Промежуточное программное обеспечение Ice*

О. В. Сухорослов

Институт системного анализа Российской академии наук (ИСА РАН)

В статье рассматривается объектно-ориентированное промежуточное программное обеспечение (ППО) Ice. Описываются архитектура, модель программирования и дополнительные сервисы Ice. Проводится сравнение Ice с другими технологиями ППО. Делаются выводы о применимости Ice для реализации распределенных научных приложений и вычислительных сред.

Введение

Взаимодействие между компонентами распределенного приложения осуществляется путем передачи по сети заявок на обслуживание в виде вызовов удаленных объектов или процедур. Данное взаимодействие может быть реализовано путем непосредственного использования возможностей сетевых операционных систем. Однако такой подход является неэффективным с точки зрения удобства разработки и сопровождения. Существует большой концептуальный разрыв между низкоуровневыми функциями сетевых операционных систем (чтение и запись байтовых потоков) и высокоуровневыми потребностями разработчиков (вызов операций удаленных объектов). При непосредственном использовании сетевой операционной системы разработчику распределенного приложения приходится самостоятельно ликвидировать этот разрыв,

* Работа выполнена при поддержке Совета по грантам Президента РФ (грант МК-3128.2007.9), фонда РФФИ (грант № 05-07-90182-в), Президиума РАН (программа фундаментальных исследований 15П, проект 1.1) и Федерального агентства по науке и инновациям (государственный контракт № 02.514.11.4053).

что повышает трудоемкость разработки и вероятность ошибок. Гетерогенность распределенной системы, то есть использование на узлах сети различных представлений данных, операционных систем и языков программирования, еще более усложняет разработку приложения.

Для ликвидации упомянутого разрыва между сетевыми операционными системами и распределенными приложениями размещают слой *промежуточного ПО (middleware, ППО)* [1]. ППО позволяет разработчику подняться на более высокий уровень абстракции, который реализуется на основе низкоуровневых примитивов, предоставляемых сетевыми операционными системами. Слой ППО реализует уровень сессии и уровень представления эталонной модели ISO/OSI, тем самым, избавляя разработчиков распределенных систем от работы непосредственно с реализацией транспортного уровня. На уровне представления ППО решает задачи, связанные с устранением различий в представлении данных на различных узлах сети и преобразованием структур данных в формат, необходимый для их передачи транспортным уровнем. На уровне сессий ППО обеспечивает отображение ссылок на компоненты в физические адреса, реализацию удаленных вызовов и синхронизацию клиентов и серверов.

Впервые концепция ППО была реализована в начале 80-х годов в виде *удаленных вызовов процедур (Remote Procedure Calls, RPC)*. Как следует из названия, данная разновидность ППО обеспечивает взаимодействие между компонентами распределенного приложения путем реализации вызова процедур, предоставляемых этими компонентами, по сети.

В середине 90-х годов произошло появление *объектно-ориентированного ППО* [1], которое позволило разработчикам создавать распределенные приложения с использованием объектно-ориентированной парадигмы. Принципы инкапсуляции, наследования и полиморфизма позволяют отделить спецификацию распределенного приложения от его непосредственной реализации, тем самым, обеспечивая интероперабельность, расширяемость и независимость реализаций отдельных компонентов приложения. Объектно-ориентированное ППО, такое как CORBA и DCOM, подняло распределенное программирование на новый уровень. Тем не менее, ни одной из этих технологий не удалось стать общепринятым стандартом разработки распределенных приложений. Связано это было с рядом причин.

CORBA (Common Object Request Broker Architecture) [2] является наиболее широко используемым стандартом объектно-ориентированного ППО. Разработка спецификаций CORBA ведется с 1989 года в рамках

некоммерческого консорциума Object Management Group (OMG) [3]. Спецификации CORBA были реализованы многими производителями, в том числе на различных языках программирования и операционных системах. Тем не менее, зачастую трудно было найти одну реализацию, поддерживающую все требуемые спецификации, языки и операционные системы. Несмотря на стандартизацию CORBA, реализации различных производителей не были полностью совместимыми и использовали проприетарные расширения, что затрудняло их совместное использование.

Представленная в 1996 году компанией Microsoft альтернативная технология *DCOM (Distributed Component Object Model)* [5] являлась проприетарной и поддерживала только операционные системы семейства Windows. Кроме того, DCOM имела ограниченную масштабируемость, связанную в первую очередь с использованием распределенного механизма «сборки мусора». Эти особенности заметно ограничивали сферу применения DCOM.

Общепризнанным фактом является чрезмерная сложность CORBA и DCOM. Для полноценного освоения этих технологий требовались месяцы и годы. На сложность спецификаций CORBA повлияли особенности процесса их стандартизации в рамках консорциума, а также отсутствие эталонной реализации. В результате этого в спецификациях оказалось много избыточной или вовсе бесполезной функциональности, которая никогда не была реализована [4]. Сложность спецификаций привела к сложности API и реализаций, что не лучшим образом влияло на производительность.

В 1997 году компания Sun Microsystems добавила в язык Java поддержку *удаленных вызовов методов (remote method invocation, RMI)* [6]. RMI позволяет объекту-клиенту, находящемуся в одной виртуальной машине Java, вызывать метод удаленного объекта-сервера, находящегося в другой виртуальной машине. Первоначально RMI создавалась для поддержки сетевого взаимодействия между объектами, написанными исключительно на языке Java. Позднее в RMI была добавлена поддержка взаимодействия с CORBA-объектами. По сравнению с CORBA, Java RMI предоставляет сравнительно бедный набор средств распределенного программирования. На основе RMI компания Sun реализовала ряд технологий более высокого уровня, таких как Enterprise Java Beans (EJB).

В начале 2000-х годов интерес к CORBA пошел на спад, и развитие технологии замедлилось. Ряд производителей отказался от дальнейшей поддержки своих реализаций CORBA. Интерес оставшихся про-

изводителей к развитию стандартов CORBA ослаб. Это привело к тому, что многие недостатки в спецификациях CORBA по-прежнему не устранены, либо были устранены с большой задержкой. Последняя версия стандарта CORBA (3.0.2) датируется 2002 годом.

В том же 2002 году Microsoft заменила DCOM новой *платформой .NET (.NET Framework)* [7]. В состав .NET вошла технология .NET Remoting, ставшая логическим развитием и устранившая многие недостатки DCOM. Однако область применения этой технологии по-прежнему ограничена ОС Windows. Несмотря на наличие реализаций отдельных частей платформы.NET для других операционных систем, эти реализации не являются полными и не используются большинством разработчиков. В конце 2006 года на смену .NET Remoting пришла технология Windows Communication Foundation, основанная на Web-сервисах.

Одновременно со спадом интереса к объектно-ориентированному ППО, большую популярность среди разработчиков распределенных приложений получили Web-сервисы [8]. World Wide Web Consortium (W3C) определяет Web-сервис как «программную систему, поддерживающую интероперабельное межмашинное взаимодействие по сети» [9]. Взаимодействие с Web-сервисом осуществляется посредством передачи XML-сообщений поверх стандартных Web-протоколов, таких как HTTP. Интерфейс Web-сервиса, то есть структура сообщений, принимаемых и возвращаемых сервисом, описывается с помощью языка WSDL.

Главной целью технологии Web-сервисов является обеспечение интероперабельности. Согласно приведенному определению, запрашивающая сторона может получить доступ к сервису при помощи стандартного протокола HTTP. В идеальном случае, любая запрашивающая сторона может взаимодействовать с любым приложением, являющимся Web-сервисом, безотносительно языка программирования и среды выполнения, используемыми каждой из сторон. Это свойство Web-сервисов делает данный подход привлекательным для интеграции приложений на межорганизационном уровне и построения крупномасштабных распределенных систем.

Архитектура Web-сервисов [9] является реализацией более общей *сервисно-ориентированной архитектуры (Service-Oriented Architecture, SOA)* [10], в основе которой лежит понятие *сервиса* как некоторой функциональности, доступной по сети. Сервисы публикуют свои описания в специальной службе, называемой *реестром сервисов (service registry)*. Приложения, которым требуется определенная функциональность, обнаруживают подходящий сервис при помощи реестра и затем

отправляют запрос найденному сервису. Функциональность сервиса может одновременно использоваться в контексте сразу нескольких приложений. Кроме того, при выполнении запросов сервис может использовать функциональность других сервисов. Таким образом, в SOA достигается переход от монолитных распределенных приложений к приложениям, состоящим из набора *слабо связанных (loosely coupled)* распределенных компонентов, обнаруживаемых динамически в сети.

В настоящее время Web-сервисы активно развиваются при поддержке индустрии и претендуют на роль универсального ППО, обеспечивающего интероперабельность и воплощающего на практике сервис-ориентированный подход. Несмотря на это Web-сервисы обладают рядом серьезных недостатков, таких как низкая производительность и эффективность протокола, сложный язык описания интерфейсов, отсутствие высокоуровневой модели программирования и соответствующих спецификаций, проблемы с интероперабельностью и непереносимость приложений между проприетарными платформами, большое количество развивающихся спецификаций и общая незрелость технологии.

Таким образом, выбор любой из упомянутых выше технологий ППО неизбежно связан с определенными проблемами, будь то сложность технологии, ее закрытость или низкая производительность. В любом из случаев разработчику приходится идти на некоторый компромисс.

В сложившейся ситуации чрезвычайно интересной представляется технология *Ice (Internet Communication Engine)* [11, 12], разработка которой ведется компанией ZeroC, Inc. [13] с начала 2000-х годов. При создании Ice преследовались следующие цели:

- реализовать объектно-ориентированное ППО, поддерживающее работу в гетерогенных средах;
- обеспечить всю функциональность, необходимую при создании реалистичных распределенных приложений;
- избегать излишнего усложнения платформы, обеспечивая легкость ее изучения и использования;
- обеспечить высокую производительность и эффективность реализации в плане использования сетевого трафика и системных ресурсов;
- реализовать встроенный механизм безопасности, поддерживающий работу в глобальных сетях.

Фактически, Ice является попыткой реализовать ППО, сравнимое по своим возможностям с CORBA, но лишенное при этом упомянутых

выше недостатков. Ice поддерживает разработку серверных и клиентских приложений на многих популярных языках, таких как C++, Java, C#, Visual Basic, Python, PHP и Ruby. Ice доступен на всех распространенных разновидностях операционных систем. Также существует версия Ice-E (Embedded Ice) для мобильных и встроенных устройств. Немаловажной особенностью является то, что Ice распространяется вместе с исходным кодом под лицензией GNU General Public License (GPL). Это позволяет бесплатно использовать Ice в open source проектах.

Первая версия Ice вышла в феврале 2003 года. За прошедшие пять лет Ice трансформировался в мощную технологию ППО с впечатляющим списком поддерживаемых языков и платформ, стабильными спецификациями и API, а также обширной документацией. Во многом это обусловлено участием в разработке Ice известных экспертов в области ППО, таких как Мичи Хеннинг (Michi Henning), автор книги «Advanced CORBA Programming with C++». Ice используется в ряде крупных коммерческих систем, таких, например, как Skype.

Тем не менее данная технология пока мало известна широкому кругу прикладных программистов, в том числе в академической среде. В то же время представляется, что Ice является практически идеальной технологией для реализации научных распределенных приложений и вычислительных сред, обладающей рядом преимуществ перед широко используемыми сейчас технологиями Web-сервисов. Данная статья призвана восполнить пробел, связанный с отсутствием материалов о Ice на русском языке, а также привлечь внимание разработчиков к данной технологии.

В первой главе рассматриваются архитектура и модель программирования Ice. Во второй главе приводится описание дополнительных сервисов, входящих в состав Ice. Третья глава посвящена сравнению Ice с технологией CORBA и Web-сервисами. В заключении делаются выводы о применимости Ice для реализации распределенных научных приложений и вычислительных сред.

1. Архитектура и модель программирования Ice

Ice является объектно-ориентированным ППО, то есть предоставляет средства для разработки объектно-ориентированных распределенных приложений. Данные приложения реализуются в соответствии с известной моделью «клиент-сервер». *Клиентами* называются активные

сущности, запрашивающие определенные сервисы у сервера. *Серверами* называются пассивные сущности, предоставляющие сервисы в ответ на запросы клиентов. Зачастую отдельные части распределенного приложения не являются «чистыми» клиентами или серверами, а сочетают в себе обе функции. Поэтому корректнее говорить о роли, которую играет та или иная часть приложения в контексте конкретного удаленного вызова.

Ice поддерживает разработку приложений в гетерогенной среде, то есть отдельные части приложения (клиенты и серверы) могут быть реализованы на различных языках программирования и работать под управлением различных операционных систем на различных вычислительных платформах.

1.1. Основные понятия

1.1.1. Ice-объект

Модель программирования Ice основана на понятии *Ice-объекта (Ice object)*. Это абстрактная сущность, которая описывается при помощи следующих утверждений:

- Ice-объект — сущность в локальном или удаленном адресном пространстве, которая может отвечать на запросы клиентов.
- Ice-объект может функционировать на одном сервере или сразу на нескольких.
- Каждый Ice-объект имеет один или несколько *интерфейсов* — наборов именованных *операций*, поддерживаемых объектом. Клиенты отправляют запросы объекту путем вызова его операций.
- Операция имеет ноль или более *параметров*, а также *возвращаемое значение*. Параметры и возвращаемые значения имеют определенный тип данных. Параметры имеют имя и направление: *in-параметры* инициализуются клиентом и передаются серверу, *out-параметры* инициализуются сервером и передаются клиенту.
- Ice-объект имеет как минимум один выделенный интерфейс, называемый *главным интерфейсом*. Кроме этого, Ice-объект может предоставлять несколько дополнительных интерфейсов, называемых *фасетами (facets)*.
- Каждый Ice-объект имеет уникальный *идентификатор (object identity)*. Идентификатор объекта отличает его от всех других объ-

ектов. Объектная модель Ice подразумевает, что идентификаторы объектов являются глобально уникальными, то есть никакие два объекта в рамках системы не могут иметь одинаковые идентификаторы.

1.1.2. Язык Slice

Интерфейсы Ice-объектов и типы данных, которыми обмениваются клиент и объект, описываются на декларативном языке *Slice (The Specification Language for Ice),* в независящем от языков программирования виде. Slice-описания транслируются во вспомогательный код на конкретном языке программирования при помощи *Slice-компилятора*. Этот вспомогательный код используется разработчиком при реализации клиента или сервера на данном языке (см. раздел 1.3). Правила, в соответствии с которыми конструкции Slice транслируются в определенный язык программирования, называются *отображениями (language mappings)*. На момент написания статьи, Ice поддерживает отображения и Slice-компиляторы для языков C++, Java, C#, Visual Basic.NET, Python, PHP и Ruby (последние два — только для разработки клиентов).

1.1.3. Прокси

Разработчик клиента использует для вызова Ice-объекта так называемый *прокси (proxy)*. Прокси — это представитель Ice-объекта на стороне клиента, в его локальном адресном пространстве. Код прокси для определенного языка программирования генерируется Slice-компилятором. В случае Java это Java-объект, содержащий методы для вызова операций Ice-объекта. Вызовы методов прокси передаются среде выполнения Ice, которая осуществляет передачу параметров вызова по сети к удаленному объекту, принимает обратно результаты вызова и возвращает их клиенту.

Прокси инкапсулирует в себе информацию, необходимую для вызова Ice-объекта: физический адрес сервера, идентификатор объекта и, дополнительно, идентификатор фасета. Данная информация может быть представлена в виде строки, называемой *прокси-строкой (stringified proxy)*. Ice API позволяет создавать прокси Ice-объекта по его прокси-строке и наоборот — получать прокси-строку по прокси. Прокси-строки используются, например, для хранения ссылок на Ice-объекты в конфигурационных файлах или базе данных.

Существует два основных типа прокси: прямые и косвенные.

Прямые прокси (direct proxy) содержат информацию об адресе сервера в виде идентификатора протокола, имени хоста и номера порта.

Среда выполнения Ice использует данную информацию при передаче вызовов от клиента к серверу. В каждом вызове содержится идентификатор вызываемого объекта. Важно отметить, что внутри прокси могут содержаться адреса нескольких серверов, обслуживающих данный объект. При установлении соединения с сервером среда выполнения Ice выбирает один из указанных адресов и, если происходит отказ, последовательно перебирает остальные адреса. Это позволяет простым образом реализовать *репликацию* серверов.

Косвенные прокси (indirect proxy) не содержат информацию об адресе сервера. Первый вариант косвенного прокси содержит только идентификатор объекта. Такой объект называется *известным объектом (well-known object)*. Второй вариант косвенного прокси помимо идентификатора объекта содержит идентификатор так называемого объектного адаптера (см. раздел 1.2.4).

Для того чтобы определить адрес сервера, на котором находится объект, среда выполнения Ice на клиентской стороне обращается к *службе имен (location service)*. Служба имен хранит соответствия между идентификаторами известных объектов или объектных адаптеров и адресами серверов. После получения адреса сервера среда выполнения Ice осуществляет передачу клиентского вызова обычным образом. Данный механизм разрешения косвенных прокси аналогичен работе службы DNS. Главным преимуществом косвенных ссылок является возможность миграции объектов и серверов без необходимости обновления прокси, используемых клиентами. Косвенные ссылки также позволяют реализовать произвольные механизмы репликации и балансирования нагрузки (см. раздел 2.5).

1.1.4. Сервант

На серверной стороне поведение (обработка вызовов операций) Ice-объектов реализуется так называемыми *сервантами (servant)*.

На практике сервант это экземпляр реализованного разработчиком класса, который зарегистрирован в среде выполнения Ice как сервант для определенных Ice-объектов. Базовый код серванта, так называемый *скелетон*, генерируется Slice-компилятором (см. раздел 1.2.3). Разработчику требуется реализовать методы класса, которые соответствуют операциям из интерфейса Ice-объекта. При поступлении вызова среда выполнения Ice на серверной стороне находит соответствующий вызываемому объекту сервант и делегирует ему обработку вызова.

Один сервант может обслуживать одновременно сразу несколько Ice-объектов. При вызове серванту передается идентификатор вызываемого объекта, в соответствии с которым сервант реализует обработку запроса.

Верно и обратное — один Ice-объект может обслуживаться несколькими сервантами. Напомним, что в прокси может быть указано несколько адресов серверов, на каждом из которых функционирует сервант данного объекта.

1.1.5. Исключения

Вызов операции Ice-объекта может привести к возникновению *исключения (exception)*. Исключения Ice бывают двух типов.

Исключения среды выполнения (run-time exceptions) возвращаются средой выполнения Ice при возникновении ошибок общего характера, связанных с передачей удаленного вызова, таких как невозможность установить соединение, разрыв соединения или отсутствие вызываемого объекта.

Пользовательские исключения (user exceptions) используются для уведомления клиента об ошибках, возникающих при обработке вызова Ice-объектом. Данные исключения являются специфичными для приложения и определяются при описании интерфейсов объектов на языке Slice. Каждая операция Ice-объекта может возвращать одно или несколько пользовательских исключений. Исключения могут иметь произвольную структуру данных. Slice поддерживает наследование исключений, что позволяет организовать иерархию исключений.

Оба типа исключений передаются приложению в виде «родных» исключений используемого языка программирования.

1.2. Структура клиента и сервера

Клиенты и серверы Ice имеют следующую структуру, изображенную на рис. 1.

Код клиента и сервера состоит из трех частей:

- код приложения, написанный программистом;
- код ядра Ice, находящийся в библиотеках Ice;
- вспомогательный код, сгенерированный Slice-компилятором.

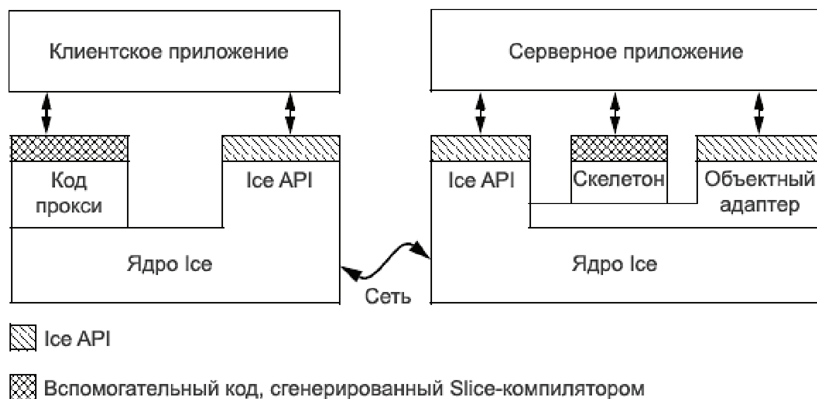

Рис. 1. Структура клиента и сервера Ice

Весь код, связанный с реализацией сетевого взаимодействия, находится в двух последних частях. Таким образом, программисту требуется реализовать только логику отправки и обработки вызовов.

1.2.1. Ядро Ice и Ice API

Ядро Ice (Ice core) содержит реализацию базового коммуникационного механизма, обеспечивающего функционирование среды выполнения Ice на клиентской и серверной сторонах. В обязанности ядра входит управление сетевыми соединениями, передача данных по сети, управление потоками и т. д. Ядро Ice представлено набором библиотек, с которыми компонуются клиентское и серверное приложения.

Доступ к ядру Ice осуществляется через интерфейс прикладного программирования *Ice API*. Этот API используется, например, для инициализации и уничтожения среды выполнения Ice. Ice API идентичен для клиентов и серверов, хотя серверные приложения используют большую часть API.

1.2.2. Прокси

Код прокси (proxy code) генерируется Slice-компилятором и, следовательно, специфичен для типов объектов и данных, содержащихся в Slice-описании. Код прокси выполняет две основные функции.

Во-первых, прокси предоставляет интерфейс для вызова объекта на клиентской стороне. Вызов метода прокси передается ядру Ice, которое осуществляет передачу запроса по сети и вызов соответствующей операции объекта.

Во-вторых, прокси осуществляет маршаллинг и демаршаллинг данных. *Маршаллингом (marshalling)* называется процесс преобразования структуры данных, представленной на используемом языке программирования, в универсальный формат, используемый при передаче данных по сети. Прокси осуществляет маршаллинг параметров вызова перед тем, как передать их ядру Ice. *Демаршаллингом (unmarshalling)* называется обратный процесс преобразования данных, полученных по сети, из универсального формата в локальное представление структуры данных на используемом языке программирования. Прокси осуществляет демаршаллинг результатов вызова, полученных от ядра Ice, перед тем, как вернуть их клиентскому приложению.

1.2.3. Скелетон

Аналогом прокси на серверной стороне является *скелетон (skeleton)*. Код скелетона генерируется Slice-компилятором и, следовательно, специфичен для типов объектов и данных, содержащихся в Slice-описании. Скелетон реализует интерфейс, с помощью которого среда выполнения Ice передает полученные вызовы для обработки серверному приложению. Скелетон также содержит код для маршаллинга и демаршаллинга данных. Демаршаллинг выполняется при приеме параметров вызова от ядра Ice, а маршаллинг при возврате результатов вызова, включая возникшие исключения. Скелетон является «заготовкой», которую программист использует при реализации серванта. Серверное приложение и скелетон вместе образуют код серванта.

1.2.4. Объектный адаптер

Объектный адаптер (object adapter) является частью Ice API, используемой только на серверной стороне.

Объектный адаптер осуществляет поиск серванта, которому следует передать обработку входящего вызова. Для этого адаптер хранит соответствия между сервантами и идентификаторами обслуживаемых ими объектов. Данные соответствия устанавливаются разработчиком серверного приложения путем регистрации серванта на объектном адаптере через Ice API.

Каждый объектный адаптер имеет одну или несколько *транспортных точек (transport endpoints)*. Транспортная точка определяет сетевой адрес, порт и протокол, используемые для вызова объектов, связанных с объектным адаптером. С адаптером может быть связано несколько транспортных точек с различными характеристиками: UDP-

точка (высокая производительность, низкая надежность), TCP-точка (гарантированная надежность), SSL (обеспечение безопасности).

В обязанности объектного адаптера также входит создание прокси для связанных с адаптером объектов. Объектный адаптер включает в прокси информацию о поддерживаемых транспортных точках. Созданный через адаптер прокси может быть передан клиенту.

1.3. Процесс разработки клиента и сервера

Рассмотрим процесс разработки клиента и сервера на Ice. Конечным результатом данного процесса являются исполняемые файлы клиента и сервера, которые могут быть запущены на любой платформе, поддерживаемой Ice, при наличии соответствующих библиотек.

На рис. 2 изображен процесс разработки клиента и сервера в случае, когда клиент реализуется на языке Java, а сервер — на языке C++.

В качестве общей основы для написания клиента и сервера, разработчики используют Slice-описание, содержащее определения интерфейсов объектов, типов данных, исключений и т. д. По сути, данный доку-

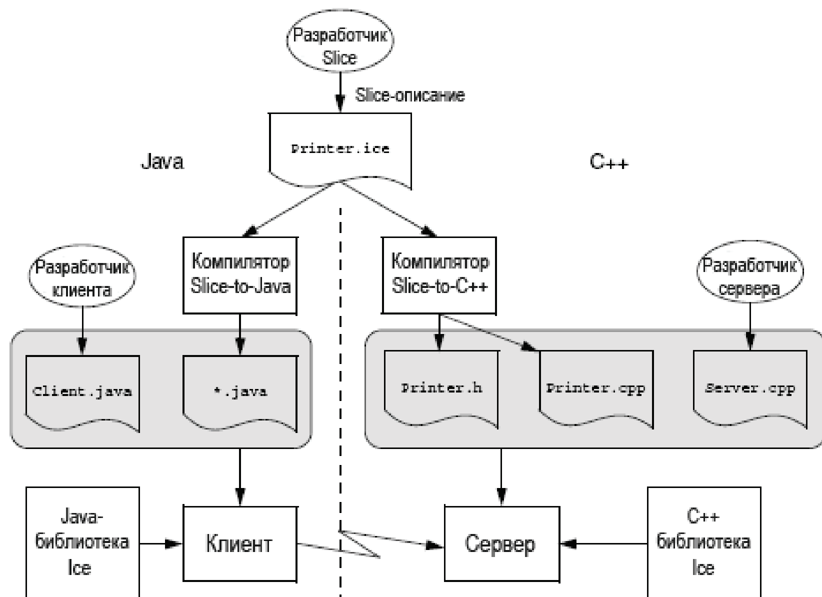

Рис. 2. Процесс разработки клиента и сервера на различных языках программирования

мент определяет контракт, который должны соблюдать разработчики клиента и сервера. Slice-описание помещается в файл с расширением «.ice» («Printer.ice» на рис. 2).

Первым шагом является генерация вспомогательного кода по Slice-описанию. Для этого разработчик клиента использует Slice-компилятор для Java (slice2java), а разработчик сервера — Slice-компилятор для C++ (slice2cpp). В обоих случаях компилятор транслирует Slice-определения в описания типов и API на используемом языке программирования. Вспомогательный код содержит:

1. Определения интерфейсов, типов данных и исключений, эквивалентные Slice-описанию.
2. API для создания прокси и код прокси.
3. Код скелетона.

Разработчик клиента пишет код клиента на Java («Client.java» на рис. 2), используя Ice API и вспомогательный код (части 1 и 2) для инициализации среды выполнения Ice, создания прокси и вызова объектов. Код клиента и вспомогательный код компилируются в исполняемый файл клиента.

Разработчик сервера пишет код сервера на C++ («Server.cpp» на рис. 2), используя Ice API и вспомогательный код (части 1 и 3) для инициализации среды выполнения Ice, реализации сервантов и их регистрации на объектном адаптере. Код сервера и вспомогательный код компилируются в исполняемый файл сервера.

Клиент и сервер также компонуются с библиотеками Ice для соответствующих языков программирования.

При компоновке клиентского и серверного приложения также используются библиотеки Ice для соответствующих языков программирования

Описанный процесс разработки поддерживает создание нескольких реализаций клиентов и серверов. Эти реализации могут быть написаны на одном или различных языках программирования, в соответствии с требованиями конкретного приложения. Независимо от этого, основанный на Slice процесс разработки гарантирует совместимость любых реализаций клиента и сервера.

Примеры исходных кодов клиентов и серверов Ice на различных языках программирования можно найти в [12] и в дистрибутиве Ice.

1.4. Модели удаленных вызовов

1.4.1. Семантика не более одного вызова

Вызовы Ice объектов имеют по умолчанию семантику *«не более одного вызова» (at-most-once semantics)*. Среда выполнения Ice гарантирует, что либо вызов будет доставлен серверу в точности один раз, либо, если все попытки передать вызов оказались неудачными, среда сообщит клиенту о невозможности доставки вызова с помощью соответствующего исключения. Ни при каких обстоятельствах вызов не будет доставлен более одного раза. Попытки повторно отправить запрос выполняются только в том случае, если достоверно известно, что предыдущая попытка завершилась неудачно. Исключением из данного правила являются вызовы с использованием протокола UDP (см. далее).

Семантика «не более одного вызова» обеспечивает безопасный вызов операций, эффект которых зависит от числа совершенных вызовов. К таким операциям относятся, например, увеличение значения счетчика или перевод денег с одного счета на другой.

Операция, несколько последовательных вызовов которой имеют тот же эффект, что и единичный вызов, называется *идемпотентной (idempotent)*. К таким операциям относятся, например, все операции чтения или присваивание заданного значения внутренней переменной. Для таких операций не требуется гарантировать однократность вызова. Ice позволяет пометить отдельные операции Ice-объекта как идемпотентные и использует при их вызове более агрессивные механизмы восстановления после ошибок, чем при вызове неидемпотентных операций.

1.4.2. Синхронные и асинхронные вызовы

По-умолчанию, при диспетчеризации вызовов на клиентской стороне Ice использует модель *синхронных вызовов*. В этом случае вызовы Ice-объектов ведут себя так же, как вызовы локальных объектов. Поток клиента блокируется на время выполнения вызова и возобновляется только когда вызов завершен и доступны все его результаты.

Ice также поддерживает модель *асинхронных вызовов (asynchronous method invocation, AMI)*. В этом случае клиент, вызывая объект при помощи прокси, наряду с обычными параметрами операции передает прокси специальный *callback-объект*. После вызова прокси управление сразу возвращается клиенту. При завершении вызова объекта, среда выполнения Ice на клиентской стороне вызывает метод callback-объекта, передавая ему результаты вызова или исключения.

Указанные модели относятся только к диспетчеризации вызовов на клиентской стороне. С точки зрения сервера обработка синхронных и асинхронных вызовов выглядит абсолютно одинаково.

1.4.3. Диспетчеризация вызовов на серверной стороне

На серверной стороне Ice поддерживает аналогичные описанным выше синхронную и асинхронную модели диспетчеризации вызовов.

По-умолчанию, используется синхронная модель, когда среда выполнения Ice на серверной стороне направляет полученный от клиента вызов серванту и ожидает завершения обработки вызова. В этом случае серверный поток блокируется на время обработки вызова сервантом.

Модель *асинхронной диспетчеризации вызовов (asynchronous method dispatch, AMD)* позволяет обрабатывать запрос, не блокируя при этом серверный поток среды выполнения Ice. Когда среда выполнения Ice сообщает серванту о получении нового вызова, сервант может отложить обработку запроса и освободить серверный поток, осуществляющий диспетчеризацию вызова. При завершении выполнения запроса, сервант уведомляет об этом среду выполнения Ice, которая отправляет результаты вызова клиенту.

Асинхронная диспетчеризация вызовов эффективна в случаях, когда вызовы клиентов обрабатываются в течение длительного времени. При синхронной диспетчеризации каждый клиент, ожидающий результаты вызова, занимает серверный поток. При наличии достаточно большого числа клиентов все потоки будут заняты, и сервер перестанет обслуживать поступающие запросы. Асинхронная диспетчеризация позволяет избежать такой ситуации. Асинхронная диспетчеризация также может применяться для реализации сервантом дополнительных действий уже после того, как результаты вызова возвращены клиенту.

Описанные модели относятся к диспетчеризации вызовов на серверной стороне и прозрачны для клиента, то есть клиент не знает, с помощью какой модели осуществляется обработка его вызова.

1.4.4. Односторонние вызовы

Ice поддерживает *односторонние вызовы (oneway invocations)*, которые имеют семантику «best effort». При одностороннем вызове управление возвращается клиенту сразу после того, как среда выполнения Ice на клиентской стороне поместила сообщение в буфер сетевого соединения. Дальнейшая отправка вызова осуществляется асинхронно операционной системой. Сервер не отвечает на односторонние вызовы, то есть трафик направлен только от клиента к серверу. Односторон-

ние вызовы являются ненадежными, и клиент не уведомляется о том, был ли вызов доставлен и обработан.

Односторонние вызовы доступны только для операций, не имеющих возвращаемого значения, out-параметров и исключений. Кроме того, вызываемый объект должен быть доступен по протоколу поточной передачи данных, такому как TCP/IP и SSL. Однако это не гарантирует, что вызовы будут обработаны в порядке их отправления, поскольку диспетчеризация вызовов на серверной стороне может осуществляться в различных потоках.

Односторонние вызовы прозрачны для серверного кода, то есть они ничем не отличаются от обычных, *двусторонних вызовов (twoway invocations)*.

Для оптимизации накладных расходов при отправке нескольких односторонних вызовов Ice позволяет отправить пакет из нескольких вызовов как одно сообщение. При *пакетном одностороннем вызове (batched oneway invocation)* вызов отправляется серверу не сразу, а помещается в буфер среды выполнения Ice на клиентской стороне. Как только все требуемые вызовы помещены в буфер, при помощи API Ice инициируется одновременная отправка всех вызовов. Вызовы передаются средой выполнения Ice как одно сообщение, что снижает накладные расходы. На серверной стороне вызовы из пакета диспетчеризуются одним потоком в том порядке, в котором они были помещены в пакет при отправке. Таким образом, гарантируется сохранение порядка при выполнении вызовов.

Пакетные односторонние вызовы применяются, например, при реализации сервисов рассылки сообщений.

1.4.5. Дейтаграммные вызовы

Дейтаграммные вызовы (datagram invocations) имеют семантику «best effort», аналогичную односторонним вызовам. Однако в качестве транспорта используется протокол UDP. Управление возвращается клиенту сразу после того, как сообщение было помещено в локальный стек UDP. Дейтаграммные вызовы являются еще более ненадежными, чем односторонние вызовы, поскольку отдельные вызовы могут быть потеряны при передаче по сети или получены сервером не в том порядке. Тем не менее, дейтаграммные вызовы могут эффективно использоваться для передачи небольших сообщений в пределах локальной сети или при реализации интерактивных приложений, где низкая задержка важнее надежности.

Пакетные дейтаграммные вызовы (batched datagram invocations) позволяют поместить в буфер несколько вызовов и затем отправить их как одну дейтаграмму. В этом случае гарантируется, что либо будут доставлены все вызовы пакета, либо ни один из них. На серверной стороне вызовы из пакета диспетчеризуются одним потоком в том порядке, в котором они были помещены в пакет при отправке. Таким образом, гарантируется сохранение порядка при выполнении вызовов.

1.5. Протокол Ice

Для передачи удаленных вызовов Ice использует двоичный протокол, работающий поверх протоколов TCP/IP и UDP. Важными характеристиками протокола Ice являются:

- поддержка использования SSL для шифрования данных, передаваемых между клиентом и сервером, что позволяет реализовать полноценный механизм безопасности;
- поддержка автоматического сжатия передаваемых данных, позволяющего экономить сетевой трафик при передаче больших объемов данных;
- возможность организации пересылки сообщений промежуточными узлами, не обладающими информацией о содержимом этих сообщений, что позволяет, например, реализовать эффективный механизм рассылки сообщений;
- поддержка двусторонней (bidirectional) передачи данных, которая позволяет, например, использовать установленное соединение для передачи сообщений приложению, находящемуся за межсетевым экраном.

2. Сервисы Ice

Помимо коммуникационного ядра, в состав Ice входит несколько дополнительных сервисов, которые ориентированы на решение типичных задач, возникающих при разработке распределенных приложений. В настоящей главе приводится описание данных сервисов.

2.1. IceBox

IceBox — сервер общего назначения, позволяющий запускать внутри одного процесса несколько серверных приложений или сервисов.

IceBox является альтернативой обычным «монолитным» серверам, обслуживающим одно определенное приложение. Внутри IceBox могут быть размещены любые серверные приложения, оформленные в виде *IceBox-сервисов*. Сервисы динамически загружаются сервером через DLL, shared-библиотеку или Java-класс. Разработчик может определить порядок загрузки и остановки сервисов.

IceBox обладает следующими преимуществами по сравнению с монолитными серверами:

- Размещение сервисов осуществляется путем редактирования конфигурационных свойств и не требует перекомпиляции сервера. Это позволяет легко объединять несколько сервисов внутри одного сервера или распределять сервисы между несколькими серверами в произвольных комбинациях.
- IceBox-сервисы реализуют стандартный интерфейс, что позволяет унифицировать процесс разработки и администрирования сервисов.
- IceBox поддерживает удаленное администрирование: запуск и остановку сервисов, а также остановку сервера.
- Сервисы, размещенные внутри одного сервера, могут взаимодействовать друг с другом с использованием оптимизированных локальных вызовов.
- Для сервисов на Java используется один экземпляр виртуальной машины, что экономит системные ресурсы.

Для создания IceBox-сервиса требуется реализовать стандартный интерфейс сервиса, содержащий операции *start* и *stop*. Операция *start* вызывается сервером при загрузке сервиса. Внутри этой операции разработчик производит инициализацию сервиса, которая обычно заключается в создании объектного адаптера и сервантов. Операция *stop* вызывается сервером при остановке сервиса. Внутри этой операции разработчик должен освободить используемые сервисом ресурсы, такие как объектный адаптер.

В состав Ice входит три реализации IceBox для языков C++, Java и C#. Сервисы на C++ и C# оформляются в виде динамически загружаемых библиотек. Сервисы на Java загружаются как Java-классы.

2.2. IceStorm

При создании распределенных приложений часто требуется организовать рассылку информации нескольким получателям. Для этого не-

обходимо вести учет получателей, осуществлять доставку информации получателям и обрабатывать возникающие при этом ошибки. Реализация данной функциональности внутри компонента, поставляющего информацию, усложняет разработку и сильно «связывает» поставщика с получателями.

IceStorm — сервис рассылки сообщений, который действует в роли посредника между отправителями и получателями информации. IceStorm использует известную модель «publish-subscribe». Отправитель передает информацию путем *публикации сообщения* на сервисе. Сервис осуществляет доставку опубликованного сообщения всем получателям. Каждое сообщение относится к определенному *каналу рассылки (topic)*. Получатели информируют сервис о своей заинтересованности в получении сообщений определенной категории путем *подписки* на канал. Сервис доставляет получателю только те сообщения, которые относятся к указанным получателем каналам. Каждый канал рассылки может иметь несколько отправителей и получателей.

Использование сервиса IceStorm упрощает реализацию рассылки информации, «развязывая» при этом отправителей и получателей информации. А именно, отправители взаимодействуют только с сервисом и не знают ничего о подписчиках канала. Аналогично, получатели взаимодействуют только с сервисом и не знают ничего об отправителях, публикующих сообщения в канал.

Отличительная особенность IceStorm состоит в том, что сообщения являются строго типизированными и представляют собой вызов некоторой операции Slice-интерфейса. Соответственно, каждый канал представляет собой некоторый Slice-интерфейс. Операции этого интерфейса определяют типы сообщений, поддерживаемых каналом. Отправитель публикует сообщения путем вызова операций на прокси с интерфейсом канала. Аналогично, доставка сообщения получателю производится путем вызова объекта, реализующего интерфейс канала. Получатель передает сервису прокси на данный объект при подписке.

Таким образом, интерфейс канала представляет собой контракт между отправителями и получателями. Подобный способ взаимодействия ничем не отличается от прямого взаимодействия отправителя с одним получателем, за исключением того, что IceStorm прозрачным образом пересылает каждый запрос нескольким получателям.

IceStorm реализует модель доставки «push» и не поддерживает модель опроса «pull». Публикуемые сообщения имеют семантику одностороннего вызова. Таким образом, отправитель не может принимать

ответы от получателей. Модель доставки сообщений получателю определяется при подписке. Поддерживаются двусторонний, односторонний, пакетный односторонний, дейтаграммный и пакетный дейтаграммный вызовы. В случае если при доставке сообщения получателю происходит ошибка, сервис автоматически аннулирует подписку данного получателя на канал.

При подписке IceStorm также позволяет указать параметры качества обслуживания. В настоящее время поддерживается только один параметр *reliability*, который контролирует порядок доставки сообщений. При включенном параметре, IceStorm пересылает получателю сообщения в порядке их получения.

IceStorm поддерживает создание федераций или графов каналов рассылки. *Граф каналов* формируется путем создания связей между каналами. Каждая связь имеет направление и стоимость. При публикации сообщения в канале, оно также передается по всем связям, исходящим из данного канала и имеющим стоимость, равную или большую стоимости сообщения. Сообщения с нулевой стоимостью передаются по всем связям. Связи с нулевой стоимостью передают сообщения с любой стоимостью. Каналы, получившие сообщение, доставляют его своим подписчикам. При этом дальнейшая передача сообщения по связям не осуществляется.

IceStorm имеет базу данных, в которой он хранит информацию о каналах и связях между ними. Передаваемые сервисом сообщения не хранятся в базе данных, поскольку они доставляются сразу же после их публикации.

Для эффективной доставки сообщений IceStorm использует конфигурируемый пул потоков, в которых осуществляются вызовы получателей. Использование пула потоков позволяет снизить влияние «медленных» или недоступных получателей на работу сервиса.

IceStorm реализован на языке C++ как IceBox-сервис.

2.3. Glacier2

При создании распределенных приложений часто требуется реализовать взаимодействие между клиентами и серверами, находящимися в частных сетях, доступ к которым защищен межсетевыми экранами. Межсетевые экраны (МЭ) ограничивают входящие сетевые соединения, а также могут осуществлять трансляцию сетевых адресов (NAT). Эти особенности усложняют разработку приложений:

- Для организации внешнего доступа к серверу на серверном МЭ должен быть открыт порт и настроена пересылка сообщений серверу. Если сервер имеет несколько транспортных точек, для каждой из них должен быть выделен отдельный порт МЭ.
- Клиентский прокси должен использовать «публичную» транспортную точку сервера, содержащую адрес и выделенный порт серверного МЭ. Аналогично, если сервер возвращает прокси в результате вызова, прокси не должен содержать внутренний адрес сервера.
- При реализации обратных вызовов клиента сервером, клиент становится сервером и к нему также относятся указанные выше проблемы. В этом случае требуется настройка МЭ на клиентской стороне.
- С ростом числа клиентов и серверов сложность администрирования приложения возрастает.

Для решения описанных проблем в Ice используется сервис Glacier2, представляющий собой специализированный МЭ для Ice-приложений. Сервис Glacier2 размещается на серверной стороне и организует защищенное взаимодействие между клиентами и обслуживаемыми им серверами. Glacier2 обладает следующими преимуществами:

- Для работы с Glacier2 обычно требуется только минимальная модификация клиентов.
- Glacier2 использует только один порт для обслуживания любого числа внутренних серверов.
- Glacier2 действует как концентратор соединений, уменьшая число соединений с внутренними серверами.
- Для работы с Glacier2 не требуется модификация серверов. Более того, серверы не знают ничего о существовании Glacier2. С точки зрения серверов это обычный клиент. Серверам не требуется знать свои публичные адреса.
- Обратные вызовы от сервера к клиенту осуществляются через существующее соединение от клиента к серверу. Тем самым устраняется необходимость в конфигурации МЭ на клиентской стороне.
- Glacier2 осуществляет пересылку вызовов и их результатов эффективным образом, не осуществляя демаршаллинг сообщений.
- Glacier2 поддерживает буферизацию и пакетирование вызовов.
- Дополнительно Glacier2 позволяет осуществлять аутентификацию клиентов, управление клиентскими сессиями и фильтрацию вызовов.

2.4. IcePatch2

При разработке распределенных приложений часто приходится осуществлять обновления дистрибутивов и синхронизацию данных на множестве машин. Сервис IcePath2 позволяет организовать распространение обновлений ПО и любых данных. IcePatch2-сервер организует доступ к директории, содержащей распространяемые данные. При каждом обновлении данных выполняется архивация всех файлов и создание специального файла со списком контрольных сумм файлов. При подключении к серверу клиент сравнивает содержимое локального и серверного списков контрольных сумм, после чего осуществляет синхронизацию файлов. В результате на клиентской стороне воссоздается текущее содержимое директории IcePath2-сервера. Данные между сервером и клиентом передаются в сжатом виде для экономии сетевого трафика.

2.5. IceGrid

Сервис *IceGrid* содержит набор средств для управления распределенной инфраструктурой из множества компьютеров. Данную инфраструктуру разработчики Ice называют «grid» в широком смысле этого слова, используемом сейчас многими компаниями. Стоит понимать, что подобный, централизованно управляемый «корпоративный grid» не имеет отношения к крупномасштабным научным Grid-системам.

IceGrid ориентирован на решение типичных задач, возникающих при работе с серверной инфраструктурой, таких как:

- установка и обновление серверных приложений на всех компьютерах;
- отслеживание текущего состояния серверов;
- распределение нагрузки между серверами;
- миграция серверного приложения с одной машины на другую;
- быстрое добавление нового компьютера.

IceGrid также упрощает разработку распределенных приложений, работающих поверх подобной инфраструктуры.

Инфраструктура IceGrid состоит из *реестра (registry)* и произвольного числа *узлов (nodes)*. Реестр хранит информацию об узлах системы и развернутых на них приложениях. Узлы осуществляют запуск и остановку произвольных Ice-серверов, а также периодически уведомляют ре-

естр о своем состоянии. *Приложение* состоит из нескольких серверов, размещенных на определенных узлах системы.

Обычно на каждой машине запускается по одному узлу. Для работы реестра не требуется много системных ресурсов, поэтому он обычно запускается на одной машине с узлом. Реестр и узел могут быть запущены в одном процессе.

Реестр IceGrid выполняет много функций. В первую очередь, он реализует *службу имен Ice (location service)*, осуществляющую разрешение *косвенных прокси* (см. раздел 1.1.3). Когда клиент впервые пытается использовать косвенный прокси, среда выполнения Ice обращается к реестру, чтобы получить по имени объекта или адаптера соответствующие транспортные точки. Использование косвенных ссылок обладает рядом преимуществ. Во-первых, клиенту не требуется знать физический адрес и порт сервера, что упрощает выделение портов и миграцию серверов с одной машины на другую. Во-вторых, механизм разрешения позволяет реализовать в реестре дополнительную функциональность:

- *Активация сервера по требованию (on demand activation)* — реестр может инициировать запуск сервера на узле при первом обращении клиента.
- *Балансирование нагрузки* — при репликации серверов на нескольких узлах, реестр возвращает клиенту адрес одной из реплик в соответствии с определенной стратегией балансирования нагрузки.

Репликация серверов в IceGrid реализуется на уровне объектных адаптеров. Объектные адаптеры нескольких серверов могут быть объединены в один «виртуальный» объектный адаптер или *группу репликации (replica group)*. При разрешении косвенного прокси, содержащего в качестве имени объектного адаптера идентификатор группы репликации, реестр возвращает клиенту транспортные точки одного или нескольких адаптеров группы. Способ выбора этих объектных адаптеров собственно и составляет стратегию балансирования нагрузки. IceGrid поддерживает следующие стратегии:

- Случайный выбор.
- Адаптивная стратегия, возвращающая адаптеры на наименее загруженных машинах. Для этого реестр использует информацию о загрузке машин, которую ему периодически отправляют узлы.

- Циклическая (round-robin) стратегия, возвращающая наиболее давно использованные адаптеры.
- Упорядоченная стратегия, возвращающая объектные адаптеры в соответствии с их приоритетом.

По умолчанию клиент связывается с реестром для разрешения косвенного прокси только один раз, при первом использовании прокси. Все дальнейшие вызовы идут напрямую, используя полученную от реестра адресную информацию. В Ice также предусмотрен механизм управления локальным кэшем сервиса имен, позволяющий клиенту периодически инициировать обновление адресной информации.

Помимо реализации службы имен, реестр IceGrid предоставляет Slice-интерфейсы для обнаружения объектов и динамического мониторинга состояния системы. Первый интерфейс позволяет клиентам осуществлять поиск объектов по их идентификатору, типу или группе репликации. Второй интерфейс позволяет клиентам получать уведомления о различных событиях, происходящих в системе, таких, например, как включение-выключение узлов или появление новых объектов.

Наконец, реестр IceGrid реализует *механизм аллокации ресурсов*, обеспечивающий координацию доступа к объектам и серверам IceGrid-приложений. Данный механизм позволяет клиенту получить монопольный доступ к некоторому объекту или серверу. Для этого клиент должен пройти аутентификацию и создать *IceGrid-сессию*, после чего клиент может осуществлять резервирование объектов через интерфейс сессии. Сервер считается зарезервированным после аллокации первого резервируемого объекта на сервере. С помощью сессии клиент может освобождать неиспользуемые ресурсы. Клиент также обязан периодически обновлять время жизни сессии. IceGrid автоматически освобождает ресурсы при закрытии сессии клиентом или истечении времени жизни сессии.

Приложение добавляется в IceGrid путем *размещения (deployment)* его описания в реестре. Описание или *дескриптор* приложения включает следующую информацию:

- серверы, из которых состоит приложение, с указанием их размещения на узлах;
- объектные адаптеры, создаваемые серверами;
- размещенные на объектных адаптерах известные и резервируемые объекты;

- группы репликации, с указанием входящих в каждую группу объектных адаптеров.

Для упрощения описания множества идентичных серверов в IceGrid предусмотрен механизм *шаблонов (templates)*. В состав IceGrid входят готовые шаблоны для сервера IceBox и сервисов IceStrom, Glacier2 и IcePatch2.

Дескриптор приложения может быть размещен в реестре несколькими способами:

- при помощи команды, считывающей дескриптор в XML-формате;
- при помощи графического интерфейса администратора;
- программным образом через API IceGrid.

Также допускается последующая модификация размещенного в IceGrid приложения с помощью приведенных выше средств.

Наиболее удобным средством управления инфраструктурой IceGrid является графический интерфейс администратора *IceGrid Admin*, поддерживающий выполнение следующих действий:

- просмотр списка узлов и серверов с информацией об их состоянии;
- конфигурация, запуск и остановка серверов;
- создание, просмотр, размещение и модификация дескрипторов приложений.

IceGrid решает проблему установки и обновления серверных приложений на нескольких машинах с помощью сервиса IcePatch2. С каждым приложением или сервером можно связать определенные директории на IcePath2-сервере. При размещении приложения узлы автоматически связываются с IcePatch2-сервером и загружают дистрибутив приложения. В дальнейшем с помощью средств администрирования IceGrid можно инициировать обновление дистрибутива приложения на всех задействованных узлах.

2.6. Freeze

Ice имеет встроенный механизм *персистентности объектов (object persistence)*, названный *Freeze*. Freeze позволяет легко организовать хранение состояния объекта в базе данных следующим образом. Разработчик описывает хранимое объектами состояние с помощью Slice и запускает специальный *Freeze-компилятор*. Данный компилятор гене-

рирует высокоуровневый код, который осуществляет запись и считывание состояния объекта из базы данных. Непосредственное взаимодействие с базой данных скрыто от разработчика. В качестве базы данных используется СУБД Berkeley DB.

Дополнительный набор средств *FreezeScript* упрощает обслуживание созданных с помощью Freeze баз данных и миграцию их содержимого при внесении изменений в хранимые структуры данных.

3. Сравнение с другими технологиями

Наибольший интерес представляет сравнение Ice с технологией CORBA, послужившей прототипом для Ice, и Web-сервисами, являющимися сегодня наиболее часто используемым ППО. Все эти технологии поддерживают несколько языков программирования и различные семейства операционных систем. За рамками настоящего сравнения остались технологии Java RMI и.NET, не поддерживающие явно гетерогенные системы.

3.1. Сравнение с CORBA

Ice использует много идей из CORBA. Главным образом это связано с тем, что авторами Ice являются специалисты, принимавших ранее участие в составлении спецификаций CORBA и разработке известных CORBA-пакетов. Имея за спиной многолетний опыт работы с CORBA, авторы Ice поставили цель разработать технологию, которая с одной стороны опирается на проверенные временем решения, а с другой — устраняет недостатки, присущие спецификациям CORBA, упрощает разработку распределенных приложений и привносит новую функциональность, не имеющую аналогов в CORBA.

3.1.1. Отличия в объектной модели

В отличие от CORBA-объекта, Ice-объект может реализовывать несколько интерфейсов, так называемых *фасетов*. Это повышает гибкость архитектуры системы. Например, с помощью фасетов можно расширять функциональность сервера, не нарушая работу существующих клиентов.

В отличие от CORBA, прокси Ice (аналогичные объектным ссылкам CORBA) не имеют скрытую от разработчика структуру. Это означает, что клиент может создать прокси без участия каких-либо других

компонентов системы. Для этого требуется знать тип и идентификатор объекта, и, возможно, физический адрес сервера (не требуется для косвенных прокси). Это устраняет необходимость в использовании специальных сервисов и API для инициализации клиентов, а также различных форматов для строкового представления объектных ссылок. Ice использует единый формат для строкового представления прокси, доступный для чтения человеком.

В объектной модели Ice предполагается, что идентификаторы объектов являются универсально уникальными. Это позволяет избежать коллизий имен при миграции и интеграции серверных приложений. Идентификаторы Ice-объектов являются *строгими (strong)*: клиент может проверить, указывают ли два прокси на один объект, с помощью локальной операции. В случае CORBA, для этого требуются удаленные вызовы, что снижает эффективность работы приложений.

3.1.2. Отличия в функциональности

Спецификации CORBA содержат многое из функциональности Ice. Однако многие из этих спецификаций либо являются необязательными для реализации, либо не реализованы в большинстве CORBA-пакетов. Поэтому зачастую трудно найти один пакет, реализующий всю необходимую функциональность. В этом плане Ice выгодно отличается от решений на основе CORBA,

Кроме того, Ice содержит функциональность, не имеющую прямых аналогов в CORBA:

- Асинхронная диспетчеризация вызовов.
- Полноценный механизм безопасности с поддержкой межсетевых экранов.
- Возможности протокола: поддержка UDP, инкапсуляция данных и поддержка пересылки сообщений без их распаковки.
- Отображения в языки C#, Visual Basic и PHP.

3.1.3. Отличия в простоте использования

Общепризнанным фактом является сложность освоения и использования CORBA. На сложность спецификаций CORBA повлияли, в первую очередь, особенности процесса их стандартизации в рамках консорциума, а также отсутствие эталонной реализации. В результате неизбежных консенсусов между участниками консорциума, в спецификациях оказалось много избыточной или вовсе бесполезной функциональ-

ности, которая никогда не была реализована [4]. Сложность спецификаций привела к сложности API и реализаций.

Одним из примеров сложности использования CORBA является необходимость получения объектных ссылок через специальный сервис именования вследствие их непрозрачности. В случае Ice, клиент может создать прокси без использования каких-либо внешних сервисов. Другим примером является сложность и неэффективность отображения CORBA IDL в C++. Отображение Slice в C++ является более простым и эффективным в использовании.

В отличие от CORBA, разработка Ice велась небольшой группой экспертов. Авторы Ice руководствовались, в первую очередь, соображениями простоты использования и реальной востребованности той или иной функциональности. Это позволило создать технологию одновременно более простую в использовании и более мощную, чем CORBA. По сути, единственными преимуществами CORBA являются статус общепризнанного стандарта и наличие независимых реализаций. Но стоит ли это дополнительных усилий по освоению технологии и времени, потраченного на разработку и отладку приложений?

3.2. Сравнение с Web-сервисами

Наиболее широко используемым и известным сегодня промежуточным ПО являются Web-сервисы. В первую очередь это связано с активной деятельностью крупных ИТ-компаний, «продвигающих» Web-сервисы как ключевую технологию для реализации сервис-ориентированной архитектуры. Web-сервисы также являются сейчас основной технологией для реализации крупномасштабных распределенных вычислительных сред типа Grid. В то время как использование Web-сервисов для B2B-интеграции может иметь свои достоинства, Web-сервисы не являются идеальной платформой для распределенных вычислительных сред по следующим причинам.

3.2.1. Отсутствие объектной модели

Web-сервисы не имеют объектной модели и не поддерживают объектно-ориентированную разработку распределенных приложений. Утверждается, что Web-сервисы не имеют внутреннего состояния (stateless). В то же время в распределенных вычислительных системах часто встречается потребность в реализации сервисов с внутренним состоянием (stateful). Отсутствие явной поддержки внутреннего состояния

привело к появлению спецификаций OGSI и WSRF [14], используемых в сегодняшних Grid-технологиях. Данные спецификации задним числом привносят в Web-сервисы хорошо известные объектно-ориентированные шаблоны, такие как Фабрика [15].

Объектно-ориентированный подход, используемый Ice и CORBA, позволяет реализовывать сервисы с внутренним состоянием или сервисы-фабрики привычным современному программисту образом.

3.2.2. Сложный язык описания интерфейсов

Применяемый в Web-сервисах язык описания интерфейсов WSDL гораздо сложнее в использовании, чем Slice или CORBA IDL. За счет использования XML-формата, WSDL-интерфейс очень трудно создавать вручную. Это привело к появлению дополнительного инструментария, который упрощает процесс спецификации интерфейса или же генерирует WSDL-интерфейс автоматически по коду серверного приложения. Последний подход противоречит общепринятой методологии разработки гетерогенных распределенных систем, когда интерфейсы компонентов описываются первоначально на платформонезависимом языке. Этот же подход (так называемый contract first approach) рекомендуют эксперты по сервис-ориентированной архитектуре [10].

В отличие от WSDL, язык Slice позволяет легко создавать и читать интерфейсы без каких-либо дополнительных средств.

3.2.3. Неэффективный протокол

Вследствие использования XML-формата, применяемый в Web-сервисах протокол SOAP является медленным и неэффективным в сравнении с двоичным протоколом Ice. Выигрыш протокола Ice по задержке и пропускной способности может достигать нескольких порядков. Стоит также учитывать, что разбор XML-сообщений требует дополнительных вычислительных ресурсов. Вследствие использования XML-сообщений, SOAP также неэффективно использует сетевой трафик. Эти особенности Web-сервисов могут оказаться критичными при реализации крупномасштабных распределенных вычислений.

Благодаря использованию двоичного протокола с поддержкой сжатия данных Ice обеспечивает заметно более высокую производительность сетевых вызовов.

3.2.4. Нестабильность спецификаций

Большое число частично пересекающихся и постоянно изменяющихся WS-* спецификаций, большинство из которых, за исключением

самых базовых, пока не являются общепринятыми стандартами, затрудняет изучение и использование Web-сервисов. Кроме того, как показывает опыт CORBA, создание спецификаций большими консорциумами без учета практического опыта реализации систем может приводить к отрицательным результатам.

В отличие от Web-сервисов, Ice имеет стабильные, хорошо документированные отображения в языки, API и протокол.

3.2.5. Отсутствие стандартных отображений в языки программирования

В отличие от Ice и CORBA, для Web-сервисов не существует спецификаций отображения в языки программирования. Каждый разработчик WS-платформы использует проприетарные API и средства. Это означает, что клиентские и серверные приложения не могут быть перенесены без каких-либо изменений с одной платформы на другую. При использовании автоматической генерации WSDL, переход с одной платформы на другую может привести к изменению WSDL и, как следствие, нарушению работы существующих клиентов.

3.2.6. Выводы

Главными преимуществами Web-сервисов над Ice являются широкая поддержка ИТ-индустрии, стандартизация и наличие множества независимых реализаций. Полученный результат напоминает сравнение Ice с CORBA. Однако в данном случае сравниваемые технологии находятся гораздо дальше друг от друга в плане архитектуры, модели программирования и принципов реализации. Описанные выше недостатки Web-сервисов вызывают, на фоне возможностей Ice, ощущение незрелости технологии и даже сомнение в правильности выбранного сторонниками Web-сервисов пути.

Безусловно, сервис-ориентированная архитектура является полезным стилем реализации гетерогенных распределенных систем. Однако это не синоним Web-сервисов, а архитектура, которая может быть реализована с использованием различных технологий, в том числе Ice.

Безусловно, интероперабельность является важным требованием, и XML часто используется для интеграции гетерогенных систем. Но это не объясняет необходимость использования XML-протокола вместо передачи XML-данных (тогда, когда это действительно нужно) в эффективном двоичном формате. Использование протокола SOAP для реализации RPC-подобных вызовов вызывает наибольшие сомнения.

Безусловно, наличие общепринятых стандартов означает возможность выбора из нескольких реализаций и независимость от одного производителя. Однако процесс стандартизации Web-сервисов имеет много общего со стандартизацией CORBA. Появившиеся в большом количестве спецификации усложняют использование Web-сервисов и создание интероперабельных реализаций. Но если в случае CORBA спецификации обеспечивали переносимость приложений, то в случае Web-сервисов переход на реализацию другого производителя сопряжен с модификацией приложения.

Заключение

Описанная в данной статье технология Ice обладает рядом достоинств, делающих ее привлекательной для реализации распределенных научных приложений и вычислительных сред:

- Простота и удобство использования в сравнении с другими технологиями ППО.
- Объектно-ориентированный стиль программирования.
- Поддержка многих популярных языков программирования и операционных систем.
- Возможность использования различных протоколов на транспортном уровне.
- Реализация всего спектра моделей взаимодействия: синхронных, асинхронных и односторонних вызовов, а также рассылки сообщений.
- Поддержка многопоточных клиентов и серверов.
- Поддержка шифрования с открытым ключом и протокола SSL для реализации безопасного взаимодействия клиентов и серверов.
- Наличие дополнительных сервисов с богатой функциональностью.

Немаловажным обстоятельством является доступность исходного кода Ice под лицензией GNU General Public License (GPL), что позволяет свободно использовать технологию в некоммерческих научных проектах.

Отдельного комментария заслуживает вопрос об использовании Ice в распределенных вычислительных средах типа Grid. Дело в том, что стандартом «де факто» при построении подобных систем сейчас являются Web-сервисы.

Конвергенция Grid-технологий и Web-сервисов началась в 2001 году, когда разработчиками инструментария Globus Toolkit [16] и компанией IBM была предложена сервис-ориентированная архитектура Grid (Open Grid Services Architecture, OGSA) [17], основанная на Web-сервисах. Отличительной особенностью данной архитектуры является поддержка сервисов с внутренним состоянием. В OGSA вошла такая функциональность, как фабрики и управление жизненным циклом сервисов, доступ к информации о сервисе, регистрация сервисов и рассылка уведомлений. Первая попытка реализовать OGSA в виде спецификации Open Grid Services Infrastructure (OGSI) оказалась неудачной, поскольку OGSI шла вразрез с направлением развития спецификаций Web-сервисов. В 2004 году при участии IBM и других крупных компаний были подготовлены новые спецификации под общим названием Web Services Resource Framework (WSRF) [14], которые были более благосклонно приняты ИТ-индустрией. Спецификации WSRF были реализованы в четвертой версии инструментария Globus Toolkit на языках Java, C++ и Python. Существует еще несколько реализаций WSRF, в частности на платформе.NET. В 2006 году организация OASIS одобрила спецификацию WSRF 1.2 в качестве стандарта. Однако в том же году компании HP, IBM, Intel и Microsoft объявили о намерении объединить свои, конкурирующие спецификации, в результате чего на смену WSRF должны прийти новые спецификации WS-ResourceTransfer (WS-RT) [18]. По всей видимости, это, наконец, приведет к появлению общепризнанного стандарта. Однако грядущий переход на новые спецификации повлечет за собой необходимость в портировании существующих приложений.

Таким образом, с начала 2000-х годов Grid-технологии фактически являются заложником общей незрелости Web-сервисов и «политических» игр крупных компаний, лоббирующих свои интересы в новых спецификациях. С учетом горького опыта CORBA, это во многом объясняет сложность освоения Grid-технологий и в целом низкий уровень их использования в повседневной практике научных исследований.

Grid-системы первого поколения, такие как Large Hadron Collider Grid (LCG), ориентированы на крупные научные проекты и виртуальные организации с большим числом участников. Сейчас происходит формирование второго поколения Grid-систем, которое обобщает идеи совместного использования вычислительных ресурсов на широкой круг ресурсов и приложений, предлагая универсальную инфраструктуру для научной кооперации [19, 20]. Данная инфраструктура базиру-

ется на сервис-ориентированном подходе: пользователи преобразуют свои ресурсы в удаленно доступные сервисы, которые могут быть обнаружены и использованы другими пользователями в своих приложениях для решения задач. Под ресурсами здесь подразумеваются не столько суперкомпьютеры и хранилища данных, сколько различные научные приложения, средства анализа данных, приборы и т. п. Системы второго поколения, за счет расширения круга ресурсов и приложений, должны обеспечивать функционирование множества виртуальных организаций среднего и малого размера. При этом требуется радикально упростить процедуры развертывания соответствующего ПО, формирования виртуальных организаций, разработки сервисов и их размещения в Grid. Сложность освоения существующих Grid-технологий сужает круг потенциальных разработчиков сервисов. Отсутствие удобных средств разработки также увеличивает время, необходимое для трансформации существующего ресурса в сервис.

В силу описанных достоинств технологии Ice и проблем существующих Grid-технологий, представляется, что использование Ice в качестве основы для построения Grid-систем второго поколения может оказаться гораздо эффективнее, чем применение технологий на основе Web-сервисов. Данный вопрос будет являться предметом дальнейших исследований.

Литература

1. *Эммерих В.* Конструирование распределенных объектов. М.: Мир, 2002.
2. CORBA / http://www.corba.org
3. Object Management Group / http://www.omg.org
4. *Henning M.* The Rise and Fall of CORBA // ACM Queue. Vol. 4. No. 5 — June 2006: 28–34.
5. DCOM Technical Overview, Microsoft / http://msdn2.microsoft.com/en-US/library/ms809340.aspx
6. Java RMI, Sun Microsystems / http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp
7. .NET Framework Developer Center, Microsoft / http://msdn.microsoft.com/netframework.
8. Web Services Activity, W3C / http://www.w3.org/2002/ws
9. Web Services Architecture, W3C Working Group Note, 2004 / http://www.w3.org/TR/ws-arch
10. *Thomas Erl.* Service-Oriented Architecture: Concepts, Technology, and Design. Prentice Hall, 2005.

11. *Henning, M.*: A New Approach to Object-Oriented Middleware // IEEE Internet Computing. 2004. Vol. 08. No 1. 66–75.
12. *Michi Henning, Mark Spruiell.* Distributed Programming with Ice. ZeroC, Inc. Revision 3.2.1, August 2007 / http://www.zeroccom/download/Ice/3.2/Ice-3.2.1.pdf
13. ZeroC, Inc / http://www.zeroc.com
14. Web Services Resource Framework v1.2, OASIS / http://www.oasis-open.org/committees/wsrf
15. *Гамма Э., Хелм Р., Джонсон Р., Влиссидес Д.* Приемы объектно-ориентированного проектирования. Паттерны проектирования. СПб., Питер, 2001.
16. Globus Toolkit / http://globus.org/toolkit
17. *Foster I., Kesselman C., Nick J. M., and Tuecke S.* The Physiology of the Grid. An Open Grid Services Architecture for Distributed Systems Integration. Technical report, Open Grid Service Infrastructure WG, Global Grid Forum, June 2002 / http://globus.org/alliance/publications/papers/ogsa.pdf
18. WS-ResourceTransfer, IBM / http://www-128.ibm.com/developerworks/library/specification/ws-wsrt
19. *Foster, I.* Service-Oriented Science // Science. 2005. Vol. 308. May 6.
20. *Foster, I.* Scaling eScience Impact. 1st Iberian Cyberinfrastructure Conference, Santiago de Compostela, Galicia, Spain, May 15, 2007 / http://www-fp.mcs.anl.gov/~foster/Talks/Scaling%20eScience%20IBERGrid.pdf