Ордена Ленина

ИНСТИТУТ ПРИКЛАДНОЙ МАТЕМАТИКИ
имени М.В.Келдыша
Российской академии наук

С.А.Богданов, В.Н.Коваленко,
Е.В.Хухлаев, О.Н.Шорин

Метадиспетчер: реализация средствами метакомпьютерной системы Globus

Москва
2001 г.

УДК 519.68

С.А.Богданов, В.Н.Коваленко, Е.В.Хухлаев, О.Н.Шорин. **Метадиспетчер: реализация средствами метакомпьютерной системы Globus.**

Рассмотрен проект реализации Метадиспетчера средствами метакомпьютерной системы Globus. Метадиспетчер – система управления заданиями в гетерогенной вычислительной Сети, объединяющей группы компьютеров, находящиеся под управлением систем управления пакетной обработкой. Приведен обзор средств Globus. Описаны функциональность и архитектура Метадиспетчера, включающая спул заданий, интерпретатор команд, планировщик и монитор заданий. Рассмотрен вопрос подготовки заданий с учетом гетерогенности.

Ключевые слова: вычислительная Сеть, управление заданиями, планирование.

S.A.Bogdanov, V.N.Kovalenko, E.V.Huhlaev, O.N.Shorin. **The Metadispatcher: the Implementation by Globus Metacomputing Infrastructure Toolkit.** – Preprint, Keldysh Inst. Appl. Mathem., Russia Academy of Science, 2001.

The Metadispatcher implementation by Globus Metacomputing Infrastructure Toolkit is proposed. The Metadispatcher is job management system at heterogeneous computational Grid. The Grid jointes several computer groups. Every computer group is managed by some Batch Management System. Globus toolkits review is presented. Functonality and architecture of the Metadispatcher are described, including jobs spool, command interpretator, sheduler and job monitor. A problem of job preparation is considered taking the heterogeneity.

Key words and phrases: computational Grid, job management, sheduling.

Работа выполнена при финансовой поддержке Российского фонда фундаментальных исследований (проект № 99-01-00389) и Министерства промышленности, науки и технологий РФ (гос. Контракт № 203-3(00)-П).

Содержание

1. Введение

Под *Метадиспетчером* мы понимаем систему управления заданиями в гетерогенной вычислительной Сети. Концепция вычислительной Сети (или Grid) [1] становится все более популярной, позволяя потенциально получить за счет программных решений и с помощью коммерческого сетевого оборудования очень большие вычислительные мощности, намного превосходящие те, которыми располагают современные суперкомпьютерные архитектуры, из географически распределенных ресурсов. Роль и место Метадиспетчера в контексте задачи организации распределенных вычислений рассмотрены в статье [2]. В данной работе освещается способ его реализации в виде надстройки над базовым программным аппаратом – метакомпьютерной системой Globus [3].

Проект Метадиспетчера исходит из представления о Сети как о совокупности *узлов.* Узел – группа компьютеров (в локальной сети), находящаяся под управлением локального монитора ресурсов, в качестве которого сейчас используются различные системы управления пакетной обработкой (СУПО) [4]. Совокупность узлов в вычислительной Сети объединена некоторой общей информационной инфраструктурой, позволяющей планировать размещение заданий на вычислительных ресурсах узлов Сети. Однако каждый узел обладает автономией и предоставляет в распоряжение Сети только ту часть своих вычислительных ресурсов и на таких условиях, которые приемлемы для управляющей этим узлом СУПО. СУПО различных типов (PBS, LSF, Condor и др.) во многом близки по предоставляемым пользователям возможностям запуска и управления вычислительными заданиями. Функциональность, предоставляемая пользователю Метадиспетчера, близка к функциональности СУПО. По существу, Метадиспетчер – это СУПО более высокого уровня.

При проектировании Метадиспетчера необходимо (помимо многих других) решить вопросы создания информационной инфраструктуры, аутентификации и авторизации, безопасного доступа к распределенно хранимым программам и данным, унифицированного безопасного запуска и управления заданиями в разнородных локальных мониторах ресурсов. В принципе в любой СУПО почти все эти вопросы так или иначе решаются, но главное отличие Метадиспетчера от СУПО заключается в том, что в *закрытой* локальной сети (в которой работает СУПО) вполне допустимы сравнительно простые подходы к их решению, непригодные в *открытой* Интернет-сети. Например, вопросы авторизации в СУПО решаются, как правило, так, что каждый пользователь просто регистрируется под одним и тем же именем на всех компьютерах, находящихся под управлением СУПО. Пользователю достаточно войти под своим именем на один из компьютеров локальной сети, чтобы получить беспрепятственный доступ (с правами, предоставленными одноименному пользователю) ко всем остальным. Понятно, что в открытой

Интернет-сети такой способ не обеспечивает должного уровня безопасности и противоречит принципу автономии узлов.

Метакомпьютерная системая Globus была выбрана в качестве базовой при разработке Метадиспетчера, потому что (помимо всего прочего) в ее рамках создан значительный задел в этих направлениях. Кроме “готовых к употреблению” комплексов программ Globus предлагает набор API (поддержанных соответствующими библиотеками), позволяющих реализовать собственные комплексы или модифицировать уже готовые. Кроме того, в Globus доступны все исходные тексты компонент и API.

Далее в работе описываются средства Globus, используемые в нашей реализации Метадиспетчера, и рассмотрены организация Метадиспетчера, потоки информации и управление заданиями. Вопросы планирования размещением заданий здесь не рассматриваются.

2. Обзор некоторых средств Globus

Globus предоставляет средства информационного обеспечения (MDS), инфраструктуру безопасности (GSI), удаленного доступа к даным (GASS), унифицированного удаленного запуска и управления заданиями (GRAM).

2.1. Информационная инфраструктура Globus

Основу информационной инфраструктуры Globus составляет метакомпьютерная информационная служба MDS (Metacomputing Directory Service) [5, 6], базирующаяся на службе директорий LDAP. Описание схемы MDS и организации сбора и актуализации информации можно найти в [7]. Начиная с версии 1.1.3, в Globus предусмотрена распределенная модель GRIS – GIIS, при которой на центральном узле вычислительной Сети работает LDAP-сервер Сети (Grid Information Index Server - GIIS), собирающий данные с локальных LDAP-серверов (Grid Resource Information Service - GRIS), работающих на отдельных узлах Сети. Периодичность, состав и конкретные способы сбора информации могут меняться в зависимости от потребностей данной Сети. Например, в GIIS может быть в основном сосредоточена *статическая* информация о конфигурации Сети, обновляемая при каждом ее изменении (подключение, удаление или изменение состава любого узла). *Динамическая* же информация о состоянии узла (загрузка и пр.) может оперативно отражаться в GRIS этого узла.

2.2. Инфраструктура безопасности Globus

Инфраструктура безопасности Globus (Globus Security Infrastructure – GSI) [8] основана на реализации протокола SSL (Secure Sockets Layer) [9], использующего алгоритм шифрования RSA [10] с *открытым* и *закрытым* ключами. Эти ключи всегда генерируются в паре. Документ, зашифрованный одним ключом из пары, можно расшифровать только с помощью парного. Эта простая фундаментальная идея лежит в основе всех приемов проверки и

удостоверения подлинности. Открытый ключ передается открыто вместе с документом. Закрытый ключ хранится секретно и ни в коем случае не передается по сети. Для удостоверения подлинности документа применяется *электронная подпись* – это свертка (хэш-функция) документа, зашифрованная с помощью закрытого ключа. Правильно подписать документ может только владелец закрытого ключа. Для проверки правильности подписи достаточно открытого ключа. Эта проверка заключается в расшифровке подписи и сравнении результата со сверткой документа.

Аутентификация основана на предъявлении X.509 *сертификата* [11], включающего идентификатор предъявителя и его открытый ключ, и подписанного Центром Сертификации (Certificate Authority - CA). Идентификатор должен однозначно характеризовать предъявителя в сети. Использование сертификата по протоколу SSL возможно только тогда, когда его предъявителю доступен закрытый ключ. Предъявитель должен доказать, что он обладает закрытым ключом (для этого нужно зашифровать им присланный другой стороной контакта тестовый документ). Поэтому перехват злоумышленником сертификата не имеет никакого смысла.

Для получения сертификата нужно сгенерировать пару ключей (открытый и закрытый), закрытый сохранить у себя, а открытый включить в заявку на сертификат (неподписанный сертификат), которая пересылается выбранному CA. CA подписывает (разумеется, если считает это возможным) сертификат и пересылает его обратно. Сертификат имеет достаточно продолжительный срок действия (например, год). Закрытый ключ хранится в отдельном файле и для повышения секретности может быть дополнительно зашифрован с помощью *пароля* (pass phrase), который требуется вводить вручную при каждом предъявлении сертификата.

При любом безопасном контакте (например, при запуске задания или при получении удаленного доступа к файлу) обе стороны (или только одна, если взаимная аутентификация не требуется) обязаны предъявить свои сертификаты и доказать, что они обладают соответствующими закрытыми ключами. Сертификат считается достоверным (а аутентификация успешной), если принимающей стороне известен подписавший его CA и он подписан правильно. Для передачи секретной информации каждая сторона контакта шифрует ее с помощью открытого ключа, извлеченного из сертификата другой стороны, но шифрование не является обязательным и в системе Globus не используется.

При сетевых контактах, как правило, клиент предъявляет не основной сертификат, а т.н. *прокси-сертификат* (proxy - заменитель). Прокси-сертификат подписывается владельцем сертификата и включает в себя также исходный сертификат, что позволяет удостовериться в его достоверности. Он обладает меньшей секретностью, поскольку его закрытый ключ хранится вместе с ним в одном файле и не защищен паролем, но по этой же причине

удобнее в использовании. Поэтому прокси-сертификат выписывается, как правило, на небольшое ограниченное время (несколько дней).

Прокси-сертификаты необходимы при передаче прав владельца сертификата по сети для выполнения некоторых действий от его имени (*делегирование)*, с тем, чтобы сохранить в секрете (не передавать по сети) закрытый ключ. Делегирование сводится к тому, что владелец сертификата подписывает и отправляет (точно так же, как и CA) другой стороне контакта присланную ему заявку на прокси-сертификат, содержащую идентификатор и открытый ключ. При делегировании можно ограничить круг полномочий прокси-сертификата, запретив использовать его для запуска процессов. Владелец делегированного прокси-сертификата в свою очередь может делегировать права (подписать прокси-сертификат следующего уровня) и т.д.

Все вышеописанные процедуры поддерживаются реализованным в Globus интерфейсом GSSAPI [12], который используют все программы, нуждающиеся в аутентификации. Для получения сертификата и ручного выписывания прокси-сертификата имеются интерактивные утилиты.

Предъявление прокси-сертификата сводится к помещению его в файл с обусловленным именем или к занесению имени файла сертификата в переменную среды X509_USER_PROXY. Список известных CA, необходимый для удостоверения подлинности сертификатов, хранится в файле с обусловленным именем.

Основная схема авторизации для контакта “клиент-сервер” в Globus заключается в том, что клиент авторизуется на сервере, если в файле авторизации (grid-mapfile) его идентификатору (извлекаемому из сертификата) поставлено в соответствие имя локального пользователя ОС. Вообще говоря, сервер выполняет запрос авторизованного клиента, пользуясь правами этого пользователя. Если идентификатор клиента в файле авторизации отсутствует, то клиент считается не авторизованным и запрос отвергается.

2.3. Удаленный доступ к внешней памяти

В рамках Globus разработана система удаленного доступа к внешней памяти (Global Access to Secondary Storage – GASS) [13]. Она упрощает запуск приложений в среде Globus. При ее использовании нет необходимости вручную пересылать файлы с помощью **ftp** или устанавливать распределенную файловую систему.

Для обеспечения удаленного доступа к файлам на хосте, где они размещены, запускается GASS-сервер, работающий по протоколам **http** (анонимный доступ), **https** (взаимная аутентификация по протоколу SSL с предъявлением X.509 сертификатов). GASS-сервер по протоколу **https** может обслуживать только клиентов, предъявляющих сертификат с точно таким же идентификатором, что и в сертификате сервера. Поэтому для каждого пользователя должен быть запущен персональный экземпляр GASS-сервера, а в URL клиент обязан указывать конкретный порт именно этого GASS-сервера.

В состав GASS входят ряд API и утилит, поддерживающих как сторону клиента (включая протокол **ftp** с анонимным доступом), так и сторону сервера.

В настоящее время идет работа по разработке серверов (а также клиентских API и утилит), поддерживающих протокол **gridftp** [14] - расширение протокола **ftp**, обеспечивающее нормальное серверное обслуживание в рамках GSI (один сервер для всех авторизованных в grid-mapfile клиентов).

API удаленного доступа к файлам (GASS File Access) включает функции открытия и закрытия файлов **globus_gass_open**, **globus_gass_close**, **globus_gass_fopen**, **globus_gass_fclose** с точно такими же параметрами, что и стандартные функции UNIX **open**, **close**, **fopen**, **fclose**. Единственное отличие состоит в том, что помимо обычного имени файла может быть URL, включающий протокол доступа и адрес удаленного хоста с портом. В результате открытия создается локальная копия удаленного файла, которая после закрытия отправляется обратно (если необходимо). После открытия файла с ним можно работать обычными функциями UNIX. Если файл открыт в режиме добавления (O_APPEND), то добавления немедленно отражаются в удаленном файле.

Утилиты **globus-gass-server** и **globus-gass-server-shutdown** позволяют запускать и останавливать персональный GASS-сервер по протоколу **http** или **https**.

API встроенного GASS-сервера (GASS server EZ) позволяет запускать GASS-сервер, выполняющийся в процессе, вызвавшем функцию API. Помимо всего прочего, этот сервер может направлять файлы в стандартные потоки вывода собственного процесса (stdout, stderr). Такой сервер особенно удобен для разработки программ запуска и управления удаленными заданиями. Именно такой механизм используется в утилите **globusrun** (см. п. 2.4.2) для перемещения (staging) и перехвата файлов.

Утилита **globus-url-copy** позволяет копировать удаленные файлы, идентифицированные URL GASS-сервера.

Утилита **globus-rcp** копирует удаленные файлы, идентифицированные адресом хоста и локальным именем файла. На хосте-приемнике автоматически средствами GRAM (см. п. 2.4) запускается GASS-сервер, после чего на хосте-источнике выполняется **globus-url-copy** с URL этого GASS-сервера. Можно копировать директорию целиком (tar) и сжимать файлы при передаче (gzip/gunzip).

Существенным недостатком этих утилит является необходимость иметь двойной запас дисковой квоты пользователя на хосте, где запускается **globus-url-copy**, поскольку последняя всегда создает локальную копию удаленного файла.

2.4. GRAM

GRAM (Globus Resource Allocation Manager) [15] обеспечивает унифицированный интерфейс для удаленного запуска заданий и управления ими в среде Globus. Пользователю предоставлены ряд клиентских утилит запуска и управления, а также API для создания собственных приложений, управляющих ресурсами на базе GRAM.

Под управлением одного GRAM может находится несколько *сервисов*, каждый из которых либо непосредственно запускает процессы задания на собственном компьютере GRAM (сервис типа fork), либо передает его некоторой СУПО (например, сервис типа pbs передает задание PBS). В последнем случае GRAM играет роль шлюза, а дальнейшая судьба задания определяется правилами, принятыми в конкретной СУПО. Как правило, сервис типа fork используется в качестве вспомогательного технического средства при обслуживании сервисов других типов и GASS, хотя в принципе его разрешается использовать и для вычислительных заданий.

Запрос на выполнение задания включает контактную строку GRAM и описание задания на языке определения ресурсов RSL (Resource Specification Language). Контактная строка задает адрес хоста и порт для контакта с GRAM, а также имя сервиса, которому должно быть поручено обслуживать задание.

Вообще говоря, язык RSL предназначен для обмена информацией о ресурсах (запрашиваемых или имеющихся) между всеми компонентами метакомпьютерной среды Globus. На RSL принято также выражать некоторые не относящиеся к ресурсам характеристики заданий,. В языке RSL имеются операторы дизъюнкции (‟|‟) и конъюнкции (‟&‟), применяемые к последовательности *параметров*, заключенной в круглые скобки. Параметр состоит из *имени* параметра, *оператора сравнения* (‟=‟ ‟>‟ ‟<‟ ‟>=‟ ‟<=‟ ‟!=‟) и *строки значения*. Допускаются также т.н. множественные запросы (multyrequest), объединенные оператором (‟+‟). GRAM не принимает произвольную строку RSL. В задании для GRAM не должно быть множественных запросов, а также дизъюнкций. Разрешена только конъюнкция. GRAM понимает только ограниченный список имен стандартных параметров (все остальные отвергаются). Приведем основные стандартные параметры:

(executable = *value*) – определяет выполняемый файл задания;

(directory = *value*) – задает рабочую директорию для задания;

(arguments = *value* ...) – задает параметры командной строки ;

(environment = (*var value*)...) – задает среду выполнения задания;

(stdin = *value*) – задает файл стандартного ввода;

(stdout = *value*) – задает файл стандартного вывода;

(stderr = *value*) – задает файл вывода ошибок;

(count = *value*) – задает число процессов (экземпляров выполнения executable);

(queue = *value*) – задает имя очереди СУПО.

Файлы могут быть специфицированы не только локальными путями (в этом случае предполагается, что они располагаются на компьютере GRAM), но и URL (в этом случае они доставляются с помощью GASS). Файлы executable и stdin доставляются на компьютер GRAM перед выполнением задания, а после выполнения их локальные копии удаляются. Стандартные потоки вывода и ошибок направляются в файлы, заданные параметрами stdout и stderr, динамически во время выполнения задания. В частности, средствами GASS (задав URL специального вида) их можно направить в стандартные потоки процесса, запустившего задание в GRAM с помощью API, или буферизовать их на компьютере GRAM.

2.4.1. Архитектура GRAM

Архитектура GRAM показана на рис. 1. Клиентское приложение с помощью API посылает запрос на выполнение задания в GRAM. Запрос принимает *gatekeeper* (привратник) - процесс, постоянно выполняющийся на хосте GRAM и слушающий соответствующий порт. После выполнения взаимной аутентификации (gatekeeper имеет собственный сертификат) проводится авторизация: определяется *имя локального пользователя ОС*, с правами которого запускаются все процессы сервиса, указанного в контактной строке. Сервис может быть сконфигурирован так, что либо это имя всегда одно и то же (задается в файле конфигурации сервисов gatekeeper'а), либо оно извлекается из файла авторизации Globus (grid-mapfile) в соответствии с идентификатором клиента. В первом случае идентификатор клиента все равно должен быть указан в grid-mapfile, но соответствующий ему локальный пользователь может быть фиктивным (не зарегистрированным в файле паролей). Это позволяет ограничить непосредственный доступ пользователей к хосту GRAM.

Для каждого задания gatekeeper запускает отдельный процесс *менеджера задания* (job manager), который существует до окончания задания. Ему передаются описание задания и делегированный сертификат. Код стандартного менеджера задания один и тот же независимо от типа сервиса. В функции менеджера задания входит доставка файлов задания (executable, stdin), запуск процессов задания, отслеживание состояния задания, выполнение команд пользователя по управлению заданием, обратная доставка файлов (stdout, stderr). Сервис типа fork запускает процессы задания на собственном хосте GRAM и управляет ими непосредственно. Для запуска, отслеживания и управления заданием в локальном мониторе ресурсов выполняется соответствующий интерфейсный скрипт, который и обращается к СУПО, используя команды qsub, qstat и пр. Для каждого типа СУПО пишутся специализированные интерфейсные скрипты, учитывающие его особенности.

Скрипту запуска передаются все стандартные параметры GRAM, извлеченные из строки RSL, описывающей задание. Они используются скриптом для формирования задания для данной СУПО. Поскольку набор стандартных параметров ограничен, то для того, чтобы передать конкретной СУПО специфическую для нее информацию, надо найти какие-то другие способы. В частности, для этого можно использовать переменные среды с согласованными именами, которые понимал бы интерфейсный скрипт.

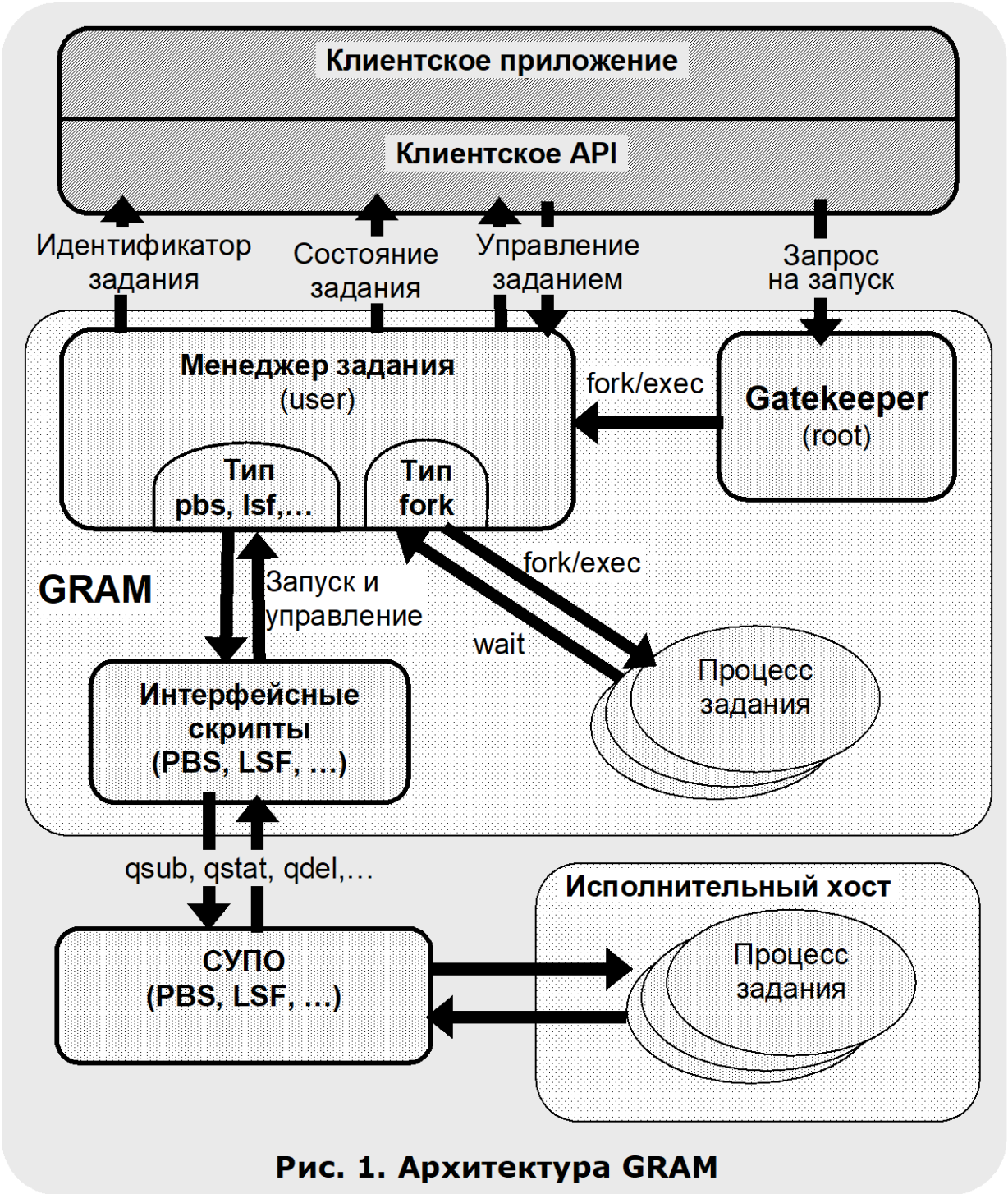

Рис. 1. Архитектура GRAM

Для любого сервиса ссылка на делегированный сертификат устанавливается в переменную среды X509_USER_PROXY, что позволяет запущенным процессам использовать делегированные права. Правда, если задание выполняется на исполнительном хосте, отличном от собственного хоста GRAM, то возникает проблема доступа к временному файлу с делегированным сертификатом по локальной сети.

Когда задание успешно запущено, менеджер задания возвращает клиенту *ярлык* (идентификатор) задания, используемый для дальнейших контактов с ним.

Кроме того, менеджер задания принимает запросы от клиентского API на получение информации о состоянии задания, на снятие задания, на подключение (или отключение) уведомления о состоянии задания. Когда задание меняет свое состояние, менеджер задания отправляет всем подключенным клиентам соответствующее уведомление. Все контакты с клиентом проходят по протоколу https с предъявлением менеджером задания делегированного сертификата. Контакт проходит успешно, если клиент предъявляет сертификат, принадлежащий пользователю, запустившему задание.

2.4.2. Клиентский интерфейс GRAM

Пользователю предоставлены несколько утилит, посредством которых можно запускать задания в GRAM и управлять ими. При обращении к любой из них необходимо иметь действующий прокси-сертификат.

Утилита **globusrun** является основным средством запуска и управления, с помощью которой реализованы все остальные утилиты. В число опций **globusrun** входят контактная строка GRAM и строка описания задания на языке RSL. В *пакетном* режиме утилита возвращает ярлык запущенного задания, используемый для последующего управления им, и отключается, не дожидаясь окончания задания. В режиме *слежения* утилита получает извещения об изменении состояния от менеджера задания и выводит уведомления об этом на терминал. Можно установить режим запуска встроенного GASS-сервера для доставки *перемещаемых* с клиентского хоста или на клиентский хост файлов (executable, stdin, stdout, stderr) или *перехвата* на терминал стандартных потоков для опущенных в строке RSL параметров stdout, stderr. **globusrun** может также выполнять функции получения состояния и принудительного снятия задания, запущенного в пакетном режиме.

Утилита **globus-job-submit** имеет опции для задания основных параметров задания. Она формирует строку описания задания на RSL, исходя из заданных опций, и вызывает globusrun в пакетном режиме, возвращая ярлык задания, предъявляемый при вызове утилит управления. Если опции, задающие stdout, stderr, опущены, то соответствующие файлы буферизуются на GRAM. Их можно получить в любой момент (во время выполнения или по

окончании задания) с помощью утилиты **globus-job-get-output**. Для очистки буферизованных файлов применяется утилита **globus-job-clean**.

Состояние запущенного задания можно узнать посредством утилиты **globus-job-status**.

Задание принудительно снимается утилитой **globus-job-cancel**.

2.4.3. Клиентское API GRAM

В состав API входят функции запуска и управления заданием:

Функция запуска задания **globus_gram_client_job_request** в числе параметров имеет контактную строку GRAM и строку описания задания на языке RSL. В случае успешного запуска она возвращает ярлык задания. При запуске можно задать контактную строку (URL по протоколу https) для получения уведомления об изменении состояния задания. Для генерации такой контактной строки и привязки к ней callback-функции служит функция **globus_gram_client_callback_allow**. Порт, указанный в контактной строке, освобождается функцией **globus_gram_client_callback_disallow**. После запуска к заданию можно привязать любое число callback-функций для получения уведомлений (**globus_gram_client_callback_register**). Состояние запущенного задания возвращается функцией **globus_gram_client_job_status.** Задание принудительно снимается функцией **globus_gram_client_job_cancel.**

3. Функциональность Метадиспетчера

Функциональность Метадиспетчера аналогична функциональности, обеспечиваемой набором клиентских утилит GRAM **globus-job-…**. Команды Метадиспетчера - запуск задания с отключением (**submit**), получение информации о состоянии задания (**status**), снятие задания (**cancel**), получение (**get-output**) и очистка (**clean**) буферизованных файлов - реализуются соответствующими клиентскими утилитами. Команда **submit** возвращает ярлык задания, предъявляемый при вызове остальных команд управления.

Задание для Метадиспетчера включает *фильтр поиска* (на языке LDAP), *строку RSL*, содержащую стандартные параметры GRAM, не включенные в фильтр поиска (такие, как directory, arguments, environment), *ссылки на файлы* (локальные пути или URL) executable, stdin, stdout, stderr.

Фильтр поиска описывает характеристики задания и ресурсы, необходимые для его выполнения. Он используется для нахождения в информационной инфраструктуре Сети узлов, на которых авторизован пользователь и которые потенциально пригодны для выполнения задания.

Приняв задание, Метадиспетчер направит его для выполнения на один из найденных узлов. GRAM, обслуживающий этот узел, будем называть *целевым*.

Файлы executable, stdin могут быть *перемещаемыми* (с клиентского хоста), *локальными* (непосредственно доступными Метадиспетчеру) и *удаленными* (заданными URL). Файлы stdout, stderr могут быть только

удаленными. Если ссылка на stdout, stderr опущена, то соответствующий файл буферизуется в целевом GRAM.

4. Архитектура Метадиспетчера

Рис.2 представляет архитектуру Метадиспетчера.

Организация вычислительной Сети Метадиспетчера базируется на средствах Globus: в каждом узле Сети выделяется шлюзовой компьютер, на котором устанавливается Globus и запускается GRAM. Этот GRAM обслуживает СУПО, управляющую узлом. Информационная инфраструктура Метадиспетчера полностью основана на схеме GIIS – GRIS.

Метадиспетчер – это комплекс программ, выполняющихся на управляющем хосте Сети (хосте Метадиспетчера).

Прием Метадиспетчером команд (в том числе передача по сети, аутентификация и авторизация), доставка файлов, запуск и управление заданиями выполняются средствами Globus, для чего на хосте Метадиспетчера устанавливается Globus и запускается GRAM с сервисом “jobmanager-meta” (типа fork), обслуживающим Метадиспетчер.

Принятые команды выполняются *интерпретатором команд*.

Информация обо всех поступивших заданиях сохраняется в хранилище заданий на диске - *спуле* - и передается *планировщику*. Планировщик, руководствуясь некоторой стратегией и информацией о ресурсах Сети, направляет задание в конкретный узел, обслуживаемый некоторым *целевым* GRAM.

Запуск и отслеживание выполнения задания в целевом GRAM выполняются *монитором задания.* Доставка удаленных файлов выполняется напрямую, минуя хост Метадиспетчера.

4.1. Спул заданий

Доступ к заданиям в спуле осуществляется функциями соответствующего **API (API SPOOL)** по ярлыку задания. Задание в спуле описывается набором **именованных атрибутов**. Значение каждого атрибута – строка. Набор имен расширяемый – API доступа к спулу позволяет определить атрибут с произвольным именем. Такая схема упрощает управление и полностью развязывает руки планировщику.

API SPOOL обеспечивает функции включения, исключения, чтения и записи описания задания с блокировкой. Прежде чем модифицировать описание задания, его необходимо прочитать с блокировкой. Блокировка необходима, чтобы синхронизировать доступ к спулу разных процессов. Блокируется каждое задание по отдельности. Если задание заблокировано, то функция чтения ожидает, когда оно будет разблокировано. Прочитать задание можно и без блокировки (при этом гарантируется целостность задания, т.е., что во время чтения никто его не модифицирует). В API входят также функции

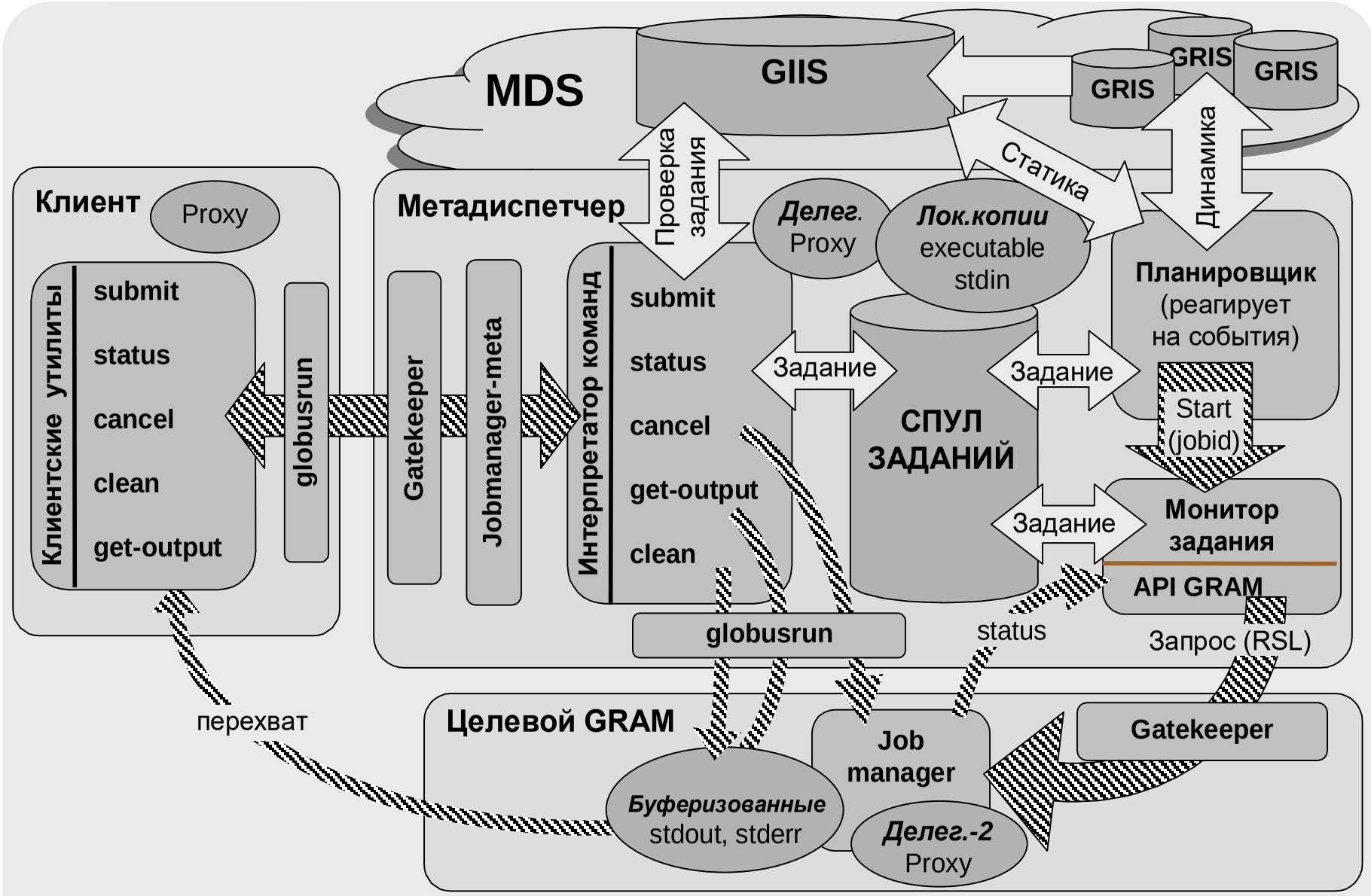

Рис. 2. Архитектура Метадиспетчера

просмотра и модификации атрибутов прочитанного задания, а также функции перебора всех заданий спула.

Задание, хранящееся в спуле, характеризуется следующими основными атрибутами:

- Ярлык задания jobid
- Идентификатор пользователя.
- Строка RSL
- STATUS - состояние задания (возможные состояния задания см. 4.6)
- Имя файла с делегированным прокси-сертификатом
- Ссылки (локальные пути или URL) на файлы executable, stdin, stdout, stderr и их локальные копии
- Список вычислительных ресурсов Сети, пригодных для выполнения задания
- Контактная строка целевого GRAM
- Ярлык задания в целевом GRAM – используется для управления заданием
- PID (идентификатор процесса) монитора задания

4.2. Аутентификация, авторизация и безопасность

Аутентификацией и авторизацией на хосте Метадиспетчера занимается штатный gatekeeper. Все пользователи, которым разрешен доступ к Метадиспетчеру, должны быть прописаны в grid-mapfile. Интерпретатору команд передается делегированный прокси-сертификат, необходимый для доставки файлов клиента и запуска заданий от его имени. В стандартном клиентском API GRAM делегированный сертификат **ограничен (limited)**, т.е. пригоден только для диалога с клиентом по протоколу https и доступа к файлам, принадлежащим клиенту, но не для запуска заданий от его имени. Небольшое исправление клиентского API позволяет передавать **полноправный (full)** сертификат, пригодный и для запуска заданий.

Все ресурсы, директории и файлы, используемые Метадиспетчером, принадлежат исключительно локальному пользователю ОС с именем **metadisp**. Никто другой доступа к ним не имеет. Все процессы Метадиспетчера – jobmanager-meta, интерпретатор команд, планировщик, монитор задания и пр. запускаются под этим пользователем.

Специализированный сервис jobmanager-meta (а не обычный jobmanager-fork) нужен для того, чтобы интерпретатор команд мог одновременно получить доступ и к ресурсам Метадиспетчера (закрытым для всех остальных пользователей хоста), и к делегированному прокси-сертификату. Дело в том, что обычный jobmanager-fork (и любой запускаемый им процесс) выполняется под *авторизованным* локальным пользователем (из grid-mapfile) и поэтому не имеет доступа к ресурсам Метадиспетчера. Чтобы получить такой доступ, интерпретатор команд Метадиспетчера должен был бы иметь флаг SUID. Но в

этом случае он не будет иметь доступа к делегированному прокси-сертификату, принадлежащему исключительно пользователю, под которым запущен менеджер задания. Проблема решается, если jobmanager-meta запустить под *собственным* локальным пользователем, имеющим доступ к ресурсам Метадиспетчера.

Если пользователю разрешено *только* запускать задания в Метадиспетчер, но он не зарегистрирован на хосте Метадиспетчера, то в grid-mapfile ему должен соответствовать некоторый фиктивный локальный пользователь, не зарегистрированный в passwd.

Использование менеджера заданий типа fork порождает проблему, связанную с *ограничением доступа* незарегистрированных в passwd пользователей к ресурсам хоста Метадиспетчера. Стандартный менеджер заданий типа fork позволяет запустить любой выполняемый файл (даже загружаемый через GASS), что совершенно недопустимо, особенно, если пользователь не зарегистрирован. Этого можно избежать, если научить jobmanager-meta всегда (независимо от значения параметра executable) запускать единственную утилиту (интерпретатор команд), на которую и возложить контроль за допустимостью команды Это единственная модификация кода штатного менеджера заданий для обслуживания Метадиспетчера.

4.3. Интерпретатор команд

Прием и выполнение команд выполняются следующим образом:

Клиентская утилита Метадиспетчера работает исключительно как посредник. Она запускает через globusrun **интерпретатор команд**, указывая в контактной строке хост Метадиспетчера, порт gatekeeper'а и сервис "jobmanager-meta", перехватывая stdout, stderr интерпретатора. В первом аргументе интерпретатора передается *имя команды*, а в следующих аргументах - опции клиентской утилиты. Чтобы обеспечить доставку перемещаемых файлов задания и перехват stdout, stderr интерпретатора, globusrun вызывается с запуском встроенного GASS-сервера.

Интерпретация команды сводится к вызову соответствующей имени команды утилиты, которой передаются остальные аргументы интерпретатора. Набор утилит, реализующий команды, можно легко расширять.

По команде **submit** интерпретатор запустит утилиту приема задания, передав ей описание задания (фильтр поиска, строку RSL и ссылки на файлы). Прежде, чем положить задание в спул, утилита проверяет его, обратившись к GIIS с запросом, основанным на фильтре поиска. В спул кладется только задание с синтаксически правильным фильтром поиска, для обслуживания которого в Сети имеются ресурсы, на которых авторизован пользователь. В случае положительного исхода проверки в описание задания включаются дополнительные атрибуты (в частности, содержащие список таких ресурсов), используемые в дальнейшем при планировании. При отрицательном ответе

задание не будет включено в спул, а пользователь получит некоторое сообщение.

Перед включением в спул утилита приема создает локальные копии перемещаемых файлов executable, stdin и собственную копию делегированного прокси-сертификата. Предварительная доставка перемещаемых executable, stdin необходима, поскольку клиентская утилита (тем самым и встроенный GASS-сервер) завершается после приема задания, не дожидаясь реального запуска в *целевой* GRAM. Таким образом, в распоряжении Метадиспетчера оказывается вся информация, необходимая для планирования и дальнейшего запуска задания от имени пользователя. В свой stdout утилита выводит ярлык задания jobid.

Остальные утилиты, реализующие команды управления заданием, получив jobid в своих аргументах, извлекают описание задания из спула и выполняют (посредством globusrun) соответствующие действия:

Утилита, реализующая команду **status**, выводит в свой stdout значение атрибута STATUS.

Утилита, реализующая команду **cancel**, снимает запущенное задание из целевого GRAM.

Утилита, реализующая команду **get-output**, направляет вывод накопленных в буферизованном файле (stdout, stderr) данных напрямую с целевого GRAM в потоки stdout, stderr клиентской утилиты, вызвавшей интерпретатор, минуя хост Метадиспетчера.

Утилита, реализующая команду **clean**, запускает на целевом GRAM утилиту очистки буферизованных файлов и удаляет задание из спула.

4.4. Планировщик

Планировщик – постоянно выполняющийся процесс, реагирующий на события, происходящие в Метадиспетчере. В число событий входят исчерпание временного интервала, поступление нового задания, успешное окончание или ошибка при выполнении запущенного задания, принудительное снятие пользователем запущенного задания и некоторые другие, связанные с изменением состояния заданий (см. п. 4.6). Если событие связано с определенным заданием, то вместе с событием передается и идентификатор этого задания. События обрабатываются **планировщиком** синхронно – пока не закончена обработка одного события, все остальные будут ждать.

Планировщику доступны все задания спула (средствами API SPOOL). Принципы, стратегия и алгоритмы планирования выходят за пределы данной работы и будут рассмотрены нами в другом месте. В своей работе планировщик использует как статическую (GIIS), так и динамическую (GRIS) информацию о ресурсах Сети. Решив, что некоторое задание следует направить в определенный узел Сети, **планировщик** формирует подходящий для СУПО этого узла набор параметров RSL, построенный на основе фильтра поиска, и стартует процесс **монитора задания**, передав ему идентификатор

задания и контактную строку *целевого* GRAM, обслуживающего выбранный узел.

4.5. Монитор задания

Монитор задания запускает задание в целевой GRAM от имени пользователя (предъявив сохраненный делегированный сертификат) и организует слежение за заданием. Все это делается посредством API GRAM. Предварительно выполняется пробный запуск с фиктивным файлом executable и без доставки стандартных файлов. Механизм пробного запуска мы рассматриваем как временную альтернативу отсутствующему пока в Globus механизму резервирования ресурсов. В какой-то мере он гарантирует успешный запуск задания. Если пробный запуск уложился в обусловленное время, то выполняется реальный запуск задания с доставкой или буферизацией файлов. Если же пробный запуск не удался или не уложился во временной лимит, то возбуждается событие, по которому **планировщик** повторяет цикл планирования и находит другой узел Сети для выполнения задания.

Процесс **монитора задания** функционирует до завершения задания. Когда состояние задания изменяется, **монитор задания** получает уведомление и заносит новое значение атрибута STATUS в спул. По завершении задания **монитор задания** удаляет задание из спула (если не было буферизации stdout, stderr), очищает временные локальные файлы и прекращается.

4.6. Жизненный путь задания

На рис.3 показан жизненный путь задания в Метадиспетчере. Вновь поступившее задание находится в состоянии **Waited** до тех пор, пока планировщик не возьмется за него и не переведет в состояние **Sheduling**.

Назначив задание к запуску в целевой GRAM, планировщик переводит его в состояние **Try_Running** и передает монитору. Монитор задания при выполнении пробного запуска переводит его в состояние **Trial_Pending**. Если во время пробного запуска произошла ошибка, то задание переходит в состояние **Trial_Failed**. Если пробный запуск не уложился в заданное время, то задание переводится в состояние **Timeout_Expired**. При неудаче пробного или реального запуска задание переходит в состояние **Running_Failed**.

Состояния **Pending**, **Active**, **Suspended**, **Failed**, **Done** унаследованы от GRAM. Задание находится в состоянии **Pending**, когда оно принято СУПО, но еще не запущено. Задание переходит в состояние **Active**, когда его процессы выполняются. Приостановленное СУПО задание находится в состоянии **Suspended**. Если задание прекратилось в результате ошибки или снятия его пользователем или системой, то оно переходит в состояние **Failed**. При успешном завершении задание переходит в состояние **Done**.

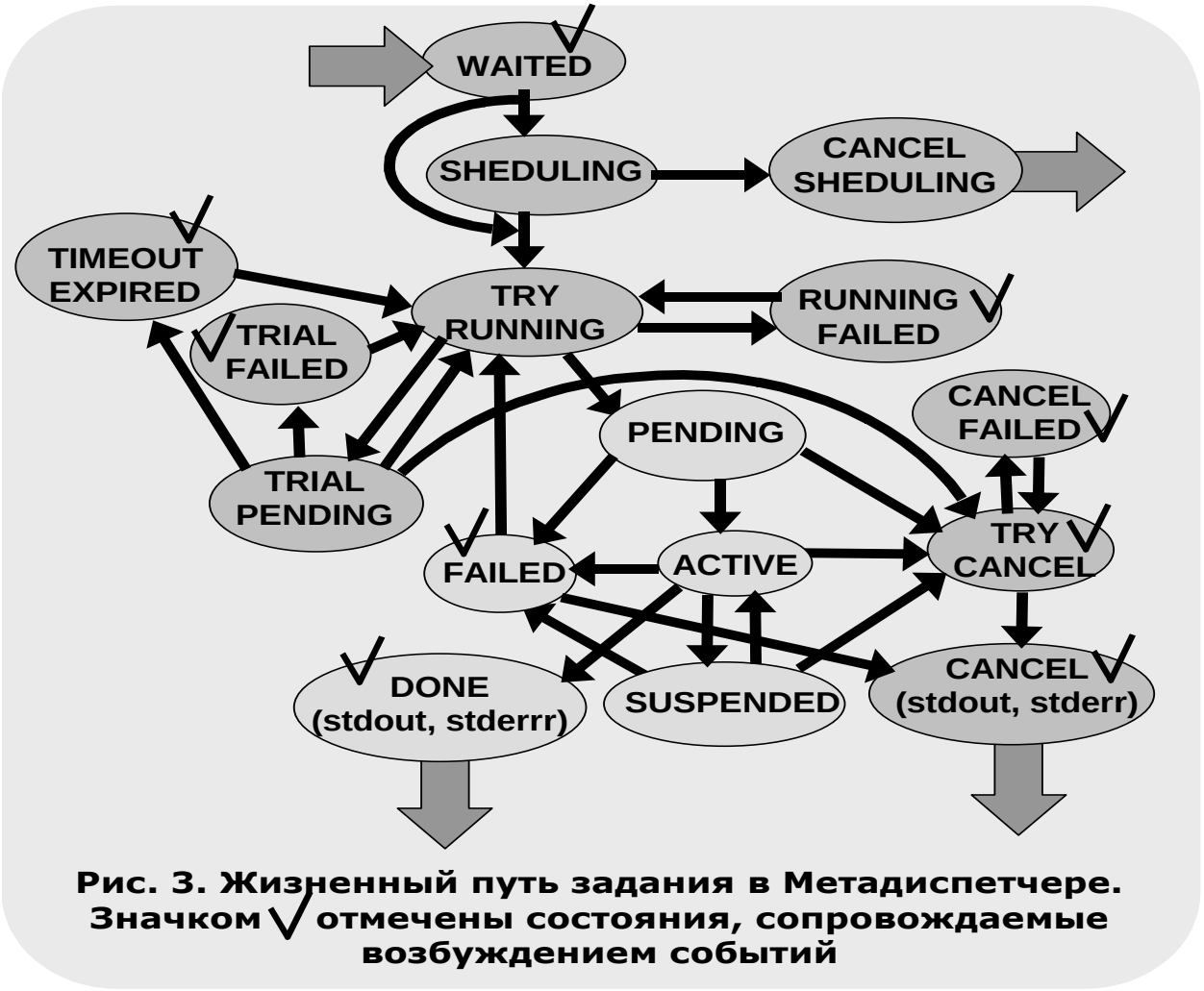

Рис. 3. Жизненный путь задания в Метадиспетчере. Значком √ отмечены состояния, сопровождаемые возбуждением событий

Когда пользователь издает команду принудительного снятия задания, оно (будучи запущенным в целевой GRAM) переводится в состояние **Try_Cancel** и делается попытка снять задание. При успешном снятии оно переходит в состояние **Cancel**, при неудаче – в состояние **Cancel_Failed**.

5. Метадиспетчер и подготовка заданий в гетерогенной среде

Если файл, указанный в параметре executable, представляет собой подготовленный для определенной архитектуры выполняемый файл (компилированную и собранную программу), то выполнение такого задания возможно только на вычислительных ресурсах Сети, обладающих именно этой архитектурой. Понятно, что такое использование Метадиспетчера значительно сужает его возможности.

Для того, чтобы получить возможность выполнять задания на вычислительных ресурсах с произвольной архитектурой, необходимо иметь средства подготовки программ в гетерогенной среде. Такое средство было нами разработано и получило название **metamake** [16]. Оно успешно

используется нами в локальной сети в рамках системы пакетной обработки PBS. Существующая версия metamake предполагает, что все исходные файлы, необходимые для подготовки, и файлы, получающиеся в результате подготовки, непосредственно доступны с любого хоста. В нашей локальной сети это условие выполняется, поскольку пользовательские директории на хостах, обслуживаемых PBS, доступны (и одинаково именуются) посредством NFS.

В рассматриваемой нами Интернет-среде это условие не может быть выполнено. Поэтому, чтобы применить metamake и получить возможность направлять задание на ресурсы с различной архитектуры, необходимо решить вопросы доступа к исходным файлам проекта и файлам, получающимся в результате подготовки. Последние желательно сохранить для повторного использования с тем, чтобы при повторном попадании задания на хост с такой же архитектурой можно было воспользоваться результатами подготовки.

Мы предлагаем организовать автоматическую доставку файлов metamake посредством утилиты **globus-rcp**, предоставляющей возможность безопасной передачи целых директорий (tar) в сжатом виде (gzip). Файл executable для задания может в этом случае представлять собой скрипт с одной единственной строчкой вызова metamake. Пользователю достаточно будет указать в опциях metamake местонахождение в сети (в формате <хост>:<директория>) директории исходных файлов и корня целевых директорий. Под этим корнем автоматически создается целевая директория с именем - различителем платформы, служащая базой для файлов (объектных, библиотек и выполняемых), созданных при подготовке проекта для данной архитектуры. Metamake сам определит, нужно ли предварительно подкачивать директорию исходных файлов и целевую директорию в локальную файловую среду, и, если нужно, то подкачает их. После того, как задание будет выполнено, metamake при необходимости переправит обратно содержимое целевой директории (если оно изменилось).

Для того, чтобы получить возможность использования metamake в среде Метадиспетчера, необходимо на каждом исполнительном хосте Сети, во-первых, установить metamake, и, во-вторых, обеспечить доступ к утилитам GASS, а на хостах, где располагаются удаленные файлы пользователей, обеспечить функционирование GRAM с сервисом jobmanager-fork и авторизовать владельцев этих файлов.

Литература

1. *I. Foster, C. Kesselman, S. Tuecke.* The Anatomy of the Grid: Enabling Scalable Virtual Organizations.//Публикуется в Intl. J. Supercomputer Applications, 2001. - ftp://ftp.globus.org/pub/globus/papers/anatomy.pdf
2. *В.Коваленко, Е.Коваленко, Д.Корягин, Е.Любимский, Е.Хухлаев.* Метадиспетчер: Управление заданиями в вычислительной Сети // Открытые системы, № 4, 2001.
3. *I.Foster, C Kesselman.* Globus: A Metacomputing Infrastructure Toolkit // Intl J. Supercomputer Applications, 11(2):115-128, 1997. - ftp://ftp.globus.org/pub/globus/papers/globus.pdf
4. *В.Коваленко, Е.Коваленко.* Пакетная обработка заданий в компьютерных сетях // Открытые системы, № 7-8, 2000. – с. 10-19. – http://www.osp.ru/2000/07-08/010.htm
5. *S.Fitzgerald, I.Foster, C.Kesselman, G. von Laszewski, W.Smith, S.Tuecke.* A Directory Service for Configuring High-Performance Distributed Computations. // Proc. 6th IEEE Symp. on High-Performance Distributed Computing, pg. 365-375, 1997. - ftp://ftp.globus.org/pub/globus/papers/hpdc97-mds.pdf
6. *М.К.Валиев, Е.Л.Китаев, М.И.Слепенков.* Служба директорий LDAP как инструментальное средство для создания распределенных информационных систем. Препринт ИПМ им. М.В.Келдыша РАН, 2000, № 23. - 22 с.
7. *М.К.Валиев, Е.Л.Китаев, М.И.Слепенков.* Использование службы директорий LDAP для представления метаинформации в глобальных вычислительных системах. Препринт ИПМ им. М.В.Келдыша РАН, 2000, № 29. - 27 с.
8. *I.Foster, C.Kesselman, G.Tsudik, S.Tuecke.* A Security Architecture for Computational Grids // Proc. 5th ACM Conference on Computer and Communications Security Conference, pg. 83-92, 1998. - ftp://ftp.globus.org/pub/globus/papers/security.pdf
9. *A.Freier, P.Karlton, P.Kocher.* The SSL Protocol. Version 3.0. Internet Draft. Netscape Communication Corporation. Nov 18 1996. - http://home.netscape.com/eng/ssl3/draft302.txt
10. *R.Rivest, A.Shamir, L.M.Adleman,* A Method for Obtaining Digital Signatures and Public-Key Cryptosystems // Communications of the ACM, v. 21, n. 2, Feb 1978, pp. 120-126.

11. *R.Housley, W.Ford, W.Polk, D.Solo.* Internet X.509 Public Key Infrastructure Certificate and CRL Profile. Internet RFC 2459. Jan 1999. - ftp://ftp.isi.edu/in-notes/rfc2459.txt

12. *J.Linn.* Generic security service application program interface. Version 2. Internet RFC 2078. Jan 1997. - http://www.ietf.org/rfc/rfc2078.txt

13. *J.Bester, I.Foster, C.Kesselman, J.Tedesco, S.Tuecke.* GASS: A Data Movement and Access Service for Wide Area Computing Systems. // Sixth Workshop on I/O in Parallel and Distributed Systems, May 5, 1999. - ftp://ftp.globus.org/pub/globus/papers/gass.pdf

14. *B.Allcock, L.Liming, S.Tuecke.* GridFTP: A Data Transfer Protocol for the Grid.- http://www.sdsc.edu/GridForum/RemoteData/Papers/gridftp_intro_gf5.pdf

15. *K.Czajkowski, I.Foster, N.Karonis, C.Kesselman, S.Martin, W.Smith, S.Tuecke.* A Resource Management Architecture for Metacomputing Systems. // Proc. IPPS/SPDP '98 Workshop on Job Scheduling Strategy for Parallel Processing, 1998 - ftp://ftp.globus.org/pub/globus/papers/gram97.pdf

16. *Е.В.Хухлаев.* Metamake – средство подготовки программ в сетевой гетерогенной среде. Препринт ИПМ им. М.В.Келдыша РАН, 1999, № 28. - 32 с.