса

Формат запроса

```
GET SERVICE_URI
Accept: application/json
```

Формат ответа

Если сервис с данным URI существует, то сервер возвращает описание сервиса в следующем виде:

```
200 OK
Content-Type: application/json

{
   "name":"SERVICE_NAME",
   "description":"SERVICE_DESCRIPTION",
   "inputs": {
      "IN_PARAM1_NAME":{"type":"...","title":"...",...},
      "IN_PARAM2_NAME":{"type":"...","title":"...",...},
   …
   },
   "outputs": {
      "OUT_PARAM1_NAME":{"type":"...","title":"...",...},
      "OUT_PARAM2_NAME":{"type":"...","title":"...",...},
      …
   }
}
```

Поясним назначение отдельных атрибутов описания сервиса:

- атрибут ***name*** содержит краткое имя сервиса;
- атрибут ***description*** содержит краткое текстовое описание сервиса и/или URI документа с подробным описанием сервиса;
- атрибут ***inputs*** содержит массив описаний входных параметров сервиса;
- атрибут ***outputs*** содержит массив описаний выходных параметров сервиса.

Описание параметра состоит из имени параметра и его схемы, оформленной в соответствии с JSON Schema. Наиболее важными атрибутами схемы являются:

- атрибут ***type***, указывающий тип значения параметра;
- атрибут ***title***, содержащий краткое текстовое описание параметра;
- атрибут ***description***, содержащий полное текстовое описание параметра;
- атрибут ***optional*** (применим только для входных параметров), указывающий на то, является ли данный параметр обязательным (значение false) или нет (значение true);
- атрибут ***default***, содержащий значение параметра по умолчанию.

Приведем пример описания тестового сервиса Echo:

```
{
  "name":"echo",
  "description":"Test service echoes back input params",
  "inputs":{
    "input1":{"type":"integer","title":"Integer input"},
    "input2":{"type":"string","title":"String input","optional":true},
    "input3":{"type":"file","title":"File input","optional":true}
  },
    "outputs":{
    "output1":{"type":"integer","title":"Integer output"},
    "output2":{"type":"string","title":"String output"},
    "output3":{"type":"file","title":"File output"}
  }
}
```

В приведенном выше описании используется специальный тип параметра file, отсутствующий в JSON Schema. Данный тип предназначен для передачи значения параметра в виде отдельного файла.

Коды ошибок

Предусмотрены следующие коды ошибок:

- 404 Not Found — сервис с данным URI не существует;
- 500 Internal Server Error — ошибка на серверной стороне.

3.3.2. Запрос к сервису

Формат запроса

```
POST SERVICE_URI
Content-Type: application/json

{
  "IN_PARAM1_NAME":IN_PARAM1_VALUE,
  "IN_PARAM2_NAME":IN_PARAM2_VALUE,
  …
}
```

В запросе должны быть указаны значения всех обязательных входных параметров сервиса.

Приведем пример запроса к тестовому сервису Echo:

```
POST http://somehost.com/echo
Content-Type: application/json

{
  "input1":1637673,
  "input2":"some string",
  "input3":{"uri":"http://otherhost.com/files/test.bin"}
}
```

В приведенном выше запросе для файлового параметра input3 указывается URI удаленного файла. В данном случае перед началом выполнения запроса сервис производит загрузку содержимого файла с помощью протокола HTTP. Данный подход позволяет организовать передачу данных большого объема вне тела запроса. Предусмотрен также вариант передачи содержимого файла непосредственно внутри запроса в кодировке Base64:

```
POST http://somehost.com/echo
Content-Type: application/json
{
   ...
   "input3":{"data":"BASE64_ENCODED_FILE_DATA"}
}
```

В будущем описанная схема передачи файловых параметров может быть расширена путем добавления других протоколов доступа к удаленным файлам, таких как FTP/GridFTP.

Формат ответа

В случае если запрос клиента принят к выполнению, то сервис возвращает следующий ответ:

```
202 Accepted
Location: REQUEST_URI
Content-Type: application/json

Текущий статус запроса в формате JSON (см. п. 3.3.3)
```

Статус 202 Accepted означает, что запрос принят к выполнению и его статус можно отслеживать с помощью нового ресурса-запроса, URI которого содержится в заголовке Location. В теле ответа передается текущее представление ресурса-запроса. Это позволяет клиенту сразу получить результат запроса в случае, если запрос может быть выполнен сервисом в течение короткого времени.

Приведем пример ответа на запрос к тестовому сервису Echo:

```
202 Accepted
Location: http://somehost.com/echo/job2706049717
Content-Type: application/json

{
   "state":"WAITING"
}
```

В данном случае запрос еще не выполнен и находится в состоянии WAITING.

Коды ошибок

Предусмотрены следующие коды ошибок:

- 404 Not Found — сервис с данным URI не существует;
- 400 Bad Request — некорректный запрос (в запросе не указаны значения всех обязательных входных параметров, значение параметра не соответствует его схеме и т. п.);
- 500 Internal Server Error — ошибка на серверной стороне.

3.3.3. Получение статуса и результатов запроса

Формат запроса

```
GET REQUEST_URI
Accept: application/json
```

Формат ответа

В случае если запрос с данным URI существует и находится в состояниях WAITING или RUNNING, то сервис возвращает ответ следующего вида:

```
200 OK
Content-Type: application/json

{
   "state":"REQUEST_STATE",
   "info":"REQUEST_STATUS_INFO",
}
```

В случае если запрос находится в состоянии DONE, то добавляется атрибут result:

```
200 OK
Content-Type: application/json

{
   "state":"DONE",
   "info":"REQUEST_STATUS_INFO",
   "result": {
      "OUT_PARAM1_NAME":OUT_PARAM1_VALUE,
      "OUT_PARAM2_NAME":OUT_PARAM2_VALUE,
      …
   }
}
```

Аналогично, если запрос находится в состоянии FAILED, то добавляется атрибут error:

```
200 OK
Content-Type: application/json

{
   "state":"FAILED",
   "info":"REQUEST_STATUS_INFO",
   "error":"ERROR_INFO"
}
```

Поясним значение каждого из используемых атрибутов:

- обязательный атрибут state содержит текущее состояние запроса, которое может принимать одно из следующих значений: WAITING, RUNNING, DONE, FAILED;
- необязательный атрибут info предназначен для передачи дополнительной информации о статусе выполнения запроса, отображаемой пользователю;
- атрибут result содержит результат выполнения запроса в виде набора значений выходных параметров;
- атрибут error содержит информацию об ошибке, приведшей к сбою при выполнении запроса.

Приведем пример ответа для тестового сервиса Echo в случае, когда выполнение запроса завершено успешно:

```
200 OK
Content-Type: application/json

{
   "state":"DONE",
   "result": {
      "output1":1637673,
      "output2":"some string",
      "output3":{"uri":"http://somehost.com/echo/job2706049717/out.bin"}
   }
}
```

На приведенном выше примере видно, что файловые выходные параметры, аналогично входным параметрам, передаются при помощи указания URI файла. В данном случае клиенту передается адрес файла, соответствующего выходному параметру output3. Данный подход позволяет организовать передачу результатов большого объема. Описанная схема передачи файловых параметров может быть расширена путем добавления других протоколов доступа к удаленным файлам.

Предусмотрен также вариант передачи содержимого выходного файла в кодировке Base64 непосредственно внутри тела ответа. Для этого к REQUEST_URI необходимо добавить query-параметр data:

```
GET REQUEST_URI?data
Accept: application/json
```

Ответ сервиса в данном случае выглядит следующим образом:

```
200 OK
Content-Type: application/json

{
   "state":"DONE",
   "result": {
      ...
      "output3":{"data":"BASE64_ENCODED_FILE_DATA"}
   }
}
```

Коды ошибок

Предусмотрены следующие коды ошибок:

- 404 Not Found — запрос с данным URI не существует;
- 500 Internal Server Error — ошибка на серверной стороне.

3.3.4. Отмена выполнения запроса и удаление результатов

Клиент может отменить выполнение запроса с помощью метода DELETE ресурса-запроса. В случае если запрос уже выполнен успешно, происходит удаление всех данных, связанных с результатом запроса. Таким образом, клиент дает знать сервису, что больше не нуждается в этих данных.

Формат запроса

```
DELETE REQUEST_URI
```

Формат ответа

В случае если запрос с данным URI существует и успешно удален, то сервис возвращает ответ

```
200 OK
```

Коды ошибок

Предусмотрены следующие коды ошибок:

- 404 Not Found — запрос с данным URI не существует;
- 500 Internal Server Error — ошибка на серверной стороне.

3.3.5. Получение файла результата запроса

После успешного выполнения запроса каждый выходной параметр типа file размещается сервисом под некоторым FILE_URI, который передается клиенту в результатах запроса. Клиент может загрузить файл с помощью метода GET ресурса-файла.

Формат запроса

```
GET FILE_URI
```

Формат ответа

В случае если файл с данным URI существует, то сервис возвращает содержимое файла в теле ответа:

```
200 OK
Content-Type: FILE_MIME_TYPE

FILE_DATA
```

Значение заголовка Content-Type соответствует типу файла и определяется, например, исходя из расширения файла.

Коды ошибок

Предусмотрены следующие коды ошибок:

- 404 Not Found — файл с данным URI не существует;
- 500 Internal Server Error — ошибка на серверной стороне.

3.3.6. Получение списка сервисов, размещенных на сервере

Формат запроса

```
GET SERVER_URI
```

Формат ответа

Сервер возвращает список размещенных сервисов с указанием URI каждого сервиса:

```
200 OK
Content-Type: application/json

{
   "SERVICE1_NAME":"SERVICE1_URI",
   "SERVICE2_NAME":"SERVICE2_URI",
   ...
}
```

Коды ошибок

Предусмотрены следующие коды ошибок:

- 404 Not Found — сервер с данным URI не существует;
- 500 Internal Server Error — ошибка на серверной стороне.

4. Программная реализация

Для демонстрации реализуемости предлагаемого подхода на практике создана программная реализация на языке Java. Разработанный сервер EveREST является контейнером алгоритмических сервисов, позволяющим быстро преобразовывать широкий спектр существующих приложений в удаленно доступные сервисы с описанным выше интерфейсом. EveREST реализован на базе библиотеки Jersey [4], являющейся в свою очередь открытой эталонной реализацией спецификации JAX-RS (Java API for RESTful Web Services) [5].

На рис. 1 изображена архитектура сервера EveREST. Взаимодействие с клиентами осуществляется с помощью встроенного веб-сервера Jetty. Входящие HTTP-запросы передаются библиотеке Jersey и, затем, реализации EveREST. Сопряжение между Jersey и EveREST осуществляется с помощью четырех Java-классов (ServerResource, ServiceResource, JobResource

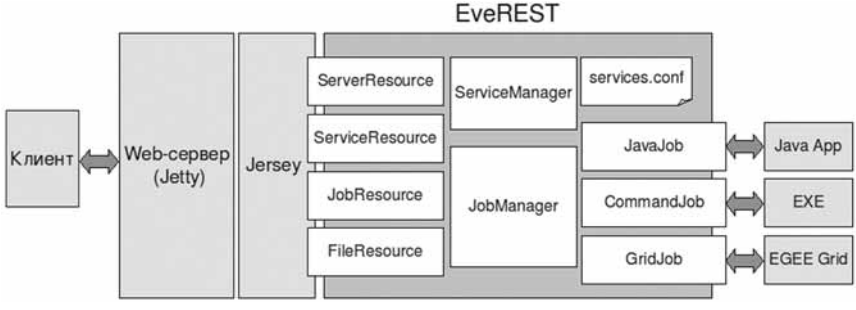

Рис. 1. Архитектура EveREST

и FileResource), которые являются реализациями соответствующих ресурсов из унифицированного интерфейса алгоритмического сервиса.

EveREST осуществляет обработку запросов клиентов в соответствии с конфигурационной информацией. Компонент ServiceManager предоставляет доступ к списку сервисов, размещенных на сервере, и их конфигурации. Данная информация считывается при запуске сервера из конфигурационного файла services.conf. Конфигурация каждого сервиса состоит из двух частей:

- внешнее описание сервиса в соответствии с интерфейсом алгоритмического сервиса (текстовая аннотация сервиса, описания входных и выходных параметров);
- внутренняя конфигурация сервиса, содержащая информацию о том, какой компонент реализует сервис и каким образом происходит передача параметров между сервером и реализацией сервиса.

Текущая версия EveREST поддерживает три типа реализации сервиса:

- Java-класс, реализующий интерфейс everest.java.JavaServiceI;
- приложение с интерфейсом командной строки;
- грид-задание, запускаемое в инфраструктуре EGEE.

В первом случае в конфигурации сервиса требуется указать только имя соответствующего Java-класса. Во втором необходимо указать запускаемую команду в специальном формате, содержащем информацию о том, каким образом параметры сервиса отображаются в аргументы команды и внешние файлы. В третьем случае требуется указать путь к файлу с JDL-описанием грид-задания, а также информацию об отображении параметров сервиса в аргументы и файлы грид-задания.

Отметим, что два последних варианта позволяют использовать в качестве реализации сервиса уже существующие приложения на любых языках программирования без разработки дополнительных программных компо-

нентов. Данная возможность позволяет быстро преобразовать в алгоритмические сервисы широкий спектр существующих приложений, не обладая при этом навыками программирования.

Компонент JobManager осуществляет управление обработкой запросов к сервисам. Запросы преобразуются в задания (job) и размещаются в очереди, откуда затем извлекаются конфигурируемым пулом потоков-обработчиков. При выполнении задания в соответствии с внутренней конфигурацией сервиса выбирается та или иная реализация обработчика задания.

В случае если реализацией сервиса является Java-класс, то используется обработчик JavaJob, осуществляющий вызов данного класса в рамках текущей виртуальной машины Java. Если реализацией сервиса является команда, то используется обработчик CommandJob, осуществляющий отображение входных параметров и запуск данной команды в отдельном процессе операционной системы. Если реализацией сервиса является грид-задание, то используется обработчик GridJob, осуществляющий отображение входных параметров и запуск данного грид-задания в инфраструктуре EGEE с помощью библиотеки jLite [7].

Все реализации обработчиков заданий осуществляют загрузку и сохранение входных файлов в выделенной для задания директории, а также управление состоянием запроса в ходе выполнения задания. Выходные файлы задания также создаются и хранятся в директории задания. После успешного выполнения задания обработчик сохраняет полученные от реализации сервиса результаты и изменяет состояние запроса на DONE.

Каждый размещаемый на сервере EveREST сервис полностью реализует описанный ранее интерфейс алгоритмического сервиса на базе формата JSON. Кроме того, EveREST поддерживает дополнительную HTML-версию данного интерфейса. На рис. 2 изображен автоматически сгенерированный сервером веб-интерфейс для отправки запроса к тестовому сервису Echo. Таким образом, EveREST обеспечивает не только легкое преобразование существующих приложений в сервисы, но и реализует генерацию веб-интерфейсов, позволяющих пользователям непосредственно обращаться к сервисам через веб-браузер.

Заключение

В работе описана модель алгоритмического сервиса, которая позволяет унифицировать удаленный доступ к широкому спектру вычислительных приложений, служащих для решения определенных классов научных и прикладных задач. Рассмотрена применимость различных технологий для реализации удаленного доступа к алгоритмическим сервисам. Разработан унифицированный интерфейс доступа к алгоритмическим сервисам на основе технологий Web и подхода REST. Создана эталонная программ-

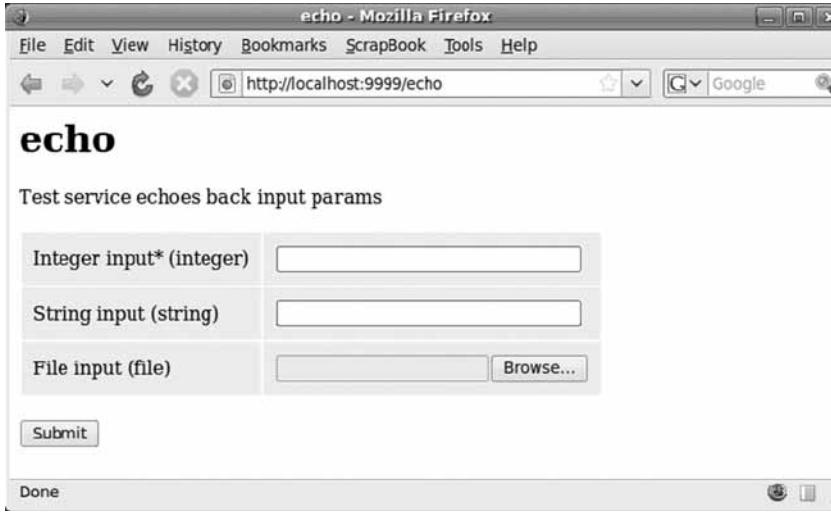

Рис. 2. Веб-интерфейс для отправки запроса к тестовому сервису Echo

ная реализация данного интерфейса, демонстрирующая применимость предлагаемого подхода на практике.

Одним из первых проектов по унификации программного доступа к удаленным вычислительным приложениям является NetSolve/GridSolve [7], использующий парадигму RPC. Изложенные в данной работе результаты являются развитием и переосмыслением идей, описанных в работах над инструментарием IARnet [8–10] на базе объектно-ориентированного ППО. Прототипом для модели алгоритмического сервиса и сервера EveREST послужил сервис обработки геоданных GDPS [11]. Автору неизвестны аналогичные разработки по созданию унифицированного интерфейса к алгоритмическим сервисам в Web на основе подхода REST. Примерами веб-сервисов на основе парадигмы RPC могут служить научные WS/Grid-сервисы и XML-RPC-сервисы (например, сервисы оптимизации NEOS [12]). Реализация асинхронной обработки запросов в рамках REST обсуждается в [2, 13]. Изложенные там подходы согласуются с разработанным интерфейсом.

В настоящее время разработанный подход проходит апробацию на практике в рамках работ над распределенной математической средой MathCloud [14]. В частности, ведется разработка пилотных алгоритмических сервисов на основе математических пакетов, а также разработка независимой реализации контейнера алгоритмических сервисов в соответствии с изложенным в статье описанием интерфейса. Дальнейшие исследования будут посвящены развитию предлагаемого подхода с учетом опыта его использования в среде MathCloud.

Литература

1. *Fielding R. T.* Architectural styles and the design of network-based software architectures. PhD Dissertation. Dept. of Information and Computer Science, University of California, Irvine, 2000.
2. *Richardson L., Ruby S.* RESTful Web Services. O'Reilly, 2007.
3. JSON Schema Proposal: http://www.json-schema.org/, 01.09.2009.
4. Jersey: https://jersey.dev.java.net/, 01.09.2009.
5. JAX-RS (Java API for RESTful Web Services): https://jsr311.dev.java.net/, 01.09.2009.
6. *Сухорослов О. В.* Высокоуровневые средства разработки Grid-приложений для инфраструктуры EGEE // Труды международной научной конференции (Нижний Новгород, 30 марта – 3 апреля 2009 г.). Челябинск: Изд. ЮУрГУ, 2009. С. 718–723.
7. NetSolve/GridSolve: http://icl.cs.utk.edu/netsolve/, 01.09.2009.
8. *Емельянов С. В., Афанасьев А. П., Волошинов В. В., Гринберг Я. Р., Кривцов В. Е., Сухорослов О. В.* Реализация Grid-вычислений в среде IARnet // Информационные технологии и вычислительные системы. М.: Институт микропроцессорных вычислительных систем РАН. 2005. № 2. С.61–75.
9. *Афанасьев А. П., Волошинов В. В., Сухорослов О. В.* Высокоуровневый инструментарий для доступа к ресурсам распределенной вычислительной среды. // Распределенные вычисления и Грид-технологии в науке и образовании: Труды второй международной конференции (Дубна, 26–30 июня 2006 г.). Дубна: ОИЯИ, 2006. С. 197–209.
10. *Afanasiev A., Sukhoroslov O., Posypkin M.* A High-Level Toolkit for Development of Distributed Scientific Applications. // Victor E. Malyshkin (Ed.): Parallel Computing Technologies, 9th International Conference, PaCT 2007, Pereslavl-Zalessky, Russia, September 3–7, 2007, Proceedings. Lecture Notes in Computer Science 4671 Springer 2007, ISBN 978–3–540–73939–5. P. 103–110.
11. *Волошинов В. В., Сухорослов О. В.* Высокоуровневый Grid-сервис обработки данных // Прикладные проблемы управления макросистемами / Под. ред. Ю. С. Попкова, В. А. Путилова. Т. 39. М.: Книжный дом «Либроком»/URSS, 2008. С. 212–219.
12. NEOS Server for Optimization: http://www-neos.mcs.anl.gov/, 01.09.2009.
13. Handling Asynchronous REST Operations: http://www.infoq.com/news/2009/07/AsynchronousRest, 01.09.2009.
14. *Астафьев А. С., Афанасьев А. П., Лазарев И. В., Сухорослов О. В., Тарасов А. С.* Научная сервис-ориентированная среда на основе технологий Web и распределенных вычислений. // Научный сервис в сети Интернет: масштабируемость, параллельность, эффективность: Труды Всероссийской суперкомпьютерной конференции (21–26 сентября 2009 г., Новороссийск). М.: Изд-во МГУ, 2009. С. 463–467.
15. Руководство пользователя ПФС «Геоанализ», ЗАО «ЭРМА СОФТ Менеджмент». М., 2006. 57 с.