УДК 004.415.532

Д. В. Кадашев [1], А. А. Кузнецов [2], Д. В. Савенко [3], В. Е. Тютюньков [4]

Новосибирский государственный университет
ул. Пирогова, 2, Новосибирск, 630090, Россия
E-mail: [1] dkadashev@gmail.com, [2] localstorm@gmail.com,
[3] prankish@gmail.com, [4] tyutyunkov@gmail.com

СИСТЕМА РАСПРЕДЕЛЕННОГО UNIT-ТЕСТИРОВАНИЯ «TESTING GRID»

Введение

Unit-тест представляет собой специальным образом организованный программный компонент, созданный исключительно в целях контроля корректности работы другого программного компонента. Исполнение unit-тестов при модификации программных компонентов имеет большое значение для поддержки работоспособности разрабатываемой программной системы и называется unit-тестированием. Unit-тестирование является важнейшей составляющей гибких методологий разработки программного обеспечения, таких как «экстремальное программирование».

Компания, занимающаяся разработкой программного обеспечения, как правило, имеет большой парк достаточно мощных компьютеров (рабочих станций разработчиков), которые простаивают значительную часть времени. Поэтому целесообразно использовать свободное машинное время для проведения unit-тестирования. Для этого потребуется объединить неоднородную среду, состоящую из компьютеров различной производительности и работающих под управлением различных ОС в единый вычислительный ресурс. Для достижения этой цели можно использовать GRID-технологии.

Суть GRID-технологии [Foster, Kesselman, 1999] состоит в том, что обычные компьютеры, объединенные в обычную сеть, превращаются с помощью программного обеспечения в единый вычислительный ресурс, способный решать сложные вычислительные задачи. Таким образом, GRID-системы представляют собой неоднородный вычислительный ресурс с некоторым количеством серверов управления.

Цель проекта «Testing Grid» – разработка системы распределенного unit-тестирования с использованием некоторых подходов GRID. Основное предназначение данной системы – автоматизация процесса unit-тестирования в рамках подразделения компании-разработчика программного обеспечения.

Несмотря на значительный интерес к GRID-технологиям, который способствовал их быстрому развитию и появлению различных GRID-решений, на сегодняшний день не существует готового решения, предназначенного для проведения распределенного unit-тестирования. С целью изучения пригодности существующих GRID-решений для распределенного unit-тестирования было принято решение о создании прототипа системы на основе популярного GRID-инструментария BOINC, обеспечивающего распределенное исполнение тестов JUnit [Morris, 2003]. Однако впоследствии было установлено, что существующие GRID-инструментарии не пригодны для целей unit-тестирования.

В результате проведенного исследования было установлено, что множество особенностей архитектуры системы BOINC [1] продиктовано необходимостью обработки запросов от очень большого количества экземпляров программ-агентов, необходимостью минимизации объемов передаваемой информации (при частоте отправки результатов вычислений порядка 1 000 шт./час), невозможностью сохранения постоянной связи с агентами, а также истори-

[1] BOINC Software Development. University of California, 2006. http://boinc.berkeley.edu/boinc_dev.php; BOINC Web (RPCS). University of California, 2006. http://boinc.berkeley.edu/web_rpc.php.

ческими причинами. В системе BOINC исполняемый код загружается один раз, а данные, обрабатываемые этим кодом, обновляются часто.

При проведении распределенного unit-тестирования мы имеем дело с относительно небольшим количеством экземпляров программ-агентов, однако для успешной работы нам необходимо каждый раз передавать агенту значительные объемы информации (последнюю сборку тестируемого программного модуля).

При более детальном рассмотрении были идентифицированы следующие недостатки системы BOINC:

– равноправие клиентов системы. В случае проведения unit-тестирования всех пользователей, не являющихся администраторами, можно разделить на разработчиков (авторов тестируемого кода) и добровольных участников, предоставляющих машинное время;

– отсутствие возможности получения расширенной информации о конфигурации клиентской системы, что не позволяет выяснить, способен ли клиент выполнить тестирование; в случае же провала тестирования, невозможно проанализировать параметры системы участника GRID-сети;

– невысокое качество реализации программных модулей, выражающееся в регулярных отказах, причину которых установить не удалось.

После того как стало очевидно, что применение GRID-инструментария BOINC нецелесообразно, было принято решение о создании оригинальной системы распределенного unit-тестирования.

После четырех месяцев интенсивной разработки была выпущена бета-версия системы «Testing Grid», общая структура которой приведена на рис. 1.

Межмодульные интерфейсы системы

Разрабатывая распределенную систему, разработчик решает задачу интеграции нескольких ресурсов. Качество предлагаемого решения в значительной мере зависит от внешних интерфейсов каждого компонента распределенной системы, поэтому мы уделили значитель-

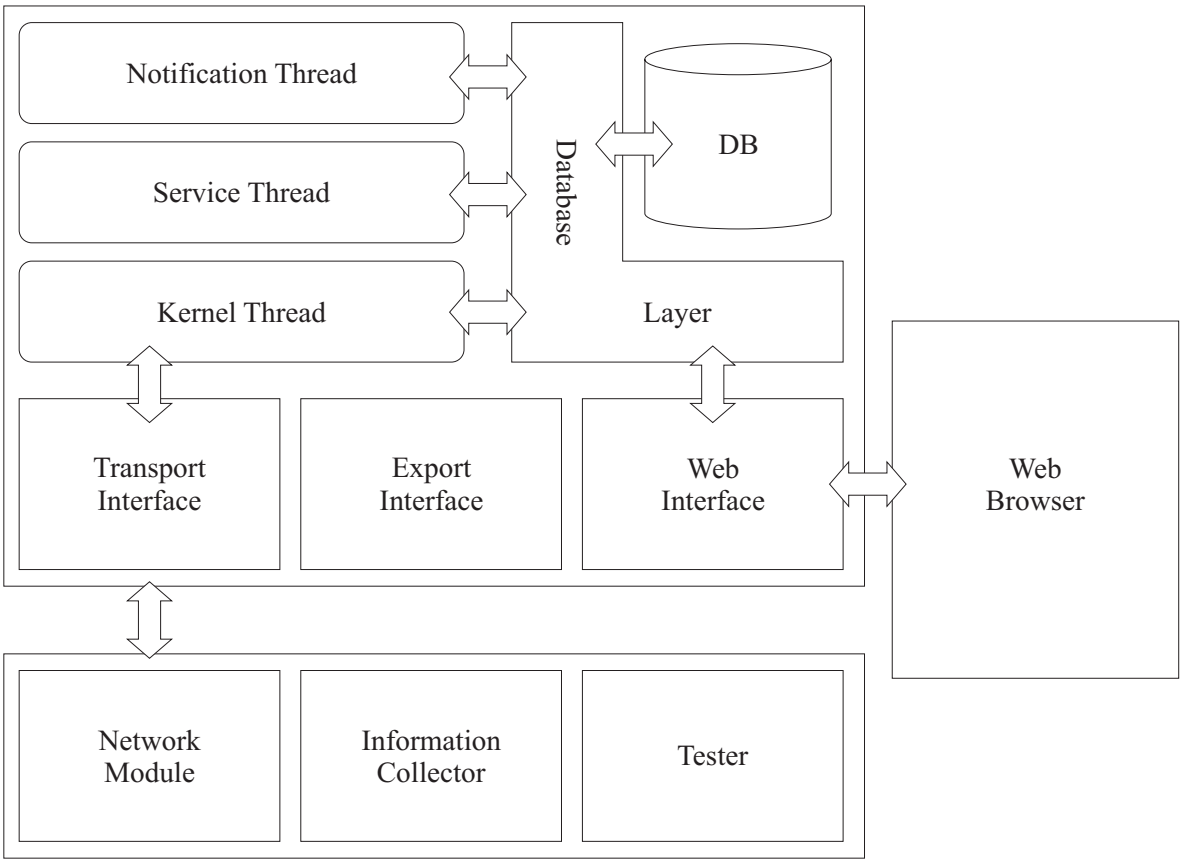

Рис. 1. Структура системы «Testing Grid»

ное внимание их проектированию. Ниже будут описаны интерфейсы компонентов, объяснены причины, по которым были приняты те или иные технологические решения.

При проектировании интерфейсов системы была поставлена задача сохранения удачных решений, использующихся в большинстве известных GRID-систем:

– использование протокола HTTP для коммуникации между клиентским и серверным приложениями;

– использование XML как универсального средства обмена структурированной информацией;

– наличие криптографического механизма защиты системы от вредоносного кода;

– наличие Web-интерфейса для управления системой.

В системе BOINC были использованы различные способы организации внешних интерфейсов системы, в том числе сценарии CGI, PHP, Python. Такая неоднородность создавала дополнительные проблемы уже при разработке прототипа системы, в частности осложняла поддержку системы.

В качестве альтернативы мы использовали Java-сервлеты как универсальное и очень гибкое средство для создания как web-интерфейса, так и интерфейса взаимодействия клиента и сервера с использованием протокола HTTP в качестве протокола транспортного уровня. Использование Java позволяет также повысить переносимость, снизить стоимость разработки и дальнейшего развития системы.

Внешние интерфейсы системы «Testing Grid» можно разделить на три части: web-интерфейс пользователя, транспортный интерфейс и сервисный интерфейс. Под сервисным интерфейсом мы будем понимать интерфейс, с помощью которого администратор получает информацию о состоянии системы в виде, пригодном для автоматизированного анализа. Интерфейс взаимодействия клиента и сервера мы будем называть транспортным интерфейсом системы.

Язык XML является универсальным средством обмена структурированной информацией в распределенной системе, именно поэтому XML играет ключевую роль в системе «Testing Grid». Использование XML позволило нам существенно упростить проверку корректности взаимодействий, отделить оформление web-интерфейса от его реализации, а также уменьшить вероятность успешного осуществления таких атак на web-интерфейс системы, как XSS.

Транспортный интерфейс

Первоначально в качестве основного протокола связи было решено использовать стандартный протокол SOAP[2], который широко используется в различных GRID-системах. Но после анализа последствий применения такого технологического решения мы были вынуждены отказаться от этой идеи, в силу того что все известные реализации данного протокола при обработке запроса загружают данные в оперативную память целиком, что не является эффективным решением ввиду большого объема передаваемых данных.

Транспортный интерфейс системы «Testing Grid» реализован с помощью набора Java-сервлетов, каждый из которых обрабатывает определенный тип запросов, поступающих от клиентских приложений. Запросы клиентских приложений представляют собой POST-запросы протокола HTTP 1.1, содержащие данные в формате XML, имеющие определенный, специфический для данного типа запроса, MIME-тип. Поступивший запрос перенаправляется для последующей обработки соответствующему Java-сервлету в зависимости от MIME-типа данных, содержащихся в теле запроса.

При разработке формата запросов оригинального протокола мы использовали идеи, лежащие в основе двух популярных протоколов: XML-RPC [Winner, 1999] и SOAP[3]. Ключевым отличием разработанного протокола от упомянутых выше протоколов является отсутствие необходимости сохранения в оперативной памяти значительных объемов двоичных данных на этапе приема и отправки сообщения. Нам удалось избавиться от необходимости отправки и приема одновременно текстовых и двоичных данных.

Организация транспортного уровня системы «Testing Grid» представлена на рис. 2. Поступивший запрос проходит несколько стадий обработки:

– разбор XML-содержимого с помощью DOM-parser;

[2] SOAP Version 1.2. W3C Recommendation, 2003. http://www.w3.org/TR/soap.

[3] SOAP With Attachments API for Java (SAAJ). Sun Developer Network. http://java.sun.com/xml/downloads/saaj.html.

– проверка корректности запроса с помощью XML Schema [Vlist, 2002];

– извлечение данных из разобранного XML-содержимого с помощью вычисления XPath-выражений [Simpson, 2002];

– обработка извлеченных данных и формирование ответа сервера.

Данная процедура обработки запроса позволяет надежно идентифицировать некорректные запросы до извлечения содержащихся в запросе данных. Использование XPath-выражений позволяет существенно упростить извлечение данных из сложных XML-документов.

Web-интерфейс

Web-интерфейс системы «Testing Grid» также реализован с помощью Java-сервлета, который принимает все поступающие GET и POST запросы.

Поступивший запрос передается обработчику, соответствующему префиксу адреса запрашиваемого ресурса. Обработчик запроса возвращает либо адрес нового ресурса для последующей переадресации запроса, либо данные в формате XML. Если обработчик вернул адрес нового ресурса, сервлет web-интерфейса выполняет переадресацию клиента средствами протокола HTTP 1.1. В противном случае полученный от обработчика XML-документ подвергается XSL-трансформации[4], результатом которой является HTML-документ, отправляемый клиенту.

Данный подход имеет ряд существенных преимуществ перед непосредственной генерацией HTML-кода. Во-первых, это отсутствие необходимости учета особенностей представления данных при формировании результата обработки запроса. Это позволяет изменять представление, не изменяя код системы. Во-вторых, при трансформации XML-документа происходит автоматическое экранирование данных, которые могут иметь особый смысл в HTML, что позволяет уменьшить вероятность успешного осуществления XSS-атак на систему «Testing Grid». В то же время к недостаткам данного подхода следует отнести достаточно высокие требования к производительности web-сервера, на котором развернут web-интерфейс системы.

Функции ядра

Ядро системы выполняет в основном распределительные и административные функции. Перед более подробным описанием ядра следует отметить, что хотя система реализована с помощью Java-сервлетов, наиболее важные части ядра работают постоянно (в виде пото-

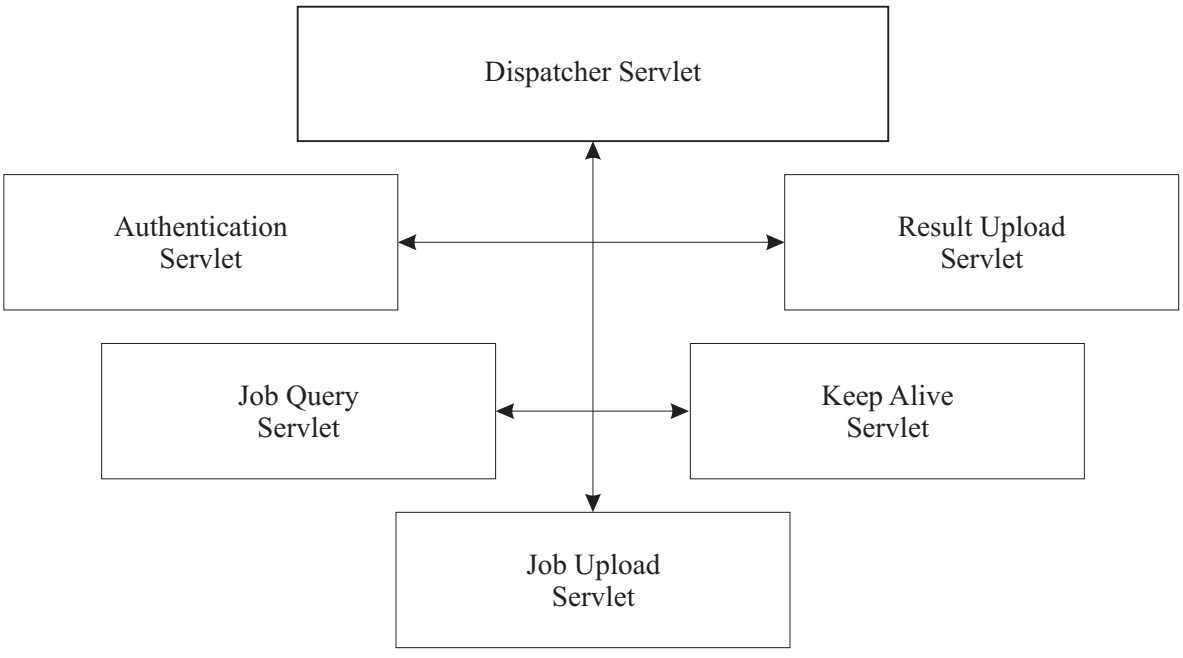

Рис. 2. Структура транспортного интерфейса системы

[4] XSL Transformations (XSLT) Version 1.0. W3C Recommendation, 1999. http://www.w3.org/tr.xslt.

ков), а не только от запроса к запросу. Это существенно отличает всю систему от традиционных web-приложений, основанных на Java-сервлетах. Ядро системы «Testing Grid» ответственно за следующее:

– аутентификация и авторизация пользователей и клиентов (класс Authenticator);

– обработка keep-alive сообщений клиентов (класс ClientHandler);

– получение результатов работы от клиентов (класс ClientHandler);

– рассылка уведомлений пользователям (класс Notifier и поток NotificationThread);

– регулярное выполнение сервисных операций над базой данных (поток ServiceThread).

Однако основная и самая сложная задача ядра – распределение заданий между клиентами. Она разбивается на несколько подзадач и решается разными модулями ядра:

– пакет checker содержит классы, реализующие проверку соответствия системных требований задания с конфигурацией клиента. Здесь заложен механизм гибкого добавления новых объектов конфигурации, не предусмотренных системой изначально, однако необходимых пользователю;

– пакет jobgiver содержит классы, реализующие стратегии планирования графиков работы клиентов;

– класс ClientHandler принимает результаты работы от клиентов, также регистрирует в базе данных клиентов, которые отрапортовали о готовности принять работу;

– поток KernelThread регулярно проверяет БД на наличие готовых к работе клиентов и проводит для них поиск заданий, пользуясь средствами пакетов checker и jobgiver.

Процесс обработки сообщений клиентов

Далее будет описан штатный процесс обработки сообщения, поступившего от клиента, а также работа ядра системы на примере одного сообщения.

Итак, подготовленное транспортным уровнем для обработки сообщение от клиента поступает либо в класс Authenticator, если клиент еще не зарегистрирован в системе, либо в класс ClientHandler, если клиент уже зарегистрирован.

Authenticator представляет собой класс-обертку для классов, работающих с СУБД, он сопоставляет поступившую информацию о пользователе (имя пользователя, пароль) с тем, что имеется в БД, и в случае нахождения данного пользователя регистрирует в БД сессию для клиента и возвращает параметры этой сессии, которые будут переданы клиенту (идентификатор сессии и некоторую служебную информацию).

ClientHandler определяет разные реакции на разные типы сообщений пользователя.

Если пришло сообщение типа keep-alive со статусом Waiting, то это означает, что клиент бездействует и ожидает работы. В этом случае клиент заносится в базу данных исполнителей, ожидающих работы. Через какой-то промежуток времени «просыпается» KernelThread и обрабатывает эту базу (этот процесс будет описан позже). Клиенту возвращается статус NoWork. Блок-схема алгоритма распределения задания изображена на рис. 3.

Если же обнаруживается, что клиент уже занесен в базу данных и ему уже поставлено какое-то задание, возвращается статус WorkAvailable, после которого клиент должен отправить дополнительный запрос на получение предназначенной ему работы. Этот запрос также обрабатывается классом ClientHandler. Необходимо заметить, что задание состоит из двух частей: формального описания на языке XML (важные характеристики теста, необходимые для выполнения) и собственно бинарных данных, представляющих собой код теста. Поэтому тест возвращается клиенту в два этапа: сначала текстовая информация, затем непосредственно код.

После выполнения теста клиент посылает отчет, который также обрабатывается классом ClientHandler (проводятся некоторые дополнительные вычисления, например для определения удачного или неудачного статуса задания в целом по данным об успешных или неудачных результатах отдельных его тестов – так называемое построение канонического результата). После этого результат записывается в базу данных.

Далее необходимо описать работу KernelThread, который и осуществляет поиск заданий для клиентов. KernelThread – это поток, который активируется через определенные проме-

жутки времени, получает список клиентов, ожидающих заданий, и пытается найти для них подходящие тесты из базы данных невыполненных тестов.

Для определения того, что тест может быть выполнен на клиенте, используется пакет Checker, содержащий проверяющие классы для всех типов параметров конфигурации. Этот пакет может быть легко расширен пользователем для добавления специфических параметров.

Для распределения заданий (планирования, составления расписания) используется пакет jobgiver.

Другие функции ядра

База данных системы требует периодических сервисных операций, которые осуществляются потоком ServiceThread. Активируясь через определенные промежутки времени, этот поток:

– удаляет некоторые служебные записи о заданиях с истекшим сроком давности;
– удаляет сессии с истекшим сроком давности;
– выполняет некоторые другие, менее значительные обслуживающие операции.

В системе предусмотрена возможность уведомления пользователей о событиях системы. Эта функциональность осуществляется потоком NotificationThread, периодически проверяющим базу данных на предмет наличия неотправленных сообщений пользователям и рассылающим их.

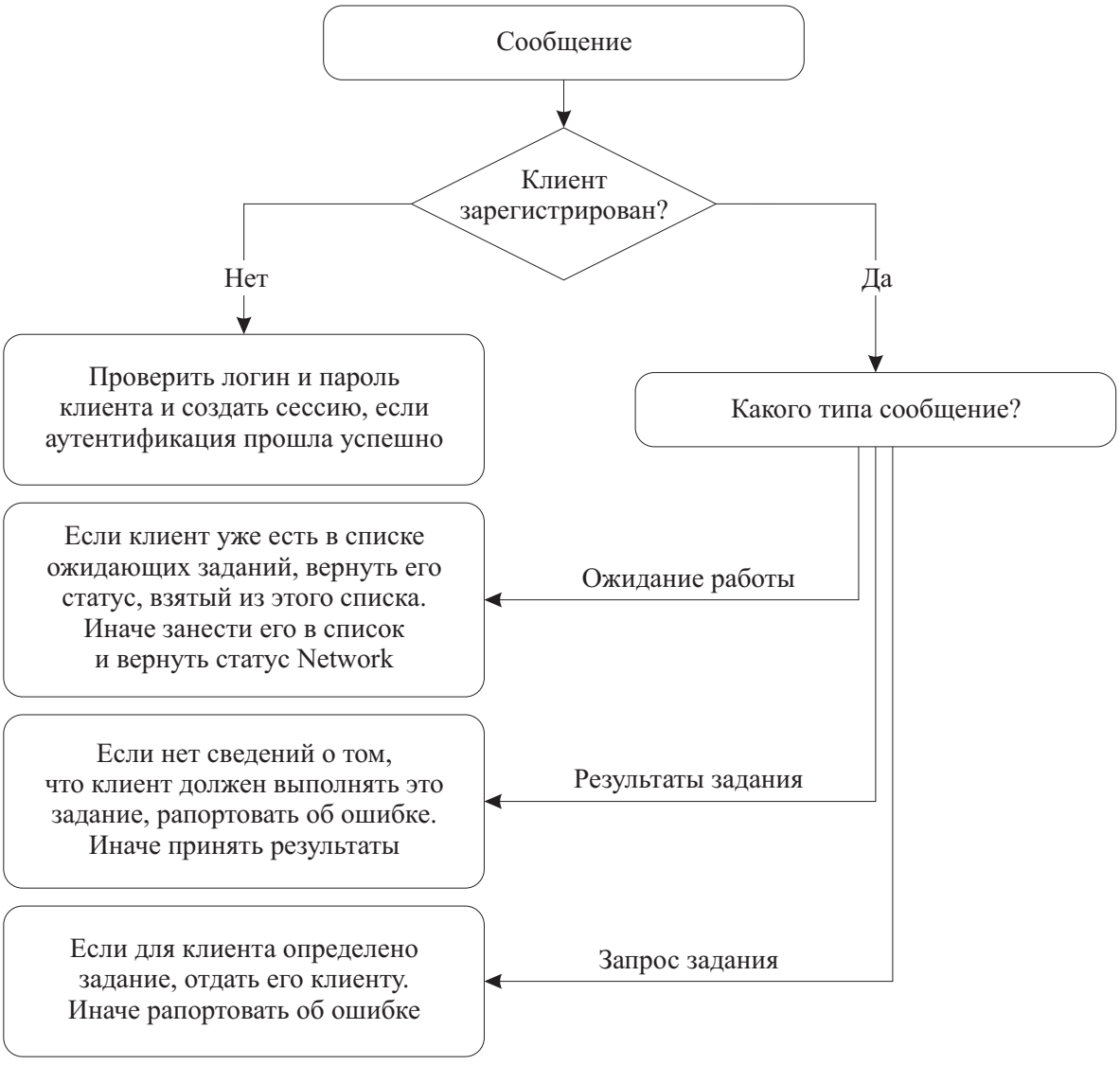

Рис. 3. Порядок обработки запроса клиента

Организация агента системы

Агент системы состоит из сетевого модуля и набора тестирующих модулей для различных типов тестов.

Сетевой модуль агента системы состоит из ядра, к которому подключаются модули расширения (например, для возможности удаленного управления агентом). Сетевым модулем в зависимости от типа задания запускается нужный тестирующий модуль.

Принцип работы

Агент системы состоит из сетевого модуля и набора тестирующих модулей для различных типов тестов.

После запуска агента он считывает конфигурационный файл (имя этого файла передается в качестве аргумента), затем осуществляется инициализация модулей расширения и агент переходит в штатное рабочее состояние.

Далее осуществляется подключение к серверу, указанному в конфигурационном файле, используя имя пользователя и пароль, указанные там же.

В случае недоступности сервера или неверного ответа от него, попытки соединиться повторяются через некоторый заданный интервал времени.

В случае если учетная запись, с которой проводилась аутентификация, заблокирована или у этой записи недостаточно прав для проведения тестирования, агент завершает работу с соответствующим сообщением.

В случае успешной аутентификации агент получает от сервера дополнительные настройки, специфичные для пользователя (например, интервалы рабочего времени в каждый день недели).

Затем серверу регулярно отправляются keep-alive-пакеты, содержащие состояние агента. В случае если текущее состояние клиента означает его готовность выполнять работу, сервером подбирается и отправляется подходящее задание. После получения задания сетевой модуль анализирует его описание и запускает нужный тестирующий модуль в зависимости от типа задания.

После завершения тестирования серверу отправляются результаты, полученные от тестирующего модуля.

Сетевой модуль агента системы состоит из ядра, к которому подключаются модули расширения (например, для возможности удаленного управления агентом). Сетевым модулем в зависимости от типа задания запускается нужный тестирующий модуль.

Ключевые особенности агента системы

Агент Testing Grid имеет следующие особенности.

Взаимодействие с сервером посредством оригинального протокола, основанного на XML, поверх протоколов HTTP или HTTPS.

Поддержка JUnit 4.1. Система JUnit широко применяется сообществом разработчиков для проведения unit-тестирования Java-кода.

Поддержка Java 1.5 и 1.6. Агент системы тестировался на JRE версий 1.5 и 1.6. Более ранние версии Java не поддерживаются, так как в системе используются типизированные контейнеры – нововведение Java 1.5, кроме того, в JUnit 4.1 применяются аннотации, так же появившиеся в Java 1.5.

Тестирующий модуль запускается в отдельном процессе, и коммуникация с ним осуществляется через сетевое соединение, что защищает агента системы от краха в случае критической ошибки в тестируемом коде.

Возможность установить ограничение по времени на тестирование каждого задания. При превышении ограничения, в случае если тестирование не закончено, тестирующий модуль прекращает тестирование, задание считается выполненным неуспешно, тест, на котором произошло превышение допустимого времени, считается «проваленным» по тайм-ауту, и результаты тестирования сообщаются сетевому модулю.

Гибкая система требований к системе, на которой будет осуществляться выполнение конкретного задания с возможностью добавления пользовательских типов требований. Каждому заданию может быть поставлен набор требований к системе, на которой будет осуществляться тестирование. Для сбора данных о системе в агенте встроены стандартные сборщики (информация о JDK и ОС), кроме того, присутствует возможность установки произвольных пользовательских сборщиков. Сборщик информации – это произвольный класс, реализующий интерфейс ModuleInfoCollector. Для установки дополнительного сборщика достаточно добавить имя класса в конфигурационный файл агента и в classpath.

Возможность удаленного управления агентом системы, получения статуса. Агент принимает сетевые соединения и далее ждет управляющих команд. В текущей версии можно приостановить работу агента, возобновить ее, получить статус, завершить работу агента. Если работа агента приостановлена, то он продолжает отсылать keep-alive-пакеты, но задания не запрашивает.

Возможность задания недельного расписания работы агента. Для каждого пользователя, зарегистрированного на сервере, можно задать желаемые интервалы работы агента в различные дни недели. При соединении с сервером с заданным именем пользователя агент получает расписание для этого пользователя. Во время, не попадающее в рабочие интервалы, агент остается на связи с сервером, отсылает keep-alive-пакеты, но не запрашивает работу.

Возможность запроса заданий только с определенной меткой. Каждому заданию в системе может быть поставлена определенная метка – строковый идентификатор. Агент может запрашивать задания только с определенной меткой, игнорируя остальные, даже если у них выше приоритет. Это может использоваться для создания выделенной машины, которая будет использоваться для тестирования какой-то группы заданий.

Аутентификация клиентских приложений происходит на сервере по имени пользователя и паролю, что позволило ввести разграничение прав доступа.

Заключение

Разработана бета-версия системы распределенного unit-тестирования, которая в настоящий момент готова для передачи заказчику.

Несмотря на то что развертывание системы «Testing Grid» возможно на компьютерах под управлением ОС Windows, Linux, Sun Solaris, в настоящее время бета-версия системы доступна в виде RPM-пакетов для ОС Fedora Core 5.

Для работы серверной части системы потребуется контейнер Java-сервлетов Tomcat 5.5, СУБД MySQL 4.0 или более поздняя, JRE 1.5. Ведется работа над поддержкой СУБД PostgreSQL.

Список литературы

Foster I., Kesselman C. The Grid: Blueprint for a New Computing Infrastructure. Morgan Kaufmann Publishers, 1999.

Morris D. JUnit Automates Java Testing // Midrange Programmer, 2003. http://www.itjungle.com/mpo/mpo110603-story01.html.

Simpson J. E. XPath and XPointer. Locating Content in XML Documents. O'Reilly, 2002.

Vlist E. XML Schema. The W3C's Object Oriented Descriptions for XML. O'Reilly, 2002.

Winner D. XML-RPC Specification. UserLand Software, 1999. http://www.xmlrpc.com/spec.

Материал поступил в редколегию 20.04.2007