Министерство образования и науки РФ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
АМУРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
(ФГБОУ ВО «АмГУ»)

РАСПРЕДЕЛЕННЫЕ СИСТЕМЫ ОБРАБОТКИ ИНФОРМАЦИИ

сборник учебно-методических материалов

для направления подготовки 09.04.04 Программная инженерия

Благовещенск, 2017

Печатается по решению
редакционно-издательского совета
факультета математики и информатики
Амурского государственного
Университета

Составитель: Жилиндина О.В.
Распределенные системы обработки информации: сборник учебно-методических материалов для направления подготовки 09.04.04. – Благовещенск: Амурский гос. ун-т, 2017.

1. Краткое изложение лекционного материала

1. Определение и задачи распределенной системы

1.1.Определение распределенной системы

Распределенная система — это набор независимых компьютеров, представляющийся их пользователям единой объединенной системой.

В этом определении рассматриваются два момента. Первый относится к аппаратуре: все машины автономны. Второй касается программного обеспечения: пользователи считают, что имеют дело с единой системой. Рассмотрим некоторые базовые вопросы, касающиеся как аппаратного, так и программного обеспечения.

Одна из характеристик распределенных систем состоит в том, что от пользователей скрыты различия между компьютерами и способы связи между ними. То же самое относится и к внешней организации распределенных систем. Другой важной характеристикой распределенных систем является способ, при помощи которого пользователи и приложения единообразно работают в распределенных системах, независимо от того, где и когда происходит их взаимодействие.

Распределенные системы должны относительно легко поддаваться расширению, или масштабированию. Эта характеристика является прямым следствием наличия независимых компьютеров, но в то же время не указывает, каким образом эти компьютеры на самом деле объединяются в единую систему. Распределенные системы обычно существуют постоянно, однако некоторые их части могут временно выходить из строя. Пользователи и приложения не должны уведомляться о том, что эти части заменены или исправлены или что добавлены новые части для поддержки дополнительных пользователей или приложений.

Для того чтобы поддержать представление различных компьютеров и сетей в виде единой системы, организация распределенных систем часто включает в себя дополнительный уровень программного обеспечения, находящийся между верхним уровнем, на котором находятся пользователи и приложения, и нижним уровнем, состоящим из операционных систем, как показано на рис. 1. Соответственно, такая распределенная система обычно называется *системой промежуточного уровня (middleware).*

Рис. 1. Распределенная система организована в виде службы промежуточного уровня.

Отметим, что промежуточный уровень распределен среди множества компьютеров

Рассмотрим некоторые примеры распределенных систем. В качестве первого примера рассмотрим сеть рабочих станций в университете или отделе компании. Вдобавок к персональной рабочей станции каждого из пользователей имеется пул процессоров машинного зала, не назначенных заранее ни одному из пользователей, но динамически выделяемых им при необходимости. Эта распределенная система может обладать единой файловой системой, в которой все файлы одинаково доступны со всех машин с использованием постоянного пути доступа. Кроме того, когда пользователь набирает команду, система может найти наилучшее место для выполнения запрашиваемого действия, возможно, на собственной рабочей станции пользователя, возможно, на простаивающей рабочей станции, принадлежащей кому-то другому, а может быть, и на одном из свободных процессоров машинного зала. Если система в целом выглядит и ведет себя как классическая однопроцессорная система с разделением времени (то есть многопользовательская), она считается распределенной системой. В качестве второго примера рассмотрим работу информационной системы, которая поддерживает автоматическую обработку заказов. Обычно подобные сис-

темы используются сотрудниками нескольких отделов, возможно в разных местах. Так, сотрудники отдела продаж могут быть разбросаны по обширному региону или даже по всей стране. Заказы передаются с переносных компьютером, соединяемых с системой при помощи телефонной сети, а возможно, и при помощи сотовых телефонов. Приходящие заказы автоматически передаются в отдел планирования, превращаясь там во внутренние заказы на поставку, которые поступают в отдел доставки, и в заявки на оплату, поступающие в бухгалтерию. Система автоматически пересылает эти документы имеющимся на месте сотрудникам, отвечающим за их обработку. Пользователи остаются в полном неведении о том, как заказы на самом деле курсируют внутри системы, для них все это представляется так, будто вся работа происходит в централизованной базе данных.

В качестве третьего примера рассмотрим World Wide Web. Web предоставляет простую, целостную и единообразную модель распределенных документов. Чтобы увидеть документ, пользователю достаточно активизировать ссылку. После этого документ появляется на экране. Нет необходимости знать, с какого сервера доставляется документ, достаточно лишь информации о том, где он расположен. Публикация документа очень проста: требуется задать ему уникальное имя в форме *унифицированного указателя ресурса (Uniform Resource Locator, URL),* которое ссылается на локальный файл с содержимым документа. Если бы Всемирная паутина представлялась своим пользователям гигантской централизованной системой документооборота, она также могла бы считаться распределенной системой. К сожалению, пользователи сознают, что документы находятся в различных местах и распределены по различным серверам.

1.2. Задачи распределенных систем

Обсудим четыре важных задачи. Распределенные системы могут легко соединять пользователей с вычислительными ресурсами и успешно скрывать тот факт, что ресурсы разбросаны по сети и могут быть открытыми и масштабируемыми.

Соединение пользователей с ресурсами

Основная задача распределенных систем — облегчить пользователям доступ к удаленным ресурсам и обеспечить их совместное использование, регулируя этот процесс. Ресурсы могут быть виртуальными, однако традиционно они включают в себя принтеры, компьютеры, устройства хранения данных, файлы и данные. Web-страницы и сети также входят в этот список. Существует множество причин для совместного использования ресурсов. Одна из очевидных — это экономичность. Гораздо дешевле разрешить совместную работу с принтером нескольких пользователей, чем покупать и обслуживать отдельный принтер Для каждого пользователя. Точно так же имеет смысл совместно использовать дорогие ресурсы, такие как суперкомпьютеры или высокопроизводительные хранилища данных.

Соединение пользователей с ресурсом также облегчает кооперацию и обмен информацией, что иллюстрируется Интернетом с его простыми протоколами для обмена файлами, почтой, документами, аудио- и видеоинформацией. Связь через Интернет привела к появлению многочисленных виртуальных организаций, в которых географически удаленные друг от друга группы сотрудником работают вместе при помощи систем групповой работы (groupware) — программ для совместного редактирования документов, проведения телеконференций и т. п. Подобным образом подключение к Интернету привело к появлению электронной коммерции, позволяющей покупать и продавать любые виды товаров, без реального посещения магазина.

Однако по мере роста числа подключений и степени совместного использования ресурсов все более и более важными становятся вопросы безопасности. В настоящее время системы имеют слабую защиту от подслушивания или вторжения по линиям связи. Пароли и другая особо важная информация часто пересылаются по сетям открытым текстом (незашифрованными) или хранятся на ненадежных серверах. Имеется много возможностей для улучшения безопасности. Так для заказа товаров необходимо просто сообщить номер своей кредитной карты. Редко требуется подтверждение того, что покупатель действительно владеет этой картой. В будущем заказ товара таким образом будет возможен только в том случае, если можно физически подтвердить факт обладания картой при помощи считывателя карт.

Другая проблема безопасности состоит в том, что прослеживание коммуникаций позволяет построить профиль предпочтений конкретного пользователя. Подобное отслеживание нарушает пра-

ва личности, особенно если производится без уведомления пользователя. Связанная с этим проблема состоит в том, что рост подключений ведет к росту нежелательного общения, такого как получаемые по электронной почте бессмысленные письма. Можно защититься, используя специальные информационные фильтры, которые сортируют входящие сообщения на основании их содержимого.

Прозрачность

Важная задача распределенных систем состоит в сокрытии факта, что процессы и ресурсы физически распределены по множеству компьютеров. Распределенные системы, которые представляются пользователям и приложениям в виде единой компьютерной системы, называются прозрачными (transparent). Рассмотрим реализацию прозрачности в распределенных системах.

Концепция прозрачности применима к различным аспектам распределенных систем.

Прозрачность	Описание
Доступ	Скрывается разница в представлении данных и доступе к ресурсам
Местоположение	Скрывается местоположение ресурса
Перенос	Скрывается факт перемещения ресурса в другое место
Смена местоположения	Скрывается факт перемещения ресурса в процессе обработки в другое место
Репликация	Скрывается факт репликации ресурса
Параллельный доступ	Скрывается факт возможного совместного использования ресурса несколькими конкурирующими пользователями
Отказ	Скрывается отказ и восстановление ресурса
Сохранность	Скрывается, хранится ресурс (программный) на диске или находится в оперативной памяти

Таблица 1. Различные формы прозрачности в распределенных системах

Прозрачность доступа (access transparency) должна скрыть разницу в представлении данных и в способах доступа пользователя к ресурсам. Так, при пересылке целого числа с рабочей станции на базе процессора Intel на Sun SPARC необходимо знать, что процессоры Intel оперируют с числами формата «младший — последним» (то есть первым передается старший байт), а процессор SPARC использует формат «старший — последним» (то есть первым передается младший байт). В данных могут присутствовать и другие несоответствия. Например, распределенная система может содержать компьютеры с различными операционными системами, каждая из которых имеет собственные ограничения на способ представления имен файлов. Разница в ограничениях на способ представления имен файлов, так же как и работа с ними, должны быть скрыты от пользователей и приложений.

Важная группа типов прозрачности связана с местоположением ресурсов. Прозрачность местоположения (location transparency) должна скрыть от пользователя, где физически расположен в системе нужный ему ресурс. Важную роль в реализации прозрачности местоположения играет именование. Прозрачность местоположения может быть достигнута путем присвоения ресурсам логических имен, то есть имен, в которых не содержится закодированных сведений о местоположении ресурса. Примером такого имени может быть URL: http://-www.prenhall.com/index.html, в котором не содержится никакой информации о реальном местоположении главного web- сервера издательства Prentice Hall. URL также не дает никакой информации о том, находился ли файл index.html в указанном месте постоянно или оказался там недавно. О распределенных системах, в которых смена местоположения ресурсов не влияет на доступ к ним, говорят как об обеспечивающих прозрачность переноса (migration transparency). Более сложна ситуация, когда местоположение ресурсов может измениться в процессе их использования, причем пользователь или приложение этого не заметят. В таком случае говорят, что система поддерживает прозрачность смены местоположения (relocation transparency). Примером могут служить мобильные пользователи, рабо-

тающие с беспроводным переносным компьютером и не отключающиеся от сети при перемещении с места на место.

В распределенных системах важное значение имеет репликация. Ресурсы могут быть реплицированы для их лучшей доступности или повышения их производительности путем помещения копии близко к месту, из которого к ней осуществляется доступ. Прозрачность репликации (replication transparency) позволяет скрыть тот факт, что существует несколько копий ресурса. Для скрытия факта репликации от пользователей необходимо, чтобы все реплики имели одно и то же имя. Соответственно, система, которая поддерживает прозрачность репликации, должна поддерживать и прозрачность местоположения, поскольку иначе невозможно обращаться к репликам без указания их истинного местоположения.

Главная цель распределенных систем — обеспечить совместное использование ресурсов. Во многих случаях совместное использование ресурсов достигается посредством кооперации, например в случае коммуникаций. Однако существует множество примеров настоящего совместного использования ресурсов. Например, дна независимых пользователя могут сохранять свои файлы на одном файловом сервере или работать с одной и той же таблицей в совместно используемой базе данных. В таких случаях ни один из пользователей не знает о том, что тот же ресурс задействован другим пользователем. Это явление называется прозрачностью параллельного доступа (concurrency transparency). Отметим, что подобный параллельный доступ к совместно используемому ресурсу сохраняет этот ресурс в непротиворечивом состоянии. Непротиворечивость может быть обеспечена механизмом блокировок, когда пользователи, каждый по очереди, получают исключительные права на запрашиваемый ресурс. Более изощренный вариант - использование транзакций, однако механизм транзакций в распределенных системах труднореализуем.

Имеется еще одна важная сторона распределенных систем - прозрачность отказов. *Прозрачность отказов (failure transparency)* означает, что пользователя не уведомляют о том, что ресурс не в состоянии правильно работать и что система восстановилась после этого отказа. Маскировка сбоев

— это одна из сложнейших проблем в распределенных системах и необходимая их часть. Основная трудность состоит в маскировке проблем, возникающих в связи с невозможностью отличить неработоспособные ресурсы от ресурсов с очень медленным доступом. Так, контактируя с перегруженным web-сервером, браузер выжидает положенное время, а затем сообщает о недоступности страницы. При этом пользователь не должен думать, что сервер не работает.

Последний тип прозрачности, который обычно ассоциируется с распределенными системами, — это *прозрачность сохранности (persistence transparency),* маскирующая реальную (диск) или виртуальную (оперативная память) сохранность ресурсов. Так, например, многие объектно- ориентированные базы данных предоставляют возможность непосредственного вызова методов для сохраненных объектов. В этот момент происходит следующее: сервер баз данных сначала копирует состояние объекта с диска в оперативную память, затем выполняет операцию и записывает состояние на устройство длительного хранения. Пользователь, однако, остается в неведении о том, что сервер перемещает данные между оперативной памятью и диском. Сохранность играет важную роль в распределенных системах.

Степень прозрачности

Существуют ситуации, когда попытки полностью скрыть от пользователя распределенность не разумны. Это относится, например, к требованию присылать свежую электронную газету до 7 утра по местному времени, если адресат живет в другом часовом поясе.

Так же в глобальной распределенной системе, которая соединяет процесс в Сан-Франциско с процессом в Амстердаме, не удастся скрыть факт, нельзя пересылать сообщения от одного процесса к другому быстрее, чем за несколько сотен мс. Скорость передачи сигнала ограничивается не столько скоростью света, сколько скоростью работы промежуточных переключателей.

Существует равновесие между высокой степенью прозрачности и производительностью системы. Многие приложения, предназначенные для Интернета, многократно пытаются установить связь с сервером, пока не откажутся от этого. Попытки замаскировать сбой на промежуточном сервере, вместо попытки работать через другой сервер, замедляют работу системы. Было бы эф-

фективнее прекратить эти попытки или позволить пользователю прервать попытки установления контакта.

Необходимо, чтобы реплики, находящиеся на разных континентах, были в любой момент гарантированно идентичны. Другими словами, если одна копия изменилась, изменения должны распространиться на все системы дотого, как они выполнят какую-либо операцию. Одиночная операция обновления может в этом случае занимать несколько секунд и ее невозможно проделать незаметно для пользователей.

Таким образом, достижение прозрачности распределения — это важная цель при проектировании и разработке распределенных систем, но она не должна рассматриваться в отрыве от других характеристик системы, например производительности.

Открытость

Другая важная характеристика распределенных систем - это открытость. *Открытая распределенная система (open distributed system)* — это система, предлагающая службы, вызов которых требует стандартный синтаксис и семантику. Например, в компьютерных сетях формат, содержимое и смысл посылаемых и принимаемых сообщений подчиняются типовым правилам. Эти правила формализованы в протоколах. В распределенных системах службы обычно определяются через *интерфейсы (interfaces),* которые часто описываются при помощи *языка определения интерфейсов (Interface Definition Language, IDL).* Описание интерфейса на IDL почти исключительно касается синтаксиса служб. Оно точно отражает имена доступных функций, типы параметров, возвращаемых значений, исключительные ситуации, которые могут быть возбуждены службой и т. п. Сложно точно описать, что делает эта служба, то есть семантику интерфейсов. На практике подобные спецификации задаются неформально, посредством естественного языка.

Определение интерфейса допускает возможность совместной работы произвольного процесса, нуждающегося в таком интерфейсе, с другим произвольным процессом, предоставляющим этот интерфейс. Определение интерфейса также позволяет двум независимым группам создать абсолютно разные реализации этого интерфейса для двух различных распределенных систем, которые будут работать абсолютно одинаково. Правильное определение самодостаточно и нейтрально. «Самодостаточно» означает, что в нем имеется все необходимое для реализации интерфейса. Важно отметить, что спецификация не определяет внешний вид реализации, она должна быть нейтральной. Самодостаточность и нейтральность необходимы для обеспечения переносимости и способности к взаимодействию. *Способность к взаимодействию (interoperability)* характеризует, насколько две реализации систем или компонентов от разных производителей в состоянии совместно работать, полагаясь только на то, что службы каждой из них соответствуют общему стандарту. *Переносимость (portability)* характеризует то, насколько приложение, разработанное для распределенной системы А, может без изменений выполняться в распределенной системе В реализуя те же, что и в А интерфейсы.

Важная характеристика открытых распределенных систем — это гибкость. Под гибкостью понимается легкость конфигурирования системы, состоящей из различных компонентов, возможно от разных производителей.

Не должны вызывать затруднений добавление к системе новых компонентов или замена существующих, при этом прочие компоненты, с которыми не производилось никаких действий, должны оставаться неизменными. Другими словами, открытая распределенная система должна быть расширяемой. К гибкой системе несложно добавить части, работающие под управлением другой операционной системы, или заменить всю файловую систему целиком.

Отделение правил от механизмов

В построении гибких открытых распределенных систем решающим фактором оказывается организация этих систем в виде наборов относительно небольших и легко заменяемых или адаптируемых компонентов. Это предполагает необходимость определения не только интерфейсов верхнего уровня, с которыми работают пользователи и приложения, но также и интерфейсов внутренних модулей системы и описания взаимодействия этих модулей. Это новый подход. Множество старых и современных систем создавались цельными так, что компоненты одной гигантской

программы разделялись только логически. Независимая замена или адаптация компонентов, не затрагивающая систему в целом, была почти невозможна.

Необходимость изменений в распределенных системах часто связана с тем, что компонент не оптимальным образом соответствует нуждам конкретного пользователя или приложения. Рассмотрим кэширование в World Wide Web. Браузеры обычно позволяют пользователям адаптировать правила кэширования под их нужды путем определения размера кэша, а также того, должен ли кэшируемый документ проверяться на соответствие постоянно или только один раз за сеанс. Однако пользователь не может воздействовать на другие параметры кэширования, такие как длительность сохранения документа в кэше или очередность удаления документов из кэша при его переполнении. Также невозможно создавать правила кэширования на основе *содержимого* документа. Так, например, пользователь может пожелать кэшировать железнодорожные расписания, которые редко изменяются, но не информацию о пробках на улицах города.

Необходимо отделить правила от механизма. В случае кэширования в Web браузер должен предоставлять только возможности для сохранения документов в кэше и одновременно давать пользователям возможность решать, какие документы и насколько долго там хранить. На практике это может быть реализовано предоставлением большого списка параметров, значения которых пользователь сможет (динамически) задавать. Еще лучше, если пользователь получит возможность сам устанавливать правила в виде подключаемых к браузеру компонентов. Разумеется, браузер должен понимать интерфейс этих компонентов, поскольку ему нужно будет, используя этот интерфейс, вызывать процедуры, содержащиеся в компонентах.

Масштабируемость

Масштабируемость системы может измеряться по трем различным показателям. Во-первых, система может быть масштабируемой по отношению к ее размеру, что означает легкость подключения к ней дополнительных пользователей и ресурсов. Во-вторых, система может масштабироваться географически, то есть пользователи и ресурсы могут быть разнесены в пространстве. В-третьих, система может быть масштабируемой в административном смысле, то есть быть проста в управлении при работе во множестве административно независимых организаций. Однако, система, обладающая масштабируемостью по одному или нескольким из этих параметров, при масштабировании часто дает потерю производительности.

Проблемы масштабируемости

Если система нуждается в масштабировании, необходимо решить множество разнообразных проблем. Рассмотрим масштабирование по размеру. Если возникает необходимость увеличить число пользователей или ресурсов, сталкиваются с ограничениями, связанными с централизацией служб, данных и алгоритмов (табл. 2). Многие службы централизуются потому, что при их реализации предполагалось наличие в распределенной системе только одного сервера, запущенного на конкретной машине. При увеличении числа пользователей сервер легко может стать критическим местом системы. Даже если имеется фактически неограниченный запас по мощности обработки и храпения данных, ресурсы связи с этим сервером в конце концов будут исчерпаны.

Концепция	Пример
Централизованные службы	Один сервер на всех пользователей
Централизованные данные	Единый телефонный справочник, доступный в режиме подключения
Централизованные алгоритмы	Организация маршрутизации на основе полной информации

Таблица 2. Примеры ограничений масштабируемости

К сожалению, использование единственного сервера время от времени неизбежно, например, для служб управления особо конфиденциальной информацией, такой как истории болезни, банковские счета, кредиты и т. п. В подобных случаях необходимо реализовывать службы на одном сервере в отдельной хорошо защищенной комнате и отделять их от других частей распределенной

системы посредством специальных сетевых устройств. Копирование информации, содержащейся на сервере, в другие места для повышения производительности не допускается.

Централизация данных так же вредна, как и централизация служб. Невозможно отслеживать телефонные номера и адреса 50 миллионов человек. Предположим, что каждая запись укладывается в 50 символов. Необходимой емкостью обладает один 2,5-гигабайтный диск. Но и в этом случае наличие единой базы данных вызовет перегрузку входящих и исходящих линий связи. Предположим что в Интернет служба доменных имен (DNS) реализована в виде одной таблицы. DNS обрабатывает информацию с миллионов компьютеров во всем мире и предоставляет службу, необходимую для определения местоположения web-серверов. Если бы каждый запрос на интерпретацию URL передавался на единственный DNS-сервер, воспользоваться Web не смог бы никто.

Централизация алгоритмов так же не выдерживает критики. В больших распределенных системах гигантское число сообщений необходимо направлять по множеству каналов. Теоретически для вычисления оптимального пути необходимо получить полную информацию о загруженности всех машин и линий и по алгоритмам из теории графов вычислить все оптимальные маршруты. Эта информация затем должна быть роздана по системе для улучшения маршрутизации.

Проблема состоит в том, что сбор и транспортировка всей информации может перегрузить часть сети. Следует избегать алгоритма, который требует передачи информации, собираемой со всей сети, на одну из ее машин для обработки с последующей раздачей результатов. Использовать следует только децентрализованные алгоритмы. Эти алгоритмы обычно обладают следующими свойствами, отличающими их от централизованных алгоритмов;

- ни одна из машин не обладает полной информацией о состоянии системы;
- машины принимают решения на основе локальной информации;
- сбой на одной машине не вызывает нарушения алгоритма;
- не требуется предположения о существовании единого времени.

Первые три свойства поясняют ранее сказанное. Последнее, менее очевидно, но не менее важно. Алгоритм, реализующий утверждение: «Ровно в 12:00:00 все машины должны определить размер своих входных очередей», работать не будет, поскольку невозможно синхронизировать все часы в мире. Алгоритмы должны принимать во внимание отсутствие полной синхронизации таймеров. Чем больше система, тем большим будет и рассогласование. В одной локальной сети путем определенных усилий можно добиться, чтобы рассинхронизация всех часов не превышала нескольких миллисекунд, но невозможно сделать это в масштабе страны или множества стран.

У географической масштабируемости имеются свои проблемы. Одна из основных причин сложности масштабирования существующих распределенных систем, разработанных для локальных сетей, состоит в том, что в их основе лежит принцип *синхронной связи (synchronous communication).* В этом виде связи запрашивающий службу агент, которого принято называть *клиентом (client),* блокируется до получения ответа. Этот подход обычно успешно работает в локальных сетях, когда связь между двумя машинами продолжается максимум сотни микросекунд. Однако в глобальных системах необходимо принять во внимание факт, что связь между процессами может продолжаться сотни миллисекунд, то есть на три порядка дольше.

Другая проблема, препятствующая географическому масштабированию, состоит в том, что связь в глобальных сетях фактически всегда организуется от точки к точке и потому ненадежна. В противоположность глобальным, локальные сети обычно дают высоконадежную связь, основанную на широковещательной рассылке, что делает разработку распределенных систем для них значительно проще. Для примера рассмотрим проблему локализации службы. В локальной сети система просто рассылает сообщение всем машинам, опрашивая их на предмет предоставления нужной службы. Машины, предоставляющие службу, отвечают на это сообщение, указывая в ответном сообщении свои сетевые адреса. Невозможно реализовать подобную схему определения местоположения в глобальной сети. Вместо этого необходимо обеспечить специальные места для расположения служб, которые может потребоваться масштабировать на весь мир и обеспечить их мощностью для обслуживания миллионов пользователей.

Географическая масштабируемость жестко связана с проблемами централизованных решений, которые мешают масштабированию по размеру. Если имеется система с множеством централизованных компонентов, то географическая масштабируемость будет ограничиваться проблемами производительности и надежности, связанными с глобальной связью. Кроме того, централизованные компоненты способны вызвать перегрузку сети.

Во многих случаях остается открытым вопрос, как обеспечить масштабирование распределенной системы на множество административно независимых областей. Основная проблема, которую нужно при этом решить, состоит в конфликтах правил, относящихся к использованию ресурсов (и плате за них), управлению и безопасности.

Так, множество компонентов распределенных систем, находящихся в одной области, обычно может быть доверено пользователям, работающим в этой области. В этом случае системный администратор может тестировать и сертифицировать приложения, используя специальные инструменты. Пользователи доверяют своему системному администратору. Однако это доверие не распространяется за границы области.

Если распределенные системы распространяются на другую область, могут потребоваться два типа проверок безопасности. Во-первых, распределенная система должна противостоять атакам из новой области. Так, например, пользователи новой области могут получить ограниченные права доступа к файловой службе системы в исходной области, скажем, только на чтение. Точно так же может быть закрыт доступ чужих пользователей и к аппаратуре, такой как дорогостоящие устройства печати или высокопроизводительные компьютеры. Во-вторых, новая область сама должна быть защищена от злонамеренных атак из распределенной системы. Типичным примером является загрузка по сети программ, таких как апплеты в web- браузерах. Изначально новая область не знает, чего ожидать от чужого кода, и потому строго ограничивает ему права доступа.

Технологии масштабирования

Поскольку проблемы масштабируемости в распределенных системах, связанные с производительностью, вызываются ограниченной мощностью серверов и сетей, существуют три основные технологии масштабирования; сокрытие времени ожидания связи, распределение и репликация.

Сокрытие времени ожидания связи применяется и случае географического масштабирования. Основная идея: по возможности избежать ожидания ответа на запрос от удаленного сервера. Например, если была запрошена служба удаленной машины, альтернативой ожиданию ответа от сервера будет осуществление на запрашивающей стороне других возможных действий. Это означает разработку запрашивающего приложения в расчете на использование исключительно *асинхронной связи (в-sinchronous communication).* Когда будет получен ответ, приложение прервет свою работу и вызовет специальный обработчик для завершения отправленного ранее запроса. Асинхронная связь часто используется в системах пакетной обработки и параллельных приложениях, в которых во время ожидания одной задачей завершения связи предполагается выполнение других более или менее независимых задач. Для осуществления запроса может быть запущен новый управляющий поток выполнения. Хотя он будет блокирован на время ожидания ответа, другие потоки процесса продолжат свое выполнение.

Однако многие приложения не в состоянии эффективно использовать асинхронную связь. Например, когда в интерактивном приложении пользователь посылает запрос, он обычно не в состоянии делать ничего кроме, ожидания ответа. В этих случаях наилучшим решением будет сократить необходимый объем взаимодействия, например, переместив часть вычислений, обычно выполняемых на сервере, на клиента, процесс которого запрашивает службу. Стандартный случай применения этого подхода — доступ к базам данных с использованием форм. Обычно заполнение формы сопровождается посылкой отдельного сообщения на каждое поле и ожиданием подтверждения приема от сервера, как показано па рис. 2, *а.* Сервер может перед приемом введенного значения проверить его на синтаксические ошибки. Более успешное решение состоит в том, чтобы перенести код для заполнения формы и, возможно, проверки введенных данных на клиента, чтобы он мог послать серверу целиком заполненную форму (рис. 2, б*).* Такой подход — перенос кода на клиента - в настоящее время широко поддерживается в Web посредством Java-апплетов.

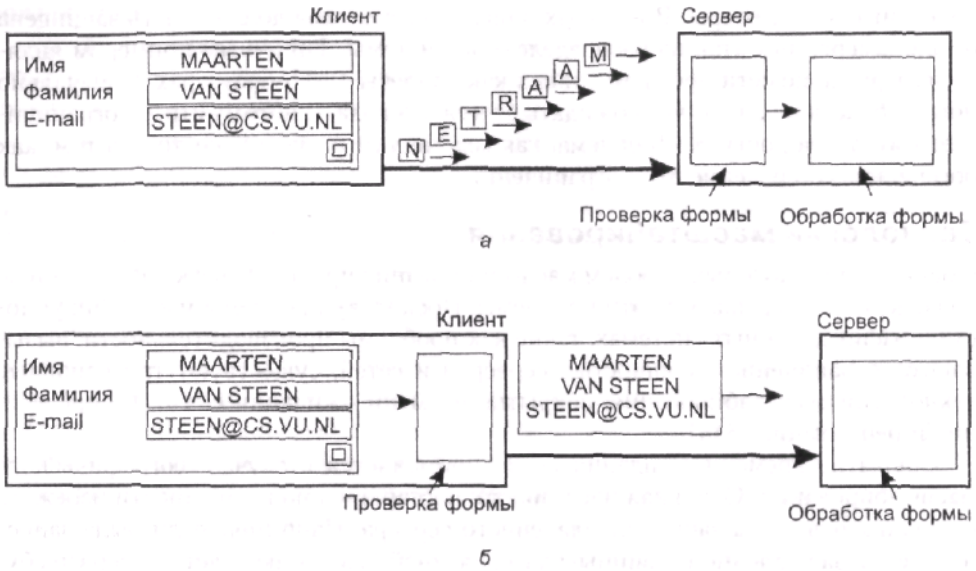


Рис. 2. Разница между проверкой формы по мере заполнения на сервере (а) и на клиенте *(б)*

Следующая важная технология масштабирования — *распределение (distribution).* Распределение предполагает разбиение компонентов на мелкие части и последующее разнесение этих частей по системе. Примером распределения является система доменных имен Интернета (DNS). Пространство DNS-имен организовано иерархически, в виде дерева *доменов (domains),* которые разбиты на неперекрывающиеся *зоны (zones),* как показано на рис. 3. Имена каждой зоны обрабатываются одним сервером имен. Можно считать, что каждое доменное имя является именем хоста в Интернете и ассоциируется с сетевым адресом этого хоста. В основном интерпретация имени означает получение сетевого адреса соответствующего хоста. Рассмотрим имя nl.vu.cs.flits. Для интерпретации этого имени оно сначала передается на сервер зоны *Z1* (рис. 1.3), который возвращает адрес сервера зоны Z2, который, вероятно, сможет обработать остаток имени, vu.cs.flits. Сервер зоны *Z2* вернет адрес сервера зоны *Z3,* который способен обработать последнюю часть имени и вернуть адрес соответствующего хоста.

Эти примеры демонстрируют, как *служба именования,* предоставляемая DNS, распределена по нескольким машинам и как это позволяет избежать обработки всех запросов на интерпретацию имен одним сервером.

В качестве другого примера рассмотрим World Wide Web. Для большинства пользователей Web представляется гигантской информационной системой документооборота, в которой каждый документ имеет свое уникальное имя — URL. Концептуально можно предположить даже, что все документы размещаются на одном сервере. Однако среда Web физически разнесена по множеству серверов, каждый из которых содержит некоторое количество документов. Имя сервера, содержащего конкретный документ, определяется по URL-адресу документа. Только благодаря подобному распределению документов Всемирная паутина смогла вырасти до ее современных размеров.

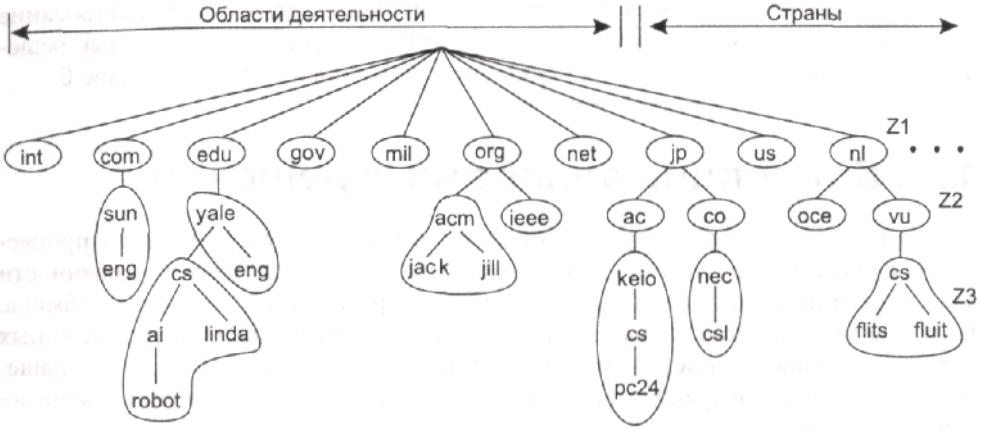


Рис. 3. Пример разделения пространства DNS-имен на зоны

При рассмотрении проблем масштабирования, часто проявляющихся в виде падения производительности, нередко хорошей идеей является *репликация (replication)* компонентов распределенной системы. Репликация – это создание копии ресурса его владельцем для увеличения производительности его совместной обработки данного ресурса. Репликация не только повышает доступность, но и помогает выровнять загрузку компонентов, что ведет к повышению производительности. Кроме того, в сильно географически рассредоточенных системах наличие близко лежащей копии позволяет снизить остроту большей части ранее обсуждавшихся проблем ожидания завершения связи.

Кэширование (caching) представляет собой особую форму репликации, причем различия между ними нередко малозаметны или вообще искусственны. Как и в случае репликации, результатом кэширования является создание копии ресурса, обычно в непосредственной близости от клиента, использующего этот ресурс. Однако в противоположность репликации кэширование — это действие, предпринимаемое потребителем ресурса, а не его владельцем.

На масштабируемость может плохо повлиять один существенный недостаток кэширования и репликации. Поскольку мы получаем множество копий ресурса, модификация одной копии делает ее отличной от остальных. Соответственно, кэширование и репликация вызывают проблемы *непротиворечивости (consistency).*

Допустимая степень противоречивости зависит от степени загрузки ресурсов. Гак, множество пользователей Web считают допустимым работу с кэшированным документом через несколько минут после его помещения в кэш без дополнительной проверки. Однако существует множество случаев, когда необходимо гарантировать строгую непротиворечивость, например, при игре на электронной бирже. Проблема строгой непротиворечивости состоит в том, что изменение в одной из копий должно немедленно распространяться на все остальные. Кроме того, если два изменения происходят одновременно, часто бывает необходимо, чтобы эти изменения вносились в одном и том же порядке во все копии. Для обработки ситуаций такого типа обычно требуется механизм глобальной синхронизации. К сожалению, реализовать масштабирование подобных механизмов крайне трудно, и может быть невозможно. Это означает, что масштабирование путем репликации может включать в себя отдельные *немасштабируемые* решения.

Концепции аппаратных решений

Несмотря на то, что все распределенные системы содержат по нескольку процессоров, существуют различные способы их организации в систему. В особенности это относится к вариантам их соединения и организации взаимного обмена. Рассмотрим аппаратное обеспечение распределенных систем, в частности варианты соединения машин между собой

Будем рассматривать системы, построенные из набора независимых компьютеров. На рис. 4 все компьютеры разделены на две группы. Системы, в которых компьютеры используют память совместно, обычно называются *мультипроцессорами (multiprocessors),* а работающие каждый со своей памятью

— *мультикомпьютерами (mullicomputers).* Основная разница между ними состоит в том, что мультипроцессоры имеют единое адресное пространство, совместно используемое всеми процессорами. Если один из процессоров записывает, например, значение 44 по адресу 1000, любой другой процессор, который после этого прочтет значение, лежащее по адресу 1000, получит 44. Все машины задействуют одну и ту же память.

В отличие от таких машин в мультикомпьютерах каждая машина использует свою собственную память. После того как один процессор запишет значение 44 по адресу 1000, другой процессор, прочитав значение, лежащее по адресу 1000, получит то значение, которое хранилось там раньше. Запись по этому адресу значения 44 другим процессором никак не скажется на содержимом его памяти. Типичный пример мультикомпьютера — несколько персональных компьютеров, объединенных в сеть.

Каждая из этих категорий может быть подразделена на дополнительные категории на основе архитектуры соединяющей их сети. На рис. 4 эти две архитектуры обозначены как *шинная (bus)* и *коммутируемая (switched).* Под шиной понимается одиночная сеть, плата, шина, кабель или другая среда, соединяющая ее машины между собой. Подобную схему использует кабельное телевиде-

ние: кабельная компания протягивает вдоль улицы кабель, а всем подписчикам делаются отводки от основного кабеля к их телевизорам.

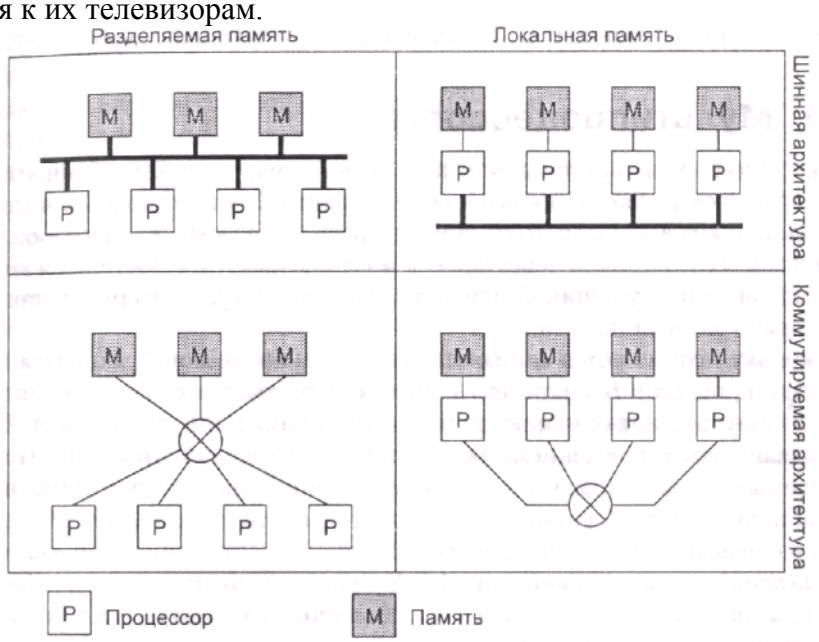


Рис. 4. Различные базовые архитектуры процессоров и памяти распределенных компьютерных систем

Коммутируемые системы, в отличие от шинных, не имеют единой магистрали, такой как у кабельного телевидения. Вместо нее от машины к машине тянутся отдельные каналы, выполненные с применением различных технологий связи. Сообщения передаются по каналам с принятием явного решения о коммутации с конкретным выходным каналом для каждого из них. Так организована глобальная телефонная сеть.

Мы проведем также разделение распределенных компьютерных систем на *гомогенные (homogeneous)* и *гетерогенные {heterogeneous).* Это разделение применяется исключительно к мультикомпьютерным системам. Для гомогенных мультикомпьютерных систем характерна одна соединяющая компьютеры сеть, использующая единую технологию. Одинаковы также и все процессоры, которые в основном имеют доступ к одинаковым объемам собственной памяти. Гомогенные мультикомпьютерные системы нередко используются в качестве параллельных (работающих с одной задачей), в точности как мультипроцессорные.

В отличие от них гетерогенные мультикомпьютерные системы могут содержать целую гамму независимых компьютеров, соединенных разнообразными сетями. Так, например, распределенная компьютерная система может быть построена из нескольких локальных компьютерных сетей, соединенных коммутируемой магистралью FDDI или ATM.

В следующих трех пунктах рассмотрим мультипроцессорные, а также гомогенные и гетерогенные мультикомпьютерные системы. Эти вопросы не связаны напрямую с распределенными системами, однако они помогают лучше их понять, поскольку организация распределенных систем часто зависит от входящей с их состав аппаратуры.

Мультипроцессоры

Мультипроцессорные системы обладают характерной особенностью: все процессоры имеют прямой доступ к общей памяти. Мультипроцессорные системы шинной архитектуры состоят из некоторого количества процессоров, подсоединенных к общей шине, а через нее — к модулям памяти. Простейшая конфигурация содержит плату с шиной или материнскую плату, в которую вставляются процессоры и модули памяти.

Поскольку используется единая память, когда процессор *A* записывает слово в память, а процессор *B* микросекундой позже считывает слово из памяти, процессор *B* получает информацию, записанную в память процессором *A*. Память, обладающая таким поведением, называется *согласованной (coherent).* Проблема данной схемы состоит в том, что в случае 4 или 5 процессоров

шина оказывается стабильно перегруженной и производительность резко падает. Решение состоит в размещении между процессором и шиной высокоскоростной *кэш-памяти (cache memory),* как показано на рис. 5. В кэше сохраняются данные, обращение к которым происходит наиболее часто. Все запросы к памяти происходят через кэш. Если запрошенные данные находятся в кэш- памяти, то на запрос процессора реагирует она и обращения к шине не выполняются. Если размер кэш-памяти достаточно велик, вероятность успеха, называемая также *коэффициентом кэш-попаданий (hit rate),* велика и шинный трафик в расчете на один процессор резко уменьшается, позволяя включить в систему значительно больше процессоров. Общепринятыми являются размеры кэша от 512 Кбайт до 1 Мбайт, коэффициент кэш-попаданий при этом обычно составляет 90 % и более.

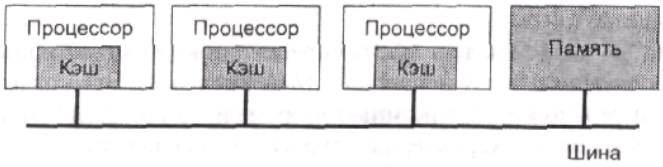


Рис. 5. Мультипроцессорная система с шинной архитектурой

Однако введение кэша создает серьезные проблемы само по себе. Пусть два процессора, *A* и *B,* читают одно и то же слово в свой внутренний кэш. Затем *A* перезаписывает это слово. Когда процессор *B* захочет воспользоваться этим словом, он считает старое значение из своего кэша, а не новое значение, записанное процессором *A*. Память стала несогласованной, и программирование системы осложнилось. Кэширование тем не менее активно используется в распределенных системах, здесь также сталкиваются с проблемами несогласованной памяти

Проблема мультипроцессорных систем шинной архитектуры состоит в их ограниченной масштабируемости, даже в случае использования кэша. Для построения мультипроцессорной системы с более чем 256 процессорами для соединения процессоров с памятью необходимы другие методы. Один из вариантов - разделить общую память на модули и связать их с процессорами через *коммутирующую решетку (crossbar switch),* как показано на рис. 6, *а.* Как видно из рисунка, с ее помощью каждый процессор может быть связан с любым модулем памяти. Каждое пересечение представляет собой маленький электронный *узловой коммутатор (crosspoint switch),* который может открываться и закрываться аппаратно. Когда процессор желает получить доступ к конкретному модулю памяти, соединяющие их узловые коммутаторы открываются, организуя запрошенный доступ. Достоинство узловых коммутаторов в том, что к памяти могут одновременно обращаться несколько процессоров, хотя если два процессора одновременно хотят получить доступ к одному и тому же участку памяти, то одному из них придется подождать.

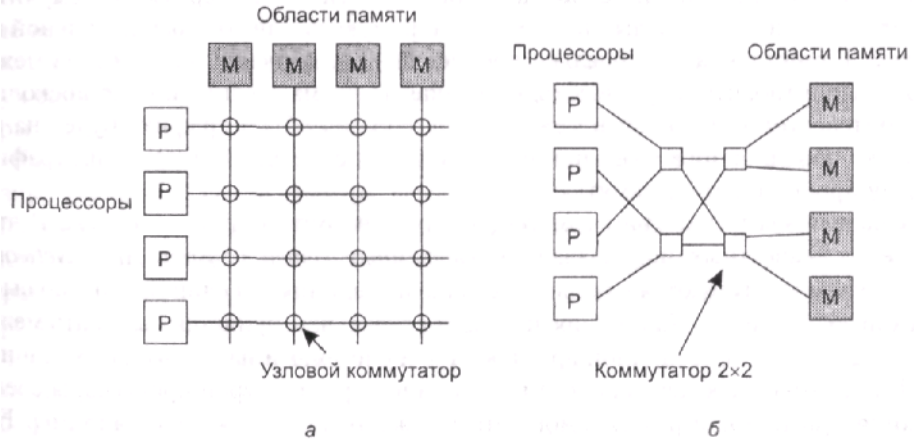


Рис. 6. Коммутирующая решетка (а). Коммутирующая омега-сеть (б)

Недостатком коммутирующей решетки является то, что при наличии n процессоров и n модулей памяти потребуется n^2 узловых коммутаторов. Для больших значений n это число может

превысить возможности. Поэтому были найдены альтернативные коммутирующие сети, требующие меньшего количества коммутаторов. Один из примеров таких сетей — *омега-сеть (omega network),* представленная на рис. 6, б. Эта сеть содержит четыре коммутатора 2x2, то есть каждый из них имеет по два входа и два выхода. Каждый коммутатор может соединять любой вход с любым выходом. Если внимательно изучить возможные положения коммутаторов, становится ясно, что любой процессор может получить доступ к любому блоку памяти. Недостаток коммутирующих сетей состоит в том, что сигнал, идущий от процессора к памяти или обратно, вынужден проходить через несколько коммутаторов. Поэтому, чтобы снизить задержки между процессором и памятью, коммутаторы должны иметь очень высокое быстродействие.

Пытаются уменьшить затраты на коммутацию путем перехода к иерархическим системам. В этом случае с каждым процессором ассоциируется некоторая область памяти. Каждый процессор может быстро получить доступ к своей области памяти. Доступ к другой области памяти происходит значительно медленнее. Эта идея была реализована в машине с *неунифицировапиым доступом к памяти (NonUniform Memory Access, NUMA).* Хотя машины NUMA имеют лучшее среднее время доступа к памяти, чем машины на базе омега-сетей, у них есть свои проблемы, связанные с тем, что размещение программ и данных необходимо производить так, чтобы большая часть обращений шла к локальной памяти.

Гомогенные мультикомпьютерные системы

В отличие от мультипроцессоров построить мультикомпыотерную систему относительно несложно. Каждый процессор напрямую связан со своей локальной памятью. Единственная оставшаяся проблема — это общение процессоров между собой. Необходима схема соединения, но поскольку интересна только связь между процессорами, объем трафика будет на несколько порядков ниже, чем при использовании сети для поддержания трафика между процессорами и памятью.

Рассмотрим гомогенные мультикомпьютерные системы. В этих системах, известных под названием *системных сетей (System Area Networks, SAN),* узлы монтируются в большой стойке и соединяются единой, обычно высокоскоростной сетью. Как и в предыдущем случае, необходимо выбирать между системами на основе шинной архитектуры и системами на основе коммутации.

В мультикомпьютерных системах с шинной архитектурой процессоры соединяются при помощи разделяемой сети множественного доступа, например Fast Ethernet. Скорость передачи данных в сети обычно равна 100 Мбит/с. Как и в случае мультипроцессоров с шинной архитектурой, мультикомпьютерные системы с шинной архитектурой имеют ограниченную масштабируемость. В зависимости от того, сколько узлов в действительности нуждаются в обмене данными, обычно не следует ожидать высокой производительности при превышении системой предела в 25-100 узлов.

В коммутируемых мультикомпьютерных системах сообщения, передаваемые от процессора к процессору, *маршрутизируются* в соединительной сети в отличие от принятых в шинной архитектуре широковещательных рассылок. Было предложено и построено множество различных топологий. Две популярные топологии — квадратные решетки и гиперкубы — представлены на рис. 7. Решетки просты для понимания и удобны для разработки на их основе печатных плат. Они прекрасно подходят для решения двухмерных задач, например задач теории графов или компьютерного зрения (глаза робота, анализ фотографий).

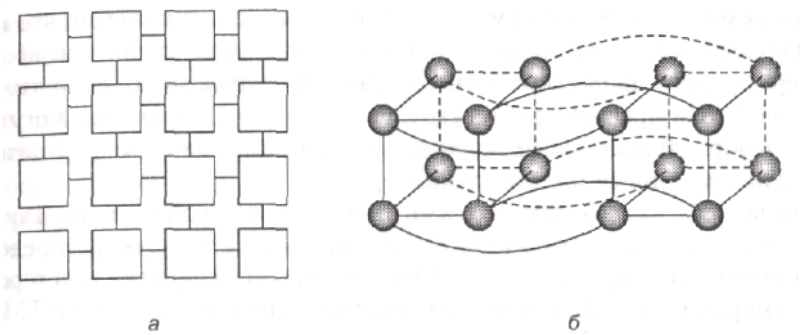


Рис. 7. Решетка (а). Гиперкуб (б)

Гиперкуб (hypercube) представляет собой куб размерности *n.* Гиперкуб, показанный на рис. 1.7, *б,* четырехмерен. Его можно представить в виде двух обычных кубов, с 8 вершинами и 12 ребрами каждый. Каждая вершина — это процессор. Каждое ребро — это связь между двумя процессорами. Соответствующие вершины обоих кубов соединены между собой. Для расширения гиперкуба в пятое измерение мы должны добавить к этой фигуре еще один комплект из двух связанных кубов, соединив соответствующие вершины двух половинок фигуры. Таким же образом можно создать шестимерный куб, семимерный и т. д.

Коммутируемые мультикомпьютерные системы могут быть очень разнообразны. На одном конце спектра лежат *процессоры с массовым параллелизмом (Massively Parallel Processors, MPP),* гигантские суперкомпьютеры стоимостью во много миллионов долларов, содержащие тысячи процессоров. Нередко они собираются из тех же процессоров, которые используются в рабочих станциях или персональных компьютерах. От других мультикомпьютерных систем их отличает наличие патентованных высокоскоростных соединительных сетей. Эти сети проектируются в расчете на малое время задержки и высокую пропускную способность. Кроме того, предпринимаются специальные меры для защиты системы от сбоев. При наличии тысяч процессоров каждую неделю несколько будут выходить из строя. Нельзя допустить, чтобы поломка одного из них приводила к выводу из строя всей машины.

На другом конце спектра мы обнаруживаем популярный тип коммутируемых микрокомпьютеров, известных как *кластеры рабочих станций (Clusters Of Work-Nations, COW),* основу которых составляют стандартные персональные компьютеры или рабочие станции, соединенные посредством коммерческих коммуникационных компонентов. Соединительные сети это то, что отличает COW от MPP. Кроме того, обычно не предпринимается никаких особых мер для повышения скорости ввода-вывода или защиты от сбоев в системе. Подобный подход делает COW проще и дешевле.

Гетерогенные мультикомпьютерные системы

Наибольшее число существующих в настоящее время распределенных систем построено по схеме гетерогенных мультикомпьютерных. Это означает, что компьютеры, являющиеся частями этой системы, могут быть очень разнообразны, например, по типу процессора, размеру памяти и производительности каналов ввода-вывода. На практике роль некоторых из этих компьютеров могут исполнять высокопроизводительные параллельные системы, например мультипроцессорные или гомогенные мультикомпьютерные. Соединяющая их сеть также может быть очень неоднородной.

Другим примером гетерогенности является создание крупных мультикомпьютерных систем с использованием существующих сетей и каналов. Так, например, не является чем-то необычным существование кампусных университетских распределенных систем, состоящих из локальных сетей различных факультетов, соединенных между собой высокоскоростными каналами. В глобальных системах различные станции могут, в свою очередь, соединяться общедоступными сетями, например сетевыми службами, предлагаемыми коммерческими операторами связи, например SMDS или Frame relay.

В отличие от систем, обсуждавшихся в предыдущих пунктах, многие крупномасштабные гетерогенные мультикомпьютерные системы нуждаются в глобальном подходе. Это означает, что приложение не может предполагать, что ему постоянно будет доступна определенная производительность или определенные службы.

Переходя к вопросам масштабирования, присущим гетерогенным системам, и учитывая необходимость глобального подхода, присущую большинству из них, следует заметить, что создание приложений для гетерогенных мультикомпьютерных систем требует специализированного программного обеспечения.

Концепции программных решений

Распределенные системы похожи на традиционные операционные системы. Они работают как *менеджеры ресурсов (resource managers)* существующего аппаратного обеспечения, которые помогают множеству пользователей и приложений совместно использовать такие ресурсы, как процессоры, память, периферийные устройства, сеть и данные всех видов. Более важно, что рас-

пределенная система скрывает сложность и гетерогенную природу аппаратного обеспечения, на базе которого она построена, предоставляя виртуальную машину для выполнения приложений.

Рассмотрим операционные системы с точки зрения распределенности. Операционные системы для распределенных компьютеров можно разделить на две категории — сильно связанные и слабо связанные системы. В сильно связанных системах операционная система в основном работает с одним, глобальным представлением ресурсов, которыми она управляет. Слабо связанные системы могут представляться набором операционных систем, каждая из которых работает на собственном компьютере. Однако эти операционные системы функционируют совместно, делая собственные службы доступными другим.

Это деление на сильно и слабо связанные системы связано с классификацией аппаратного обеспечения, приведенной в предыдущем разделе. Сильно связанные операционные системы обычно называются *распределенными операционными системами (Distributed Operating System, DOS)* и используются для управления мультипроцессорными и гомогенными мультикомпьютерными системами. Как и у традиционных однопроцессорных операционных систем, основная цель распределенной операционной системы состоит в сокрытии тонкостей управления аппаратным обеспечением, которое одновременно используется множеством процессов.

Слабо связанные *сетевые операционные системы (Network Operating Systems*) используются для управления гетерогенными мультикомпьютерными системами. Хотя управление аппаратным обеспечением и является основной задачей сетевых операционных систем, они отличаются от традиционных. Это отличие вытекает из того факта, что локальные службы должны быть доступными для удаленных клиентов.

Чтобы действительно составить распределенную систему, служб сетевой операционной системы недостаточно. Необходимо добавить к ним дополнительные компоненты, чтобы организовать лучшую поддержку прозрачности распределения. Этими дополнительными компонентами будут средства, известные как *системы промежуточного уровня (middleware),* которые и лежат в основе современных распределенных систем. В табл. 3 представлены основные данные по распределенным и сетевым операционным системам, а также средствам промежуточного уровня.

Система	Описание	Основное назначение
Распределенные операционные системы	Сильно связанные операционные системы для мультипроцессоров и гомогенных мультикомпьютерных систем	Сокрытие и управление аппаратным обеспечением
Сетевые операционные системы	Слабо связанные операционные системы для гетерогенных мультикомпьютерных систем (локальных или глобальных сетей)	Предоставление локальных служб удаленным клиентам
Средства промежуточного уровня	Дополнительный уровень поверх сетевых операционных систем, реализующий службы общего назначения	Обеспечение прозрачности распределения

Таблица 3. Краткое описание распределенных и сетевых операционных систем, а также средств промежуточного уровня

Распределенные операционные системы

Существует два типа распределенных операционных систем. Мультипроцессорная операционная система (multiprocessor operating system) управляет ресурсами мультипроцессора. Мультикомпьютерпая операционная система (multicomputer operating system) предназначена для гомогенных мультикомпъютеров. Функциональность распределенных операционных систем в основном не отличается от функциональности традиционных операционных систем, предназначенных для компьютеров с одним процессором за исключением того, что она поддерживает функционирование нескольких процессоров.

Операционные системы для однопроцессорных компьютеров Операционные системы традиционно строились для управления компьютерами с одним процессором. Основной задачей этих систем была организация легкого доступа пользователей и приложений к разделяемым устройствам, таким как процессор, память, диски и периферийные устройства. Говоря о разделении ресурсов, мы имеем в виду возможность использования одного и того же аппаратного обеспечения различными приложениями изолированно друг от друга. Для приложения это выглядит так, словно эти ресурсы находятся в его полном распоряжении, при этом в одной системе может выполняться одновременно несколько приложений, каждое со своим собственным набором ресурсов. В этом смысле говорят, что операционная система реализует виртуальную машину (virtual machine), предоставляя приложениям средства мультизадачности.

Важным аспектом совместного использования ресурсов в такой виртуальной машине является то, что приложения отделены друг от друга. Так, невозможна ситуация, когда при одновременном исполнении двух приложений, А и В, приложение А может изменить данные приложения В, просто работая с той частью общей памяти, где эти данные хранятся. Также требуется гарантировать, что приложения смогут использовать предоставленные им средства только так, как предписано операционной системой. Например, приложениям обычно запрещено копировать сообщения прямо в сетевой интерфейс. Взамен операционная система предоставляет первичные операции связи, которые можно использовать для пересылки сообщений между приложениями на различных машинах.

Следовательно, операционная система должна полностью контролировать использование и распределение аппаратных ресурсов. Поэтому большинство процессоров поддерживают как минимум два режима работы. В режиме ядра (kernel mode) выполняются все разрешенные инструкции, а в ходе выполнения доступна вся имеющаяся память и любые регистры. Напротив, в пользовательском режиме (user mode) доступ к регистрам и памяти ограничен. Так, приложению не разрешено работать с памятью за пределами набора адресов, установленного для него операционной системой, или обращаться напрямую к регистрам устройств. На время выполнения кода операционной системы процессор переключается в режим ядра. Однако единственный способ перейти из пользовательского режима в режим ядра — это сделать системный вызов, реализуемый через операционную систему. Поскольку системные вызовы

— это лишь базовые службы, предоставляемые операционной системой, и поскольку ограничение доступа к памяти и регистрам нередко реализуется аппаратно, операционная система в состоянии полностью их контролировать.

Существование двух режимов работы привело к такой организации операционных систем, при которой практически весь их код выполняется в режиме ядра. Результатом часто становятся гигантские монолитные программы, работающие в едином адресном пространстве. Оборотная сторона такого подхода состоит в том, что перенастроить систему часто бывает нелегко. Другими словами, заменить или адаптировать компоненты операционной системы без полной перезагрузки, а возможно и полной перекомпиляции и новой установки очень трудно. С точки зрения открытости, проектирования программ, надежности или легкости обслуживания монолитные операционные системы — это не самая лучшая из идей.

Более удобен вариант с организацией операционной системы в виде двух частей. Одна часть содержит набор модулей для управления аппаратным обеспечением, которые могут выполняться в пользовательском режиме. Например, управление памятью состоит в основном из отслеживания,

какие блоки памяти выделены под процессы, а какие свободны. Единственный момент, когда необходима в работе в режиме ядра, — это установка регистров блока управления памятью.

Вторая часть операционной системы содержит небольшое микроядро (microkernel), содержащее исключительно код, который выполняется в режиме ядра. На практике микроядро должно содержать только код для установки регистров устройств, переключения процессора с процесса на процесс, работы с блоком управления памятью и перехвата аппаратных прерываний. Кроме того, в нем обычно содержится код, преобразующий вызовы соответствующих модулей пользовательского уровня операционной системы в системные вызовы и возвращающий результаты. Такой подход приводит к организации, показанной на рис. 8.

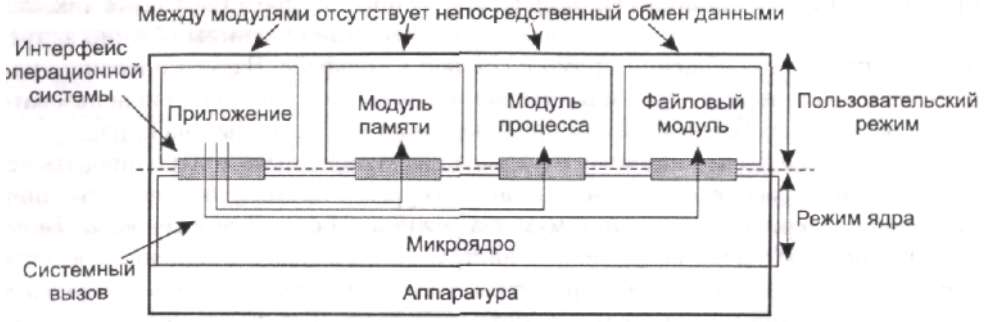


Рис. 8. Разделение приложений в операционной системе посредством микроядра

Использование микроядра дает разнообразные преимущества. Наиболее важное из них состоит в гибкости: поскольку большая часть операционной системы исполняется в пользовательском режиме, относительно несложно заменить один из модулей без повторной компиляции или повторной установки всей системы. Другое преимущество заключается в том, что модули пользовательского уровня могут размещаться на разных машинах. Можно установить модуль управления файлами не на той машине, на которой он управляет службой каталогов. Другими словами, подход с использованием микроядра отлично подходит для переноса однопроцессорных операционных систем на распределенные компьютеры.

У микроядер имеется два существенных недостатка. Во-первых, они работают иначе, чем существующие операционные системы. Во-вторых, микроядро требует дополнительного обмена, что слегка снижает производительность. Однако, снижение производительности в 20 % вряд ли можно считать фатальным.

Мультипроцессорные операционные системы

Важным, но часто не слишком очевидным расширением однопроцессорных операционных систем является возможность поддержки нескольких процессоров, имеющих доступ к совместно используемой памяти. Концептуально это расширение несложно. Все структуры данных, необходимые операционной системе для поддержки аппаратуры, включая поддержку нескольких процессоров, размещаются в памяти. Основное различие заключается в том, что данные доступны нескольким процессорам и должны быть защищены от параллельного доступа для обеспечения целостности.

Однако многие операционные системы, особенно предназначенные для персональных компьютеров и рабочих станций, не могут легко поддерживать несколько процессоров. Основная причина такого поведения состоит в том, что они были разработаны как монолитные программы, которые могут выполняться только в одном потоке управления. Адаптация таких операционных систем под мультипроцессорные означает повторное проектирование и новую реализацию его ядра. Современные операционные системы изначально разрабатываются с учетом возможности работы в мультипроцессорных системах.

Многопроцессорные операционные системы нацелены на поддержание высокой производительности конфигураций с несколькими процессорами. Основная их задача — обеспечить прозрачность числа процессоров для приложения. Сделать это достаточно легко, поскольку сообщение между различными приложениями или их частями требует тех же примитивов, что и в многозадачных однопроцессорных операционных системах. Сообщение происходит путем работы с данными в специальной совместно используемой области данных, и все что нужно — это защи-

тить данные от одновременного доступа к ним. Защита осуществляется посредством примитивов синхронизации. Два наиболее важных (и эквивалентных) примитива — это семафоры и мониторы.

Семафор (semaphore) может быть представлен в виде целого числа, поддерживающего две операции: up (увеличить) и down (уменьшить). При уменьшении сначала проверяется, превышает ли значение семафора 0. Если это так, его значение уменьшается и выполнение процесса продолжается. Если же значение семафора нулевое, вызывающий процесс блокируется. Оператор увеличения совершает противоположное действие. Сначала он проверяет все заблокированные в настоящее время процессы, которые были неспособны завершиться в ходе предыдущей операции уменьшения. Если таковые существуют, он разблокирует один из них и продолжает работу. В противном случае он просто увеличивает счетчик семафора. Разблокированный процесс выполняется до вызова операции уменьшения. Важным свойством операций с семафорами является то, что они атомарны (atomic), то есть в случае запуска операции уменьшения или увеличения до момента ее завершения (или до момента блокировки процесса) никакой другой процесс не может получить доступ к семафору.

Известно, что программирование с использованием семафоров для синхронизации процесса вызывает множество ошибок, кроме случаев простой защиты разделяемых данных. Основная проблема состоит в том, что наличие семафоров приводит к неструктурированному коду. Похожая ситуация возникает при частом использовании инструкции goto. В качестве альтернативы семафорам многие современные системы, поддерживающие параллельное программирование, предоставляют библиотеки для реализации мониторов.

Формально монитор (monitor) представляет собой конструкцию языка программирования, такую же, как объект в объектно-ориентированном программировании. Монитор может рассматриваться как модуль, содержащий переменные и процедуры. Доступ к переменным можно получить только путем вызова одной из процедур монитора. В этом смысле монитор очень похож на объект. Объект также имеет свои защищенные данные, доступ к которым можно получить только через методы, реализованные в этом объекте. Разница между мониторами и объектами состоит в том, что монитор разрешает выполнение процедуры только одному процессу в каждый момент времени. Другими словами, если процедура, содержащаяся в мониторе, выполняется процессом A (говорят, что A вошел в монитор) и процесс B также вызывает одну из процедур монитора, B будет блокирован до завершения выполнения A (то есть до тех пор, пока A не покинет монитор).

В качестве примера рассмотрим простой монитор для защиты целой переменной (листинг 1). Монитор содержит одну закрытую (private) переменную count, доступ к которой можно получить только через три открытых (public) процедуры — чтения текущего значения, увеличения на единицу и уменьшения. Конструкция монитора гарантирует, что любой процесс, который вызывает одну из этих процедур, получит атомарный доступ к внутренним данным монитора.

```
monitor Counter { private: int count = 0: public:
int valueC) { return count; }  void incrf) {  count =  count +  1 ;} void decr() { count = count - 1;}
}
```

Листинг 1. Монитор, предохраняющий целое число от параллельного доступа

Мониторы пригодны для простой защиты совместно используемых данных. Однако для условной блокировки процесса необходимо большее.

Предположим, что нужно заблокировать процесс при вызове операции уменьшения, если обнаруживается, что значение count равно нулю. Для этой цели в мониторах используются условные переменные (condition variables). Это специальные переменные с двумя доступными операциями: wait (ждать) и signal (сигнализировать). Когда процесс A находится в мониторе и вызывает для условной переменной, хранящейся в мониторе, операцию wait, процесс A будет заблокирован и откажется от своего исключительного доступа к монитору. Соответственно, процесс B, ожидавший получения исключительного доступа к монитору, сможет продолжить свою работу. В определенный момент времени B может разблокировать процесс A вызовом операции signal для условной переменной, которого ожидает A. Чтобы предотвратить наличие двух активных процессов внутри монитора, следует дополнить схему так, чтобы процесс, подавший сигнал, покидал монитор. Монитор из листинга 2 — это новая реализация обсуждавшегося ранее семафора.

```
monitor Counter { private: int count-0:
int blocked_procs = 0; condition unblocked: public: int valueO { return count;}
void inert) { if (blockecLprocs == 0) count - count + 1: else
signal( unblocked );
}
void decrC)      { If (count — 0) { blocked_procs = blocked_procs + 1; waitC unblocked );
blocked_procs = blocked_procs - 1: i / else count = count - 1;
} )
```

Листинг 2. Монитор, предохраняющий целое число от параллельного доступа и блокирующий процесс

Мониторы являются конструкциями языка программирования. Java поддерживает мониторы, разрешая каждому объекту предохранять себя от параллельного доступа путем использования в нем инструкции synchronized и операций wait и notify. Библиотечная поддержка мониторов обычно реализуется на базе простых семафоров, которые могут принимать только значения 0 и 1. Такие семафоры часто называются переменными-мьютсксами (mutex variables), или просто мьютексами. С мьютексами ассоциируются операции lock (блокировать) и unlock (разблокировать). Захват мьютекса возможен только в том случае, если его значение равно единице, в противном случае вызывающий процесс будет блокирован. Соответственно, освобождение мьютекса означает установку его значения в 1, если нет необходимости разблокировать какой- нибудь из ожидающих процессов. Условные переменные и соответствующие им операции также поставляются в виде библиотечных процедур.

Мультикомпьютерные операционные системы

Мультикомпьютерные операционные системы обладают гораздо более разнообразной структурой и значительно сложнее, чем мультипроцессорные. Эта разница проистекает из того, что структуры данных, необходимые для управления системными ресурсами, не должны отвечать условию легкости совместно использования, поскольку их не нужно помещать в физически общую память. Единственно возможным видом связи является передача сообщении (message passing). Мультикомпьютерные операционные системы в основном организованы так, как показано на рис. 9.

Каждый узел имеет свое ядро, которое содержит модули для управления локальными ресурсами памятью, локальным процессором, локальными дисками

Кроме того, каждый узел имеет отдельный модуль для межпроцессорного воздействия, то есть посылки сообщений на другие узлы и приема сообщении от них.

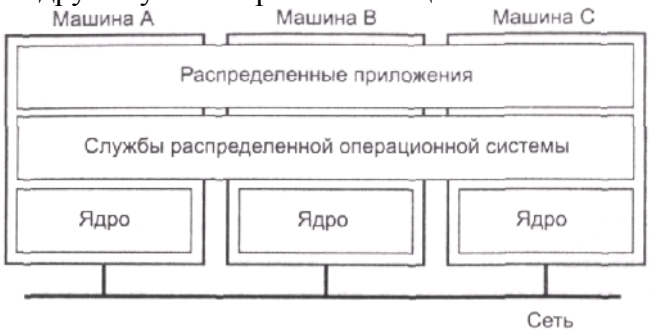


Рис. 9. Общая структура мультикомпьютерных операционных систем

Поверх каждого локального ядра лежит уровень программного обеспечения общего назначения, реализующий операционную систему в виде виртуальной машины, поддерживающей параллельную работу над различными задачами. Этот уровень может предоставлять абстракцию мультипроцессорной машины. Другими словами, он предоставляет полную программную реализацию совместно используемой памяти. Дополнительные средства, обычно реализуемые на этом уровне, предназначены, например, для назначения задач процессорам, маскировки сбоев аппаратуры, обеспечения прозрачности сохранения и общего обмена между процессами. Эти средства типичны для операционных систем.

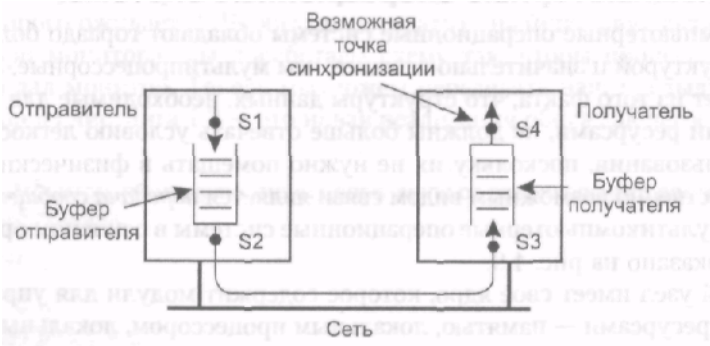


Рис. 10. Возможности блокировки и буферизации при пересылке сообщений

Мультикомпьютерные операционные системы, не предоставляющие средств для совместного использования памяти, могут предложить приложениям только средства для обмена сообщениями. К сожалению, семантика примитивов обмена сообщениями в значительной степени разная для разных систем. Понять эти различия проще, если отмечать, буферизуются сообщения или нет. Кроме того, необходимо учитывать, блокируется ли посылающий или принимающий процесс. На рис. 10 продемонстрирован вариант с буферизацией и блокировкой.

Существует два возможных места буферизации сообщений — на стороне отправителя или на стороне получателя. Это приводит к четырем возможным точкам синхронизации, то есть точкам возможной блокировки отправителя или получателя. Если буферизация происходит на стороне отправителя, это дает возможность заблокировать отправителя, только если его буфер полон, что показано точкой синхронизации S1 на рисунке. С другой стороны, процедура помещения сообщения в буфер может возвращать состояние, показывающее, что операция успешно выполнена. Это позволяет отправителю избежать блокировки по причине переполнения буфера. Если же отправитель не имеет буфера, существует три альтернативных точки блокировки отправителя: отправление сообщения (точка S2), поступление сообщения к получателю (точка S3), принятие сообщения получателем (точка S4), Если блокировка происходит в точке S2, S3 или S4, наличие или отсутствие буфера на стороне отправителя не имеет значения.

Блокировка получателя имеет смысл только в точке синхронизации S3 и может производиться, только если у получателя нет буфера или если буфер пуст. Альтернативой может быть опрос получателем наличия входящих сообщений. Однако эти действия часто ведут к пустой трате процессорного времени или слишком запоздалой реакции на пришедшее сообщение, что, в свою очередь, приводит к переполнению буфера входящими сообщениями и их потере.

Другой момент, важный для понимания семантики обмена сообщениями,

- надежность связи. Отличительной чертой надежной связи является получение отправителем гарантии приема сообщения. На рис. 10 надежность связи означает, что все сообщения гарантированно достигают точки синхронизации S4. При ненадежной связи всякие гарантии отсутствуют. Если буферизация производится на стороне отправителя, о надежности связи ничего определенного сказать нельзя. Также операционная система не нуждается в гарантированно надежной связи в случае блокировки отправителя в точке S2.

С другой стороны, если операционная система блокирует отправителя до достижения сообщением точки S3 или S4, она должна иметь гарантированно надежную связь, В противном случае можно оказаться в ситуации, когда отправитель ждет подтверждения получения, а сообщение было потеряно при передаче. Отношение между блокировкой, буферизацией и гарантиями относительно надежности связи суммированы в табл. 4.

Точка синхронизации	Буферизация отправителя	Гарантия надежной связи
Блокировка отправителя до наличия свободного места в буфере	Да	Нет необходимости

Блокировка отправителя до посылки сообщения	Нет	Нет необходимости
Блокировка отправителя до приема сообщения	Нет	Необходима
Блокировка отправителя до обработки сообщения	Нет	Необходима

Таблица 4. Соотношение между блокировкой, буферизацией и надежностью связи

Множество аспектов проектирования мультикомпьютерных операционных систем одинаково важны для любой распределенной системы. Основная разница между мультикомпьютерными операционными системами и распределенными системами состоит в том, что в первом случае обычно подразумевается, что аппаратное обеспечение гомогенно и полностью управляемо. Множество распределенных систем, однако, строится на базе существующих операционных систем.

Системы с распределенной разделяемой памятью

Практика показывает, что программировать мультикомпьютерные системы значительно сложнее, чем мультипроцессорные. Разница объясняется тем, что связь посредством процессов, имеющих доступ к совместно используемой памяти, и простых примитивов синхронизации, таких как семафоры и мониторы, значительно проще, чем работа с одним только механизмом обмена сообщениями. Такие вопросы, как буферизация, блокировка и надежность связи, только усложняют положение.

По этой причине проводились исследования по вопросу эмуляции совместно используемой памяти на мультикомпьютерных системах. Их целью было создание виртуальных машин с разделяемой памятью, работающих на мультикомпьютерных системах, для которых можно было бы писать приложения, рассчитанные на модель совместно используемой памяти, даже если физически она отсутствует. Главную роль в этом играет мультикомпьютерная операционная система.

Один из распространенных подходов — задействовать виртуальную память каждого отдельного узла для поддержки общего виртуального адресного пространства. Это приводит нас к распределенной разделяемой памяти (Distributed Shared Метол/, DSM) со страничной организацией. Принцип работы этой памяти следующий. В системе с DSM адресное пространство разделено на страницы (обычно по 4 или по 8 Кбайт), распределенные по всем процессорам системы. Когда процессор адресуется к памяти, которая не является локальной, происходит внутреннее прерывание, операционная система считывает в локальную память страницу, содержащую указанный адрес, и перезапускает выполнение вызвавшей прерывание инструкции, которая теперь успешно выполняется. Этот подход продемонстрирован на рис. 11, а для адресного пространства из 16 страниц и четырех процессоров. Это вполне нормальная страничная организации, если не считать того, что в качестве временного хранилища информации используется не диск, а удаленная оперативная память.

В этом примере при обращении процессора 1 к коду или данным со страницы 0, 2, 5 или 9 обращение происходит локально. Ссылки на другие страницы вызывают внутреннее прерывание. Так, например, ссылка на адрес со страницы 10 вызывает внутреннее прерывание операционной системы, и она перемещает страницу 10 с машины 2 на машину 1, как показано на рис. 11, б.

Одно из улучшений базовой системы, часто позволяющее значительно повысить ее производительность, — это репликация страниц, которые объявляются закрытыми на запись, например, страниц, содержащих текст программы, констант, «топько для чтения» или другие закрытые на запись структуры. Например, ести страница 10 — это секция текста программы, ее использование процессором 1 приведет к пересылке процессору 1 ее копии, а оригинал в памяти процессора 2 будет продолжать спокойно храниться, как показано на рис. 11, в. В этом случае процессоры 1 и 2 оба смогут обращаться к странице 10, не вызывая при этом никаких внутренних прерываний для выборки памяти.

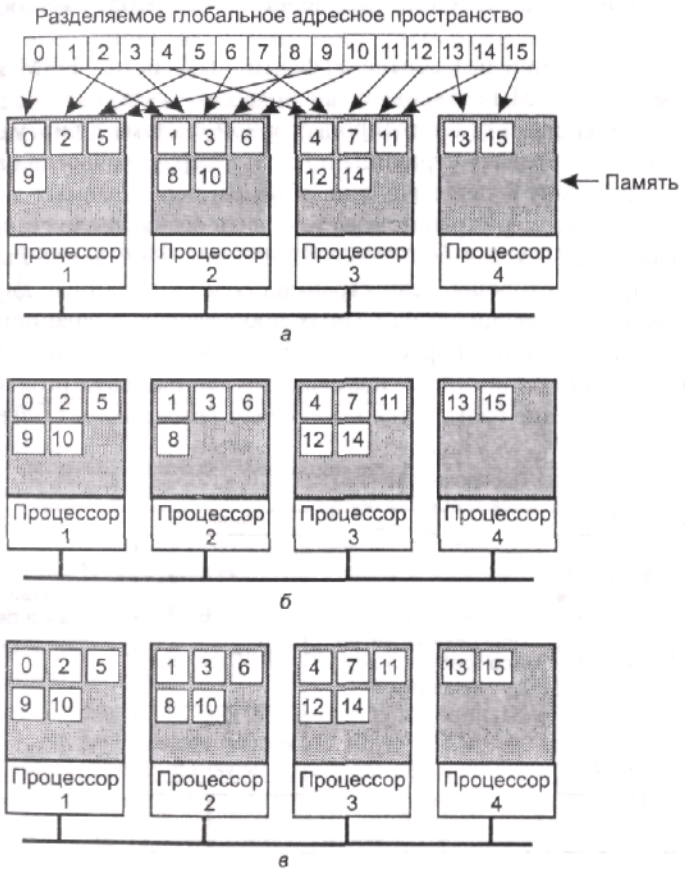


Рис. 11. Страницы адресного пространства распределены по четырем машинам (а).

Ситуация после обращения процессора 1 к странице 10 (б). Ситуация, когда запись в страницу 10 невозможна и необходима репликация (в)

Другая возможность — это репликация также и не закрытых на запись страсть любых страниц. Пока производится только чтение, никакой разницы между репликацией закрытых и незакрытых на запись страниц нет. Однако реплицированная когда страница внезапно изменяется, необходимо предпринимать специальные действия для предотвращения появления множества несовместимых копий. Обычно все копии, кроме одной, перед проведением записи объявляются неверными.

Дополнительного увеличения производительности можно добиться путем ухода от строгого соответствия между реплицируемыми страницами. Другими слонами, позволяют отдельной копии временно отличаться от других. Практика показывает, что этот подход действительно может помочь, но, может потребовать отслеживать возможную несовместимость. Поскольку основной причиной разработки DSM была простота программирования, ослабление соответствия не находит реального применения.

Другой проблемой при разработке эффективных систем DSM является вопрос о размере страниц. Затраты па передачу страницы по сети в первую очередь определяются затратами на подготовку к передаче, а не объемом передаваемых данных. Соответственно, большой размер страниц может уменьшить общее число сеансов передачи при необходимости доступа к большому количеству последовательных элементов данных. С другой стороны, если страница содержит данные двух независимых процессов, выполняющихся на разных процессорах, операционная система будет вынуждена постоянно пересылать эту страницу от одного процессора к другому, как показано на рис.

12. Размещение данных двух независимых процессов на одной странице называется ошибочным разделением (false sharing).

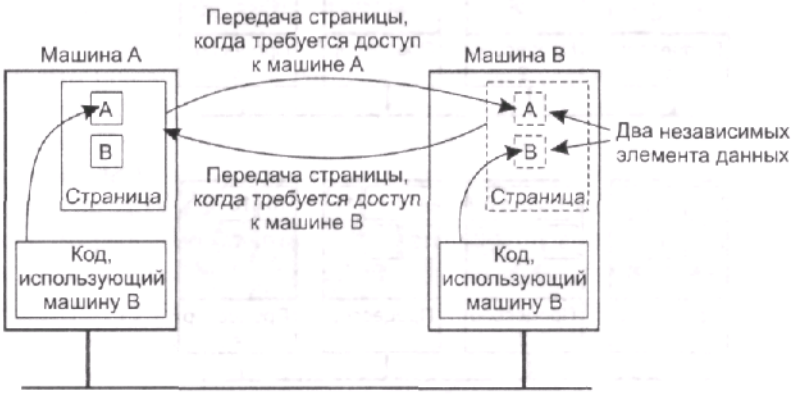


Рис. 12. Ошибочное разделение страницы двумя независимыми процессами

Для достижения высокой производительности крупномасштабных мультикомпьютерных систем прибегают к пересылке сообщений, невзирая на ее высокую, по сравнению с программированием систем (виртуальной) памяти совместного использования, сложность. Это позволяет сделать вывод о том, что DSM не оправдывает ожиданий, требуя высокопроизводительного параллельного программирования.

Сетевые операционные системы

В противоположность распределенным операционным системам сетевые операционные системы не нуждаются в том, чтобы аппаратное обеспечение, на котором они функционируют, было гомогенно и управлялось как единая система. Обычно они строятся для набора однопроцессорных систем, каждая из которых имеет собственную операционную систему, как показано на рис. 13. Машины и их операционные системы могут быть разными, но все они соединены в сеть. Кроме того, сетевая операционная система позволяет пользователям использовать службы, расположенные на конкретной машине. Возможно, будет проще описать сетевую операционную систему, кратко рассмотрев службы, которые она обычно предоставляет.

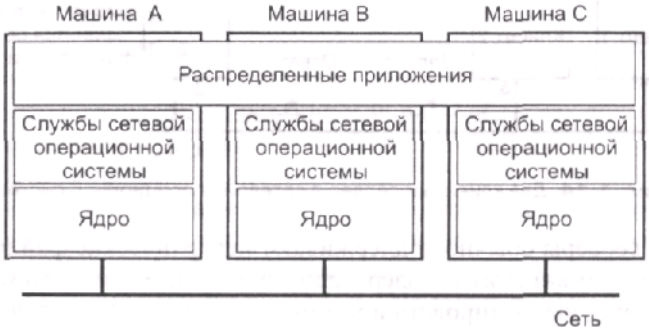


Рис. 1.13. Общая структура сетевой операционной системы

Служба, обычно предоставляемая сетевыми операционными системами, должна обеспечивать удаленное соединение пользователя с другой машиной путем применения команды.

В результате выполнения этой команды происходит переключение рабочей станции пользователя в режим удаленного терминала, подключенного к удаленной машине. Это означает, что пользователь сидит у графической рабочей станции, набирая команды на клавиатуре. Команды передаются на удаленную машину, результаты с удаленной машины отображаются в окне на экране пользователя.

Для того чтобы переключиться на другую удаленную машину, необходимо открыть новое окно и воспользоваться командой rlogin для соединения с другой машиной. Выбор удаленной машины производится вручную.

Сетевые операционные системы также имеют в своем составе команду удаленного копирования для копирования файлов с одной машины на другую. При этом перемещение файлом задастся в явном виде, и пользователю необходимо точно знать, где находятся файлы и как выполняются команды.

Такая форма связи примитивна. Это заставило проектировщиков систем искать более удобные варианты связи и совместного использования информации. Один из подходов предполагает

создание глобальной общей файловой системы, доступной со всех рабочих станций. Файловая система поддерживается одной или несколькими машинами, которые называются файловыми серверами (file servers). Файловые серверы принимают запросы от программ пользователей, запускаемых на других машинах (не на серверах), которые называются клиентами (clients), на чтение и запись файлов. Каждый пришедший запрос проверяется и выполняется, а результат пересылается назад, как показано на рис. 14.

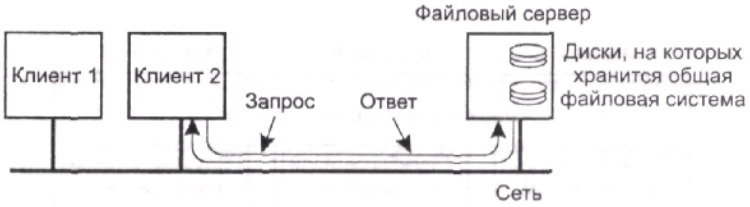


Рис. 14. Два клиента и сервер в сетевой операционной системе

Файловые серверы обычно поддерживают иерархические файловые системы, каждая с корневым каталогом, содержащим вложенные каталоги и файлы. Рабочие станции могут импортировать или монтировать эти файловые системы, увеличивая свою локальную файловую систему за счет файловой системы сервера. На рис. 15 показаны два файловых сервера. На одном из них имеется каталог под названием games, а на другом — каталог под названием work (имена каталогов выделены жирным шрифтом). Каждый из этих каталогов содержит некоторые файлы. На обоих клиентах смонтированы файловые системы обоих серверов, но в разных местах файловых систем клиентов. Клиент 1 смонтировал их в свой корневой каталог и имеет к ним доступ по путям /games и /work соответственно. Клиент 2, подобно Клиенту 1, смонтировал каталог work в свой корневой каталог, но решил, что игры (games) должны быть его частным делом. Поэтому он создал каталог, который назвал /private, и смонтировал каталог games туда. Соответственно, он получит доступ к файлу packwoman через путь /private/games/packwoman, а не /games/packwoman.

Хотя обычно не имеет значения, в какое место своей иерархии каталогов клиент смонтировал сервер, важно помнить, что различные клиенты могут иметь различное представление файловой системы. Имя файла зависит от того, как организуется доступ к нему и как выглядит файловая система на самой машине, Поскольку каждая клиентская машина работает относительно независимо от других, невозможно дать гарантии, что они обладают одинаковой иерархией каталогов для своих программ.

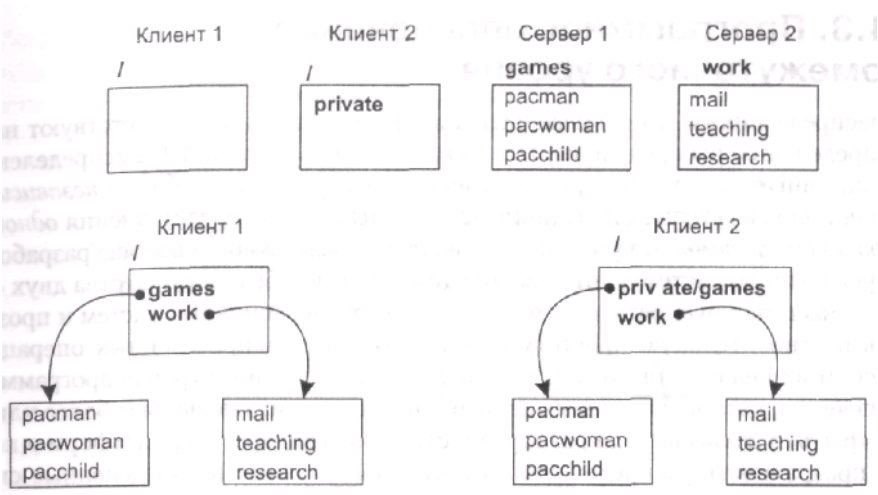


Рис. 15. Различные клиенты могут монтировать файловые системы серверов по-разному

Сетевые операционные системы выглядят примитивнее распределенных. Основная разница между этими двумя типами операционных систем состоит в том, что в распределенных операционных системах делается серьезная попытка добиться полной прозрачности, то есть создать представление единой системы.

«Нехватка» прозрачности в сетевых операционных системах имеет некоторые очевидные обратные стороны. Например, с ними часто сложно работать, поскольку пользователь вынужден

явно подсоединяться к удаленным машинам или копировать файлы с одной машины па другую. Это вызывает также проблемы с управлением. Поскольку все машины под управлением сетевой операционной системы независимы, часто и управлять ими можно исключительно независимо. В результате пользователь может получить удаленное соединение с машиной X, только имея на ней регистрацию. Таким образом, если пользователь хочет использовать один пароль на «все случаи жизни», то для смены пароля он вынужден будет явно сменить его на каждой машине. В основном все права доступа относятся к конкретной машине. Нет простого метода сменить права доступа, поскольку всюду они свои. Такой децентрализованный подход к безопасности затрудняет защиту сетевой операционной системы от атак злоумышленников.

Имеются также и преимущества по сравнению с распределенными операциями системами. Поскольку узлы сетевых операционных систем в значительной степени независимы друг от друга, добавить или удалить машину очень легко. В некоторых случаях все, что надо сделать, чтобы добавить узел, — это подсоединить соответствующую машину к общей сети и поставить в известность о ее существовании остальные машины сети. В Интернете, например, добавление нового компьютера происходит именно так. Чтобы сведения о машине попали в Интернет, необходимо дать ей сетевой адрес, а лучше символическое имя, которое будет внесено в DNS вместе с ее сетевым адресом.

Программное обеспечение промежуточного уровня

Ни распределенные, ни сетевые операционные системы не соответствуют определению распределенных систем, данному в разделе 1. Распределенные операционные системы не предназначены для управления набором независимых компьютеров, а сетевые операционные системы не дают представления одной согласованной системы. Возникает вопрос: а возможно ли вообще разработать распределенную систему, которая объединяла бы в себе преимущества двух «миров» — масштабируемость и открытость сетевых операционных систем, и прозрачность и относительную простоту в использовании распределенных операционных систем? Решение было найдено в виде дополнительного уровня программного обеспечения, который в сетевых операционных системах позволяет более или менее скрыть от пользователя разнородность набора аппаратных платформ и повысить прозрачность распределения. Многие современные распределенные системы построены в расчете на этот дополнительный уровень, который получил название программного обеспечения промежуточного уровня.

Позиционирование программного обеспечения промежуточного уровня

Многие распределенные приложения допускают непосредственное использование программного интерфейса, предлагаемого сетевыми операционными системами. Так, связь часто реализуется через операции с сокетами, которые позволяют процессам на разных машинах обмениваться сообщениями. Кроме того, приложения часто пользуются интерфейсами локальных файловых систем. Проблема такого подхода состоит в том, что наличие распределения слишком очевидно. Решение заключается в том, чтобы поместить между приложением и сетевой операционной системой промежуточный уровень программной поддержки, обеспечивающий дополнительное абстрагирование. Этот уровень называется промежуточным. Он находится посредине между приложением и сетевой операционной системой, как показано на рис. 16.

Каждая локальная система, составляющая часть базовой сетевой операционной системы, предоставляет управление локальными ресурсами и простейшие коммуникационные средства для связи с другими компьютерами. Другими словами, программное обеспечение промежуточного уровня не управляет каждым узлом, эта работа по-прежнему приходится на локальные операционные системы.

Основная наша задача — скрыть разнообразие базовых платформ от приложений. Для решения этой задачи многие системы промежуточного уровня предоставляют более или менее полные наборы служб и «не позволяют использовать иные средства для доступа к этим службам, кроме своих интерфейсов. Другими словами, обход промежуточного уровня и непосредственный вызов служб одной из базовых операционных систем считается не приемлемым.

Рис. 16. Общая структура распределенных систем с промежуточным уровнем

После появления и широкого распространения сетевых операционных систем многие организации обнаружили, что у них накопилась масса сетевых приложений, которые невозможно легко интегрировать в единую систему. Тогда производители начали создавать независимые от приложений службы верхнего уровня для этих систем. Типичные примеры обеспечивали поддержку распределенного взаимодействия и улучшенные коммуникационные возможности.

Была создана организация, призванная определить общий стандарт для решений на базе промежуточного уровня. К настоящему времени существует множество таких стандартов. Стандарты в основном несовместимы друг с другом, и продукты разных производителей, реализующие один и тот же стандарт, редко способны работать вместе. Вероятно, что это ненадолго.

Модели промежуточного уровня

Чтобы сделать разработку и интеграцию распределенных приложений как можно более простой, основная часть программного обеспечения промежуточного уровня базируется на некоторой модели, или парадигме, определяющей распределение и связь. Относительно простой моделью является представление всех наблюдаемых объектов в виде файлов. Этот подход был изначально введен в UNIX. Подобный же подход применяется в программном обеспечении промежуточного уровня, построенном по принципу распределенной файловой системы (distributed file system). Во многих случаях это программное обеспечение недалеко ушло от сетевых операционных систем в том смысле, что прозрачность распределения поддерживается только для стандартных файлов (то есть файлов, предназначенных только для храпения данных). Процессы часто должны запускаться исключительно на определенных машинах. Программное обеспечение промежуточного уровня, основанное на модели распределенной файловой системы, оказалось достаточно легко масштабируемым, что способствовало его популярности.

Другая важная ранняя модель программного обеспечения промежуточного уровня основана на удаленных вызовах процедур (Remote Procedure Calls, RPC). В этой модели акцент делается на сокрытии сетевого обмена за счет того, что процессу разрешается вызывать процедуры, реализация которых находится на удаленной машине. При вызове такой процедуры параметры прозрачно передаются на удаленную машину, где, собственно, и выполняется процедура, после чего результат выполнения возвращается в точку вызова процедуры. За исключением, вероятно, некоторой потери производительности, все это выглядит как локальное исполнение вызванной процедуры: вызывающий процесс не уведомляется об имевшем место факте сетевого обмена.

Если вызов процедуры проходит через границы отдельных машин, он может быть представлен в виде прозрачного обращения к объекту, находящемуся на удаленной машине. Это привело к появлению разнообразных систем промежуточного уровня, реализующих представление о распределенных объектах (distributed objects). Идея распределенных объектов состоит в том, что каждый объект реализует интерфейс, который скрывает все внутренние детали объекта от его пользователя. Интерфейс содержит методы, реализуемые объектом. Процесс видит интерфейс.

Распределенные объекты часто реализуются путем размещения объекта на одной из машин и открытия доступа к его интерфейсу с множества других. Когда процесс вызывает метод, реализация интерфейса на машине с процессом преобразует вызов метода в сообщение, пересылаемое объекту. Объект выполняет запрашиваемый метод и отправляет назад результаты. Затем реализа-

ция интерфейса преобразует ответное сообщение в возвращаемое значение, которое передается вызвавшему процессу. Как и в случае с RPC, процесс может оказаться не уведомленным об этом обмене.

Модели могут упростить использование сетевых систем. Наилучшим образом это видно на примере World Wide Web. Успех среды Web в основном определяется тем, что она построена на базе простой, но высокоэффективной модели распределенных документов (distributed documents). В модели, принятой в Web, информация организована в виде документов, каждый из которых размещен на машине, расположение которой абсолютно прозрачно. Документы содержат ссылки, связывающие текущий документ с другими. Если следовать по ссылке, то документ, с которым связана эта ссылка, будет извлечен из места его хранения и выведен на экран пользователя. Концепция документа не ограничивается исключительно текстовой информацией. В Web поддерживаются аудио- и видеодокументы, а также различные виды документов на основе интерактивной графики.

Службы промежуточного уровня

Существует некоторое количество стандартных для систем промежуточного уровня служб. Все программное обеспечение промежуточного уровня должно реализовывать прозрачность доступа путем предоставления высокоуровневых средств связи (communication facilities), скрывающих низкоуровневую пересылку сообщений по компьютерной сети. Интерфейс программирования транспортного уровня, предоставляемый сетевой операционной системой, полностью заменяется другими средствами. Способ, которым поддерживается связь, в значительной степени зависит от модели распределения, предлагаемой программным обеспечением промежуточного уровня пользователям и приложениям. Сюда относится удаленный вызов процедур и обращение к распределенным объектам. Кроме того, многие системы промежуточного уровня предоставляют средства для прозрачного доступа к удаленным данным, такие как распределенные файловые системы или распределенные базы данных. Прозрачная доставка документов, реализуемая в Web, — это еще один пример коммуникаций высокого уровня (однонаправленных).

Важная служба, общая для всех систем промежуточного уровня, — это именование (naming). Службы именования сравнимы с телефонными книгами или справочниками типа «Желтых страниц». Они позволяют совместно использовать и искать сущности (как в каталогах). Однако при масштабировании возникают серьезные трудности. Проблема состоит в том,что для эффективного поиска имени в большой системе местоположение разыскиваемой сущности должно считаться фиксированным. Такое допущение принято в среде World Wide Web, в которой любой документ поименован посредством URL. URL содержит имя сервера, на котором находится документ с данным URL-адресом. Таким образом, если документ переносится на другой сервер, его URL изменяется.

Многие системы промежуточного уровня предоставляют специальные средства хранения данных, также именуемые средствами сохранности (persistence).В простейшей форме сохранность обеспечивается распределенными файловыми системами, но более совершенное программное обеспечение промежуточного уровня содержит интегрированные базы данных или предоставляет средства для связи приложений с базами данных.

Если хранение данных играет важную роль для оболочки, то обычно предоставляются средства для распределенных транзакций (distributed transactions). Применение транзакций дает возможность применения множества операций чтения и записи в ходе одной атомарной операции. Под атомарностью мы понимаем то, что транзакция может быть либо успешной (когда все операции записи завершаются успешно), либо неудачной, что оставляет все задействованные данные не измененными. Распределенные транзакции работают с данными, которые, возможно, разбросаны по нескольким машинам. Предоставление таких служб, как распределенные транзакции, особенно важно, поскольку маскировка сбоев для распределенных систем нередко затруднена. К сожалению, транзакции легче масштабировать на нескольких

географически удаленных машинах, чем на множестве локальных.

Практически все системы промежуточного уровня предоставляют средства обеспечения защиты (security). По сравнению с сетевыми операционными системами проблема защиты в системах промежуточного уровня состоит в том, что они распределены. Промежуточный уровень в принципе не может «надеяться» на то, что базовые локальные операционные системы будут адекватно обеспечивать защиту всей сети. Соответственно, защита отчасти ложится на программное обеспечение промежуточного уровня. В сочетании с требованием расширяемости защита превращается в одни из наиболее трудно реализуемых в распределенных системах служб.

Промежуточный уровень и открытость

Современные распределенные системы обычно создаются в виде систем промежуточного уровня для нескольких платформ. При этом приложения создаются для конкретной распределенной системы и не зависят от платформы (операционной системы). К сожалению, эта независимость часто заменяется жесткой зависимостью от конкретной системы промежуточного уровня. Проблема заключается в том, что системы промежуточного уровня часто значительно менее открыты, чем требуется.

Открытая распределенная система определяется полнотой (завершенностью) ее интерфейса. Полнота означает реальное наличие всех необходимых для создания систем описаний. Неполнота описания интерфейса приводит к тому, что разработчики систем вынуждены добавлять свои собственные интерфейсы. Разными командами разработчиков в соответствии с одним и тем же стандартом создаются разные системы промежуточного уровня и приложения, написанные под одну из систем, не могут быть непосредственно перенесены под другую без значительных усилий.

Неполнота приводит к невозможности совместной работы двух реализаций, несмотря на то, что они поддерживают абсолютно одинаковый набор интерфейсов, но различные базовые протоколы. Так, если две реализации основаны на несовместимых коммуникационных протоколах, поддерживаемых сетевой операционной системой, не легко добиться их совместной работы. Необходимо, чтобы и протоколы промежуточного уровня, и его интерфейсы были одинаковы, как показано на рис. 17.

Для гарантии совместной работы различных реализаций необходимо, чтобы к сущностям разных систем можно было одинаково обращаться Если к сущностям в одной системе обращение идет через URL, а в другой системе — через сетевой адрес, перекрестные обращения приведут к проблемам. В подобных случаях определения интерфейсов должны точно предписывать вид ссылок.

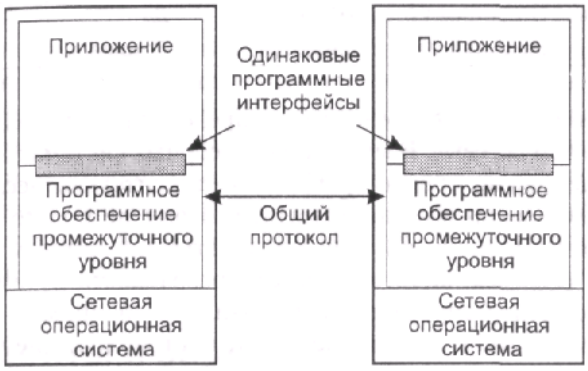


Рис. 1.17. В открытых распределенных системах должны быть одинаковыми как протоколы, используемые промежуточными уровнями каждой из систем, так и интерфейсы, предоставляемые приложениям

Сравнение систем

Краткое сравнение распределенных операционных систем, сетевых операционных систем и распределенных систем промежуточного уровня приведено в табл. 5.

Характеристика	Распределенная операционная система	Сетевая операционная	Распределенная система

	мультипро цессорная	мульти-компьютерн ая	система	промежуточного уровня
Степень прозрачности	Очень высокая	Высокая	Низкая	Высокая
Идентичность операционной системы на всех узлах	Поддержив ается	Поддержива ется	Не поддерживается	Не поддерживается
Число копий ОС	1	N	N	N
Коммуникации на основе	Совместно используем ой памяти	Сообщений	Файлов	В зависимости от модели
Управление ресурсами	Глобальное ,централизо ванное	Глобальное, распределен ное	Отдельно на узле	Отдельно на узле
Масштабируемос ть	Отсутствуе т	Умеренная	Да	Различная
Открытость	Закрытая	Закрытая	Открытая	Открытая

Таблица 5. Сравнение распределенных операционных систем, сетевых операционных систем и распределенных систем промежуточного уровня

Распределенные операционные системы работают лучше, чем сетевые. В мультипроцессорных системах нужно скрывать только общее число процессоров. Сложнее скрыть физическое распределение памяти, поэтому не просто создать мультикомпьютерную операционную систему с полностью прозрачным распределением. Распределенные системы часто повышают прозрачность путем использования специальных моделей распределения и связи. Распределенные файловые системы хорошо скрывают местоположение файлов и доступ к ним. Однако они теряют в общности, и пользователи могут получить проблемы с некоторыми приложениями.

Распределенные операционные системы гомогенны, то есть каждый узел имеет собственную операционную систему (ядро). В мультипроцессорных системах нет необходимости копировать данные — таблицы и пр., поскольку все они находятся в общей памяти и могут использоваться совместно. В этом случае вся связь также осуществляется через общую память, в то время как в мультикомпьютерных системах требуются сообщения. В сетевых операционных системах связь чаще всего базируется на файлах. Например, в Интернете большая часть обмена осуществляется путем передачи файлов. Кроме того, однако, интенсивно используется обмен сообщениями высокого уровня в виде систем электронной почты и досок объявлений. Связь в распределенных системах промежуточного уровня зависит от модели, на которой основана система.

Ресурсы в сетевых операционных системах и распределенных системах промежуточного уровня управляются на каждом узле, что делает масштабирование этих систем относительно простым. Однако практика показывает, что реализация в распределенных системах программного обеспечения промежуточного уровня часто приводит к ограниченности масштабирования. Распределенные операционные системы осуществляют глобальное управление ресурсами, что усложняет их масштабирование. В связи с централизованным подходом (когда все данные находятся в общей памяти) в мультипроцессорных системах они плохо масштабируются.

Сетевые операционные системы и распределенные системы промежуточного уровня выигрывают в открытости. В основном узлы поддерживают стандартный коммуникационный протокол типа TCP/IP, что упрощает организацию их совместной работы. Однако могут встретиться трудности с переносом приложений под разные платформы. Распределенные операционные системы в основном рассчитаны не на открытость, а на максимальную производительность.

Модель клиент-сервер

Клиенты и серверы

В базовой модели клиент-сервер все процессы в распределенных системах делятся на две возможно перекрывающиеся группы. Процессы, реализующие некоторую службу, например службу файловой системы или базы данных, называются серверами (servers). Процессы, запрашивающие службы у серверов путем посылки запроса и последующего ожидания ответа от сервера, называются клиентами (clients). Взаимодействие клиента и сервера, известное также под названием режим работы запрос-ответ (request-reply behavior), иллюстрирует рис. 18.

Рис. 18. Обобщенное взаимодействие между клиентом и сервером

Если базовая сеть надежна, как локальные сети, взаимодействие между клиентом и сервером может быть реализовано посредством простого протокола, не требующего установления соединения. В этом случае клиент, запрашивая сервер, облекает свой запрос в форму сообщения с указанием в нем службы, которой он желает воспользоваться, и необходимых для этого исходных данных. Сообщение посылается серверу. Последний, в свою очередь, постоянно ожидает исходящего сообщения, получив его, обрабатывает, упаковывает результат обработки в ответное сообщение и отправляет его клиенту. Использование не требующего соединения протокола дает существенный выигрыш эффективности. До тех пор пока сообщения не начнут пропадать или искажаться, можно вполне успешно применять протокол типа запрос-ответ. Создать протокол, устойчивый к случайным сбоям связи, — нетривиальная задача. Можно дать клиенту возможность повторно послать запрос, на который не был получен ответ. Проблема, однако, состоит в том, что клиент не может определить, действительно ли первоначальное сообщение с запросом было потеряно или ошибка произошла при передаче ответа. Если потерялся ответ, повторная посылка запроса может привести к повторному выполнению операции.

В качестве альтернативы во многих системах клиент-сервер используется надежный протокол с установкой соединения. Хотя это решение в связи с его относительно низкой производительностью не подходит для локальных сетей, оно используется в глобальных системах, для которых ненадежность является свойством соединений. Практически вес прикладные протоколы Интернета основаны на надежных соединениях по протоколу TCP/IP. В этих случаях всякий раз, когда клиент запрашивает службу, до посылки запроса серверу он должен установить с ним соединение. Сервер обычно использует для посылки ответного сообщения то же самое соединение, после чего оно разрывается. Проблема состоит в том, что установка и разрыв соединения в смысле затрачиваемого времени и ресурсов относительно дороги, особенно если сообщения с запросом и ответом невелики.

Разделение приложений по уровням

Один из главных вопросов состоит в том, как разделить клиент и сервер. Обычно четкого различия нет. Например, сервер распределенной базы данных может постоянно выступать клиентом, передающим запросы на различные файловые серверы, отвечающие за реализацию таблиц этой базы данных. В этом случае сервер баз данных сам по себе не делает ничего, кроме обработки запросов.

Однако, рассматривая множество приложений типа клиент-сервер, предназначенных для организации доступа пользователей к базам данных, рекомендуется разделять их на три уровня:

- уровень пользовательского интерфейса;

- уровень обработки;
- уровень данных.

Уровень пользовательского интерфейса содержит все необходимое для непосредственного общения с пользователем, например управление дисплеем. Уровень обработки обычно содержит приложения, а уровень данных — собственно данные, с которыми происходит работа.

Уровень пользовательского интерфейса

Уровень пользовательского интерфейса обычно реализуется на клиентах. Этот уровень содержит программы, посредством которых пользователь может взаимодействовать с приложением. Сложность программ, входящих в пользовательский интерфейс, весьма различна.

Простейший вариант программы пользовательского интерфейса не содержит ничего, кроме символьного (не графического) дисплея. Такие интерфейсы обычно используются при работе с мэйнфреймами. В том случае, когда мэйнфрейм контролирует вес взаимодействия, включая работу с клавиатурой и монитором, мы вряд ли можем говорить о модели клиент-сервер. Однако во многих случаях терминалы пользователей производят некоторую локальную обработку, осуществляя, например, эхо-печать вводимых строк или предоставляя интерфейс форм, в котором можно отредактировать введенные данные до их пересылки на главный компьютер.

В настоящее время даже в среде мэйнфреймов наблюдаются более совершенные пользовательские интерфейсы. Обычно на клиентских машинах имеется как минимум графический дисплей, на котором можно задействовать всплывающие или выпадающие меню и множество управляющих элементов, доступных для мыши или клавиатуры. Типичные примеры таких интерфейсов

— надстройка X-Windows, используемая во многих UNIX-системах, и более ранние интерфейсы, разработанные для персональных компьютеров, работающих под управлением MS-DOS и Apple Macintosh.

Современные пользовательские интерфейсы более функциональны. Они поддерживают совместную работу приложений через единственное графическое окно и в ходе действий пользователя обеспечивают через это окно обмен данными. Например, для удаления файла часто достаточно перенести значок, соответствующий этому файлу, на значок мусорной корзины Аналогичным образом многие текстовые процессоры позволяют пользователю перемешать текст документа в другое место, пользуясь только мышью.

Уровень обработки

Многие приложения модели клиент-сервер построены из трех различных частей: части, которая занимается взаимодействием с пользователем, части, которая отвечает за работу с базой данных или файловой системой, и средней части, реализующей основную функциональность приложения. Эта средняя часть логически располагается на уровне обработки. В противоположность пользовательским интерфейсам или базам данных на уровне обработки трудно выделить общие закономерности. Однако рассмотрим несколько примеров для разъяснения вопросов, связанных с этим уровнем.

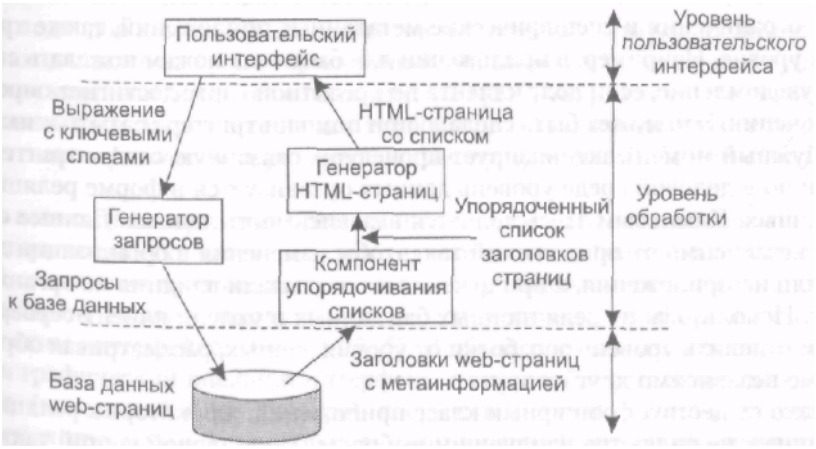


Рис.19. Обобщенная организация трехуровневой поисковой машины для Интернета

В качестве первого примера рассмотрим поисковую машину в Интернете. Если отбросить все оконные украшения, пользовательский интерфейс поисковой машины очень прост: пользователь вводит строку, состоящую из ключевых слов, и получает список заголовков web-страниц. Результат формируется из большой базы просмотренных и проиндексированных web-страниц. Ядром поисковой машины является программа, трансформирующая введенную пользователем строку в один или несколько запросов к базе данных. Затем она помещает результаты запроса в список и преобразует этот список в набор HTML-страниц. В рамках модели клиент-сервер часть, которая отвечает за выборку информации, обычно находится на уровне обработки. Эта структура показана на рис. 19.

В качестве второго примера рассмотрим систему поддержки принятия решений для фондового рынка. Так же как и в поисковой машине, эту систему можно разделить на внешний интерфейс, реализующий работу с пользователем, внутреннюю часть, отвечающую за доступ к базе с финансовой информацией, и промежуточную программу анализа. Анализ финансовых данных может потребовать замысловатых методик и технологий на основе статистических методов и искусственного интеллекта. В некоторых случаях для того, чтобы обеспечить требуемые производительность и время отклика, ядро системы поддержки финансовых решений должно выполняться на высокопроизводительных компьютерах.

Последним примером будет типичный офисный пакет, состоящий из текстового процессора, приложения для работы с электронными таблицами, коммуникационных утилит и т. д. Подобные офисные пакеты обычно поддерживают обобщенный пользовательский интерфейс, возможность создания составных документов и работу с файлами в домашнем каталоге пользователя. В этом случае уровень обработки будет включать в себя относительно большой набор программ, каждая из которых призвана поддерживать какую-то из функций обработки.

Уровень данных

Уровень данных в модели клиент-сервер содержит программы, которые предоставляют данные обрабатывающим их приложениям. Специфическим свойством этого уровня является требование сохранности (persistence). Это означает, что когда приложение не работает, данные должны сохраняться в определенном месте в расчете на дальнейшее использование. В простейшем варианте уровень данных реализуется файловой системой, но чаще для его реализации задействуется полномасштабная база данных. В модели клиент- сервер уровень данных обычно находится на стороне сервера.

Кроме простого хранения информации уровень данных обычно отвечает за поддержание целостности данных для различных приложений. Для базы данных поддержание целостности означает, что метаданные, такие как описания таблиц, ограничения и специфические метаданные приложений, также хранятся на этом уровне. Например, в приложении для банка мы можем пожелать сформировать уведомление, если долг клиента по кредитной карте достигнет определенного значения. Это может быть сделано при помощи триггера базы данных, который в нужный момент активизирует процедуру, связанную с этим триггером.

Обычно в деловой среде уровень данных организуется в форме реляционной базы данных. Ключевым здесь является независимость данных. Данные организуются независимо от приложений так, чтобы изменения в организации данных не влияли на приложения, а приложения не оказывали влияния на организацию данных. Использование реляционных баз данных в модели клиент-сервер помогает отделить уровень обработки от уровня данных, рассматривая обработку и данные независимо друг от друга.

Однако существует обширный класс приложений, для которых реляционные базы данных не являются наилучшим выбором. Характерной чертой таких приложений является работа со сложными типами данных, которые проще модели в понятиях объектов, а не отношений. Примеры таких типов данных — от простых наборов прямоугольников и окружностей до проекта самолета в случае автоматизированного проектирования. Также и мультимедийным системам значительно проще работать с видео- и аудиопотоками, используя специфичные для них операции, чем с моделями этих потоков в виде реляционных таблиц.

В тех случаях, когда операции с данными значительно проще выразить в понятиях работы с объектами, имеет смысл реализовать уровень данных средствами объектно-ориентированных баз данных. Подобные базы данных не только поддерживают организацию сложных данных в форме объектов, но и хранят реализации операций над этими объектами. Таким образом, часть функциональности, приходившейся на уровень обработки, мигрирует в этом случае на уровень данных.

Варианты архитектуры клиент-сервер

Простейшая организация предполагает наличие всего двух типов машин.

- Клиентские машины, на которых имеются программы, реализующие только пользовательский интерфейс или его часть.
- Серверы, реализующие все остальное, то есть уровни обработки и данных.

Проблема подобной организации состоит в том, что на самом деле система не является распределенной: все происходит на сервере, а клиент представляет собой простой терминал.

Многозвенные архитектуры

Один из подходов к организации клиентов и серверов — это распределение программ, находящихся на уровне приложений. В качестве первого шага рассмотрим разделение на два типа машин: на клиенты и на серверы, что приведет к физически двухзвепиой архитектуре (physically two-tiered architecture). Один из возможных вариантов организации — поместить на клиентскую сторону терминальную часть пользовательского интерфейса, как показано на (рис. 20, а), позволив приложению удаленно контролировать представление данных. Альтернативой этому подходу будет передача клиенту всей работы с пользовательским интерфейсом (рис. 20, 6). В обоих случаях отделен от приложения графический внешний интерфейс, связанный с остальной частью приложения (находящейся на сервере) посредством специфичного для данного приложения протокола. В подобной модели внешний интерфейс делает только то, что необходимо для предоставления интерфейса приложения.

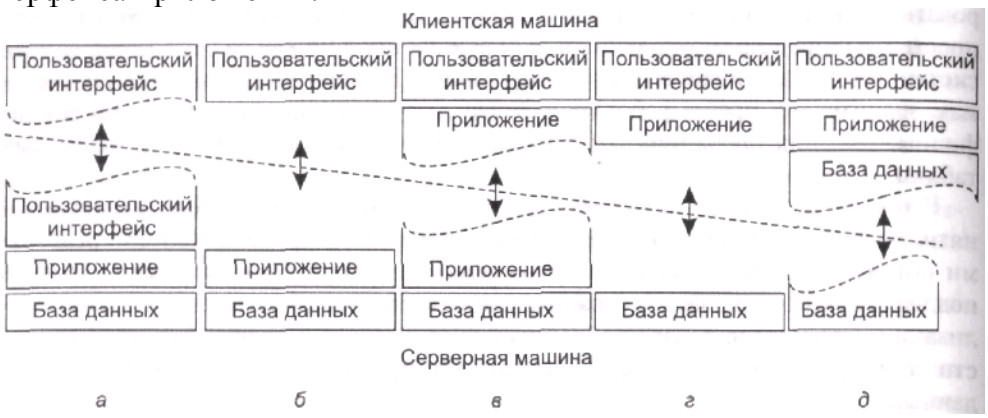
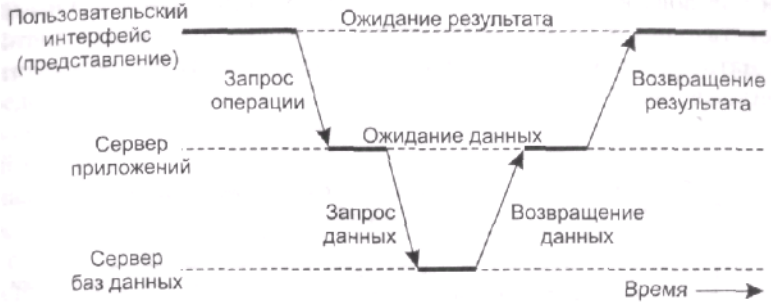


Рис. 20. Альтернативные формы организации архитектуры клиент-сервер

Можно перенести во внешний интерфейс часть приложения, как показано на рис. 20 в. Примером может быть вариант, когда приложение создает форму непосредственно перед ее заполнением. Внешний интерфейс проверяет правильность и полноту заполнения формы и при необходимости взаимодействует с пользователем. Другим примером организации системы по образцу, представленному на рис. 1.20 в, может служить текстовый процессор, в котором базовые функции редактирования осуществляются на стороне клиента с локально кэшируемыми или находящимися в памяти данными, а специальная обработка, такая как проверка орфографии или грамматики, выполняется на стороне сервера.

Во многих системах клиент-сервер популярна организация, представленная на рис. 20, г и д. Эти типы организации применяются и том случае, когда клиентская машина — персональный компьютер или рабочая станция — соединена сетью с распределенной файловой системой или базой данных. Большая часть приложения работает на клиентской машине, а все операции, с файлами или базой данных передаются на сервер. Рисунок 20, д отражает ситуацию, когда часть данных содержится на локальном диске клиента. Так, например, при работе в Интернете клиент может постепенно создать на локальном диске большой кэш наиболее часто посещаемых web- страниц.

Серверу иногда может понадобиться работать в качестве клиента. Такая ситуация, отраженная на рис. 21, приводит к физически трехзвенной архитектуре (physically three-tiered architecture).



В подобной архитектуре программы, составляющие часть уровня обработки, выносятся на отдельный сервер, но дополнительно могут частично находиться и на машинах клиентов и серверов. Типичный пример трехзвенной архитектуры — обработка транзакций. В этом случае отдельный процесс — монитор транзакций — координирует все транзакции, возможно, на нескольких серверах данных.

Рис. 21. Пример сервера, действующего как клиент

Современные варианты архитектуры

Многозвенные архитектуры клиент-сервер являются прямым продолжением разделения приложений на уровни пользовательского интерфейса, компонентов обработки и данных. Различные звенья взаимодействуют в соответствии с логической организацией приложения. Во множестве бизнес-приложений распределенная обработка эквивалентна организации многозвенной архитектуры приложений клиент-сервер. Мы будем называть такой тип распределения вертикальным распределением (vertical distribution). Характеристической особенностью вертикального распределения является то, что оно достигается размещением логически различных компонентов на разных машинах. Это понятие связано с концепцией вертикального разбиения (vertical fragmentation), используемой в распределенных реляционных базах данных, где под этим термином понимается разбиение по столбцам таблиц для их хранения на различных машинах.

Однако вертикальное распределение — это лишь один из возможных способов организации приложений клиент-сервер. В современных архитектурах распределение на клиенты и серверы происходит способом, известным как горизонтальное распределение (horizontal distribution). При таком типе распределения клиент или сервер может содержать физически разделенные части логически однородного модуля, причем работа с каждой из частей может происходить независимо. Это делается для выравнивания загрузки.

В качестве распространенного примера горизонтального распределения рассмотрим web-сервер, реплицированный на несколько машин локальной сети, как показано на рис. 22. На каждом из серверов содержится один и тот же набор web-страниц, и всякий раз, когда одна из web-страниц обновляется, ее копии немедленно рассылаются на все серверы. Сервер, которому будет передан исходящий запрос, выбирается по правилу «карусели» (round-robin). Эта форма горизонтального распределения весьма успешно используется для выравнивания нагрузки на серверы популярных web-сайтов.

Таким же образом могут быть распределены и клиентские части для несложного приложения, предназначенного для коллективной работы, когда можно не иметь сервера вообще. В этом случае говорят об одноранговом распределении (peer-to-peer distribution). Это происходит, например, если пользователь хочет связаться с другим пользователем. Оба они должны запустить одно и то же приложение, чтобы начать сеанс. Третий клиент может общаться с одним из них или обоими, для чего ему нужно запустить то же самое приложение.

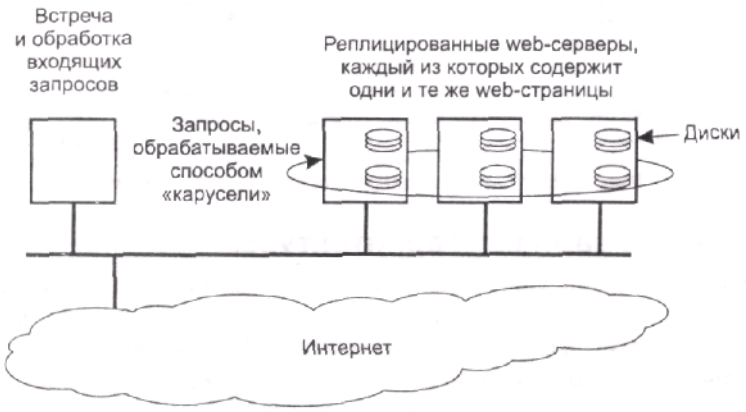


Рис. 22. Пример горизонтального распределения web-службы

2. Методические рекомендации (указания) к практическим занятиям

Практические занятия учебным планом не предусмотрены

3. Методические рекомендации (указания) к лабораторным занятиям

Лабораторно-практические занятия – как обязательный элемент образовательного процесса по данной дисциплине, призван закрепить полученные теоретические знания и обеспечить формирование основных навыков и умений практической работы в области обработки информации, с помощью распределённых систем, а также методов их построения. Они проводятся по мере изучения теоретического материала и выполняются индивидуально каждым студентом.

В ходе лабораторно-практических занятий студент должен приобрести:

- навыки и умения работы в распределенных системах;
- приемы создания структурных единиц рассматриваемых систем, и их компонентов.

Лабораторная работа 1. Распределенные базы данных

В данной лабораторной работе нужно спроектировать распределѐнную базу данных, которая должна быть размещена на двух узлах (серверах баз данных). В качестве предметной области, для которой будет создаваться распределѐнная база данных, можно выбрать ту же предметную область, для которой на IV курсе в рамках курсового проекта создавалась база данных. В данной лабораторной работе нужно создать два варианта распределѐнной базы данных: один – при помощи распределения таблиц по узлам на основе некоторой логики, другой – при помощи оптимального распределения, полученного в результате решения оптимизационной задачи. Каждый вариант разбиения предполагает создание двух баз данных на разных узлах (в данной работе базы данных на разных узлах будут моделироваться при помощи схем).

Для размещения таблиц распределѐнной базы данных следует использовать базы данных students51 и students52. Обе базы данных расположены на сервере students.ami.nstu.ru.

Для переноса курсового проекта, выполненного на IV курсе, из базы данных students необходимо с помощью программы PuTTY зайти на сервер students.ami.nstu.ru под бригад-ным логином (не текущего семестра, а под тем, с помощью которого ранее создавалась база дан-ных) и выполнить описанные далее действия.

Экспортировать ранее созданную базу данных в SQL-файл. Для этого можно воспользоваться двумя способами: выполнить команду следующего вида:

pg_dump -U логин --encoding=utf-8 -f файл.sql -n имя_схемы --no-owner -x students

Например, если курсовой проект находился в схеме pm6503 и бригадный логин также был pm6503, команда будет выглядеть так:

pg_dump -U pm6503 --encoding=utf-8 -f pm6503_dump.sql -n pm6503 --no-owner -x students

зайти в phpPgAdmin под групповым логином (http://ami.nstu.ru/phpPgAdmin, на странице выбрать сервер students.ami.nstu.ru), найти ранее созданную схему в базе students, выбрать еѐ и в правом верхнем углу нажать Экспорт.

Для каждого из вариантов разбиений создать по две пары схем в базах students51 и students52. Для этого можно использовать следующие команды:

new_schema имя_первой_схемы students51

new_schema имя_второй_схемы students52

new_schema имя_третьей_схемы students51

new_schema имя_четвертой_схемы students52

Используя sql-запросы, создающие таблицы, в каждую схему добавить необходимые таблицы. Это же можно сделать, загрузив таблицы из файла, содержащего sql-запросы на создание таблиц, командой вида

psql -U логин -d имя_базы_данных -f имя_sql_файла.sql

Распределѐнные запросы (т.е. те запросы, которые одновременно обращаются к нескольким базам данных) из базы students51 реализуются следующим образом (с использованием библиотеки dblink):

dblink(text connstr, text sql),

где sql – sql-выражение, запускаемое на выполнение на удалѐнном узле (например, "select * from customer"), a connstr – строка соединения вида

"dbname=<имя_базы> user=<логин> password=<пароль>"

Примеры использования dblink:

```
select *
from    ttt.t1,
public.dblink
('dbname=students52 user=login password=pswd',
'select * from t2'
) as t2 (t21 text,
t22 text
)
where   t1.t12 = t2.t21;
select *
from    public.dblink
('dbname=students52 user=login password=pswd',
'select id_tovar, name, id_type_tovar from имя_схемы.tovar' ) as t1 (id_tovar varchar,
name varchar,
id_type_tovar varchar
);
```

Для корректного измерения времени выполнения каждый запрос нужно выполнить несколько раз и взять среднее время выполнения.

Использование представлений

Представление (view) – объект данных, который не содержит никаких данных его владельца. Это тип таблицы, чьѐ содержание выбирается из других таблиц с помощью выполнения запроса. Поскольку значения в этих таблицах меняются, их значения автоматически будут показываться представлением.

Представление – это фактически запрос, который выполняется всякий раз, когда происходит обращение к представлению. Результат запроса при этом в каждый момент становится содержанием представления.

Для повышения производительности распределѐнных СУБД зачастую используют механизм репликации данных. Для учебных целей этот механизм можно реализовать с помощью кэширования удалѐнных таблиц СУБД в локальные представления. Таким образом, можно кэшировать удалѐнную таблицу в локальном представлении и обращаться не к удалѐнной таблице, а к еѐ локальной копии, не используя dblink.

Представление можно создать следующим образом:

```
create view имя_схемы.tovar as
select *
from    public.dblink
('dbname=students52 user=login password=password',
'select id_tovar, name, id_type_tovar from имя_схемы.tovar' ) as t1 (id_tovar varchar,
name varchar,
id_type_tovar varchar);
```

В дальнейшем в качестве таблицы в запросах следует использовать имя_схемы.tovar.

Задание

1. На основе базы данных, разработанной в рамках курса «Базы данных», создать распределѐн-ную базу (база данных должна состоять не менее, чем из 8 таблиц) в двух вариантах: Разделить таблицы БД на две группы по какому-либо смысловому признаку. Например, если имеется база данных «Спортивный клуб», то в первую группу можно отнести таблицы, используемые для административной работы: список спортсменов, список тренеров, список баз и т.д., а во вторую группу – таблицы для соревнований: график соревнований, участие спортсменов в соревнованиях и т.д.

Провести оптимальное размещение таблиц базы по двум узлам. Задачу линейного программирования можно решать любым подходящим алгоритмом с использованием любого существующего ПО. Оптимальное размещение следует рассчитывать, исходя из того, что суммарный объѐм узлов ( ) должен быть примерно равен , где – сум-марный объѐм всех таблиц (это требование нужно, чтобы исключить ситуацию, когда решением оптимизационной задачи является размещение всех таблиц на одном узле).

2. Реализовать генератор данных больших объѐмов. С помощью этого генератора заполнить оба варианта разбиения баз данных большим числом данных. В базе данных должны быть таблицы, имеющие несколько сотен записей, и таблицы, имеющие несколько тысяч записей. При этом должна сохраняться целостность как локальных баз данных, так и глобальной распределѐнной базы.

3. Оценить эффективность каждого варианта по набору sql-запросов, имевшемуся в задании к ранее выполненному курсовому проекту. Посчитать время выполнения каждого из запросов к базе данных и суммарное время работы по двум базам для каждого разбиения и сделать выводы.

Лабораторная работа 2. Механизм репликаций.

К числу преимуществ, которые обеспечивает репликация, относится повышение скорости и надежности. Чтобы обеспечить надежность, можно установить две системы и при возникновении проблем с головным сервером переключаться на резервную копию. Для увеличения скорости можно перенаправлять те запросы, которые не обновляют данные, на сервер с копиями. Разумеется, это даст эффект лишь в том случае, если запросы, не обновляющие данные, преобладают, но, как правило, чаще всего так и бывает.

MySQL, начиная с версии 3.23.15, поддерживает односторонний внутренний механизм репликации. Один сервер действует как головной, а другие - как подчиненные. Обратите внимание: один сервер может играть роль головного в одной паре и подчиненного - в другой. Головной сервер содержит двоичный журнал обновлений и индексный файл двоичных журналов для протоколирования ротации двоичных журналов. Подчиненный сервер при соединении уведомляет головной о том, в каком состоянии он находится, начиная от последнего обновления, которое было успешно опубликовано на подчиненный сервер. После этого подчиненный сервер принимает обновления, а затем блокируется и ждет, пока головной сервер не сообщит о новых обновлениях.

Обратите внимание: при реплицировании базы данных все обновления этой базы данных должны производиться через головной сервер!

Еще одно преимущество использования механизма репликации заключается в том, что можно иметь "живую" резервную копию системы, выполняя резервное копирование не на головном, а на подчиненном сервере.

Как реализована репликация: обзор

Репликация в MySQL основывается на том, что все изменения базы данных (обновления, удаления и т.д.) протоколируются в двоичном журнале на сервере, а подчиненный сервер читает сохраненные запросы из двоичного журнала головного сервера и выполняет эти запросы на своей копии данных.

Очень важно понимать, что двоичный журнал - это просто запись, начатая с фиксированного момента времени (с момента, когда вы включаете ведение записей в двоичном журнале). При установке каждого из подчиненных серверов нужно будет скопировать с головного сервера все данные, существовавшие на нем к моменту начала ведения записей в двоичном журнале. Если подчиненный сервер будет запущен с данными, не соответствующими тем, которые содержались на головном сервере к моменту запуска двоичного журнала, на подчиненном сервере может произойти сбой.

Начиная с версии 4.0.0 для записи данных на подчиненный сервер можно использовать команду LOAD DATA FROM MASTER. Обратите внимание: подчиненные серверы версии 4.0.0 не могут связываться с головными серверами версии 3.23, но подчиненные серверы версии 4.0.1 и

более поздних - могут. Подчиненный сервер версии 3.23 не может общаться с головным сервером версии 4.0.

Учтите, что команда LOAD DATA FROM MASTER в настоящее время работает только если все таблицы на головном сервере имеют тип MyISAM, и для них будет установлена глобальная блокировка чтения, чтобы не допустить никаких записей во время передачи таблиц от головного сервера к подчиненному. Данное ограничение носит временный характер. Оно обусловлено тем, что мы еще не реализовали горячее резервное копирование таблиц без блокировок. Это ограничение мы снимем для следующих ветвей версии 4.0 - как только будет реализовано горячее резервное копирование, которое позволит команде LOAD DATA FROM MASTER работать без блокирования обновлений на головном сервере.

Из-за вышеупомянутого ограничения рекомендуется использовать команду LOAD DATA FROM MASTER только в тех случаях, если набор данных на головном сервере относительно невелик или если для головного сервера допустима длительная блокировка чтения. Скорость выполнения команды LOAD DATA FROM MASTER для разных систем может быть различной, поэтому для грубой оценки времени выполнения команды можно считать, что для передачи 1 Мб данных требуется 1 секунда. Это приблизительно соответствует случаю, когда и головной, и подчиненный серверы эквивалентны Pentium с тактовой частотой 700 МГц и связаны сетью с пропускной способностью 100 Мбит/с, а размер индексного файла равен примерно половине размера файла данных. Разумеется, такая прикидка дает лишь грубую приближенную оценку и в случае каждой конкретной системы потребуются свои допущения.

После того как подчиненный сервер будут правильно сконфигурирован и запущен, он должен легко соединиться с головным сервером и ожидать обработки обновлений. Если головной сервер завершит работу или подчиненный сервер потеряет связь с головным, подчиненный сервер будет пытаться установить соединение каждый раз по истечении интервала времени, указанного в опции master-connect-retry (в секундах) до тех пор, пока не установится подсоединение и не продолжится прослушивание обновлений.

Каждый подчиненный сервер отслеживает события с момента разрыва. Головной сервер не имеет никакой информации о том, сколько существует подчиненных серверов, и какие из них обновлены последними данными в любой момент времени.

В следующем разделе процесс установки головного/подчиненного серверов рассматривается более подробно.

Как настроить репликацию

Здесь кратко описано как настроить полную репликацию вашего MySQL-сервера. Предполагается, что реплицироваться будут все базы данных и репликация ранее не настраивалась. Для того чтобы выполнить указанные здесь действия, вам придется на короткое время остановить головной сервер.

Это самый простой способ установки подчиненного сервера, однако он не единственный. Например, если уже имеется образ головного сервера, на головном сервере уже установлен ID сервера и производятся записи в журнал, подчиненный сервер можно установить, не останавливая головной сервер и даже не устанавливая блокировки.

1. Удостоверьтесь, что на головном и подчиненном(ых) серверах установлена свежая версия MySQL. Используйте версию 3.23.29 или выше. В предыдущих релизах применялся другой формат двоичного журнала и содержались ошибки, которые были исправлены в более новых релизах. Большая просьба: пожалуйста, не посылайте сообщения об ошибках, не проверив, присутствует ли эта ошибка в последнем релизе.
2. Установите на головном сервере отдельного пользователя для репликации с привилегией FILE (в версиях MySQL ниже 4.0.2) или REPLICATION SLAVE в более новых версиях MySQL. У этого пользователя должно быть также разрешение подсоединяться со всех подчиненных серверов. Если пользователь будет выполнять только репликацию (рекомендуется), то ему не нужно предоставлять какие-либо дополнительные привилегии. Например, чтобы создать пользователя с именем repl, который может иметь доступ к головному серверу с любого хоста, можно использовать такую команду:

mysql> GRANT FILE ON *.* TO repl@"%" IDENTIFIED BY '<password>';

3. Завершите работу MySQL на головном сервере.

mysqladmin -u root -p<password> shutdown

4. Создайте образ всех данных на головном сервере. Легче всего сделать это (на Unix), создав при помощи tar архив всей своей директории данных. Точное местоположение директории данных зависит от вашей инсталляции.

tar -cvf /tmp/mysql-snapshot.tar /path/to/data-dir

Пользователи Windows для создания архива каталога данных могут использовать WinZIP или другую подобную программу.

5. В my.cnf на головном сервере добавьте записи к разделу [mysqld] записи log-bin и server-id=уникальный номер к разделу [mysqld] и перезапустите сервер. Очень важно, чтобы ID подчиненного сервера отличался от ID головного сервера. Можно считать, что server-id играет роль IP-адреса - он уникально идентифицирует сервер среди участников репликации.

[mysqld]
log-bin
server-id=1

6. Перезапустите MySQL на головном сервере.

7. Добавьте в my.cnf на подчиненном сервере(ах) следующий фрагмент:

master-host=<имя хоста головного сервера>
master-user=<имя пользователя репликации >
master-password=<пароль пользователя репликации >
master-port=<порт TCP/IP для головного сервера>
server-id=<некоторое уникальное число между 2 и 2^32-1>

заменяя значения в <> значениями, соответствующими вашей системе. Значения server-id должны быть различными на каждом сервере, участвующем в репликации. Если значение server-id не определено, оно будет установлено в 1, если также не определено значение master-host, оно будет установлено в 2. Обратите внимание, что если значение server-idопущено, то головной сервер будет отказывать в соединении всем подчиненным серверам, а подчиненный сервер - отказывать в соединении головному серверу. Таким образом, опускать установку значения server-id можно лишь в случае резервного копирования с использованием двоичного журнала.

8. Скопируйте данные снимка в директорию данных на подчиненном сервере (ах). Удостоверьтесь в правильности привилегий для файлов и каталогов. Пользователь, от имени которого запускается MySQL, должен иметь возможность читать и записывать данные в них так же, как и на головном сервере.

9. Перезапустите подчиненный(ые) сервер(ы).

После выполнения указанных действий подчиненный(ые) сервер(ы) должен(ы) подсоединиться к головному серверу и подгонять свои данные под любые изменения, произошедшие на головном сервере после принятия образа.

Если не установлен идентификатор server-id для подчиненного сервера, в журнальный файл регистрации ошибок будет внесена следующая ошибка:

Warning: one should set server_id to a non-0 value if master_host is set.

The server will not act as a slave.

Если не установлен идентификатор головного сервера, подчиненные серверы не смогут подключиться к головному серверу.

Если подчиненный сервер по какой-либо причине не может выполнять репликацию, соответствующие сообщения об ошибках можно найти в журнале регистрации ошибок на подчиненном сервере.

После того как подчиненный сервер начнет выполнять репликацию, в той же директории, где находится журнал регистрации ошибок, появится файл `master.info'. Файл `master.info'используется подчиненным сервером для отслеживания того, какие записи двоичных журналов головного сервера обработаны. Не удаляйте и не редактируйте этот файл, если не

уверены в том, что это необходимо. Даже если такая уверенность есть, все равно лучше использовать команду CHANGE MASTER TO.

Возможности репликации и известные проблемы

Ниже приводится список поддерживаемых и не поддерживаемых при репликации функций:

- Реплицирование будет выполнено правильно при использовании значений AUTO_INCREMENT, LAST_INSERT_ID() и TIMESTAMP.
- Если в обновлениях присутствует функция RAND(), реплицирование будет выполнено некорректно. При реплицировании обновлений с функцией RAND() применяйте RAND(some_non_rand_expr). В качестве аргумента (some_non_rand_expr - некоторое не случайное выражение) для функции RAND() можно, например, использовать функцию UNIX_TIMESTAMP().
- На головном и подчиненном серверах следует использовать одинаковый набор символов (--default-character-set). В противном случае могут возникать ошибки дублирующихся ключей на подчиненном сервере, поскольку ключ, который считается уникальным на головном сервере, может не быть таковым при использовании другого набора символов.
- В MySQL 3.23 команда LOAD DATA INFILE будет выполнена корректно, если файл во время выполнения обновления будет находиться на головном сервере. Команда LOAD LOCAL DATA INFILE будет проигнорирована. В MySQL 4.0 это ограничение не присутствует - все разновидности команды LOAD DATA INFILE реплицируются правильно.
- Запросы на обновление, в которых используются пользовательские переменные, являются не безопасными для репликации (пока).
- Команды FLUSH не записываются в двоичный журнал и поэтому не копируются на подчиненный сервер. Проблем при этом не возникает, поскольку команды FLUSH ничего не изменяют. Однако это означает, что при непосредственном, без использования оператора GRANT, обновлении таблиц привилегий MySQL и при последующем реплицировании базы данных привилегий mysql нужно выполнить команду FLUSH PRIVILEGES на подчиненных серверах, чтобы новые привилегии вступили в силу.
- Временные таблицы, начиная с версии 3.23.29, реплицируются корректно, за исключением случая, когда при прекращении работы подчиненного сервера (не только потока подчиненного сервера) некоторые временные таблицы остаются открытыми и используются в последующих обновлениях. Для решения этой проблемы перед прекращением работы подчиненного сервера выполните команду SLAVE STOP, проверьте, чтобы переменная Slave_open_temp_tables содержала значение 0, затем выполните mysqladmin shutdown. Если значение переменной Slave_open_temp_tables не 0, перезапустите поток подчиненного сервера при помощи команды SLAVE START и проверьте, не улучшилась ли ситуация теперь. Эта проблема будет решаться более изящно, но придется подождать MySQL 4.0. В более ранних версиях при использовании временных таблиц репликации не выполняются должным образом - в таких случаях мы рекомендуем либо обновить версию MySQL, либо перед выполнением запросов, использующих временные таблицы, выполнить команду SET SQL_LOG_BIN=0 на своих клиентах.
- MySQL поддерживает лишь один головной и много подчиненных серверов. В 4.x будет добавлен алгоритм голосования, обеспечивающий автоматическое изменение головного сервера, если что-либо будет выполняться неправильно при текущем головном сервере. Будут также введены процессы 'агента', которые помогут выполнять распределение нагрузки путем посылки запросов на выборки различным подчиненным серверам.
- Начиная с версии 3.23.26 стало безопасно соединять серверы циклическими соединениями головной-подчиненный с включенной опцией log-slave-updates. Однако обратите внимание: при таком способе установки многие запросы не будут выполняться корректно, если только в коде вашего клиента не предусмотрена обработка потенциальных проблем, которые могут случаться при обновлениях, происходящих в различной последовательности на различных серверах. Это означает, что если вы сделаете установку следующим образом:
- A - > B > - C - > A

то такая установка будет работать только в том случае, если выполняются непротиворечивые обновления между таблицами. Другими словами, при вставке данных на серверах A и Снельзя вставлять на сервере A строку, которая может иметь ключ, противоречащий строке, вставляемой на сервере C. Также нельзя обновлять одинаковые строки на двух серверах, если имеет значение порядок обновлений. Обратите внимание: в версии 3.23.26 изменился формат журнала. Таким образом, если версия подчиненного сервера меньше 3.23.26, сервер не сможет считывать записи из журнала.

• Если запрос на подчиненном сервере вызывает ошибку, поток подчиненного сервера завершится, и в файле `.err' появится соответствующее сообщение. После этого нужно будет вручную установить соединение с подчиненным сервером, исправить причину ошибки (например обращение к несуществующей таблице) и затем выполнять SQL-команду SLAVE START(доступна в версии 3.23.16). При использовании версии 3.23.15 потребуется перезапустить сервер.

• Если соединение с головным сервером прервется, подчиненный сервер попытается сразу же восстановить его, и затем в случае неудачи будет повторять попытки через установленное в опции master-connect-retry количество секунд (по умолчанию 60). По этой причине безопасно выключить головной сервер и после этого перезапустить его через некоторое время. Подчиненный сервер будет также разрешать проблемы, возникающие при аварийных отключениях электричества в узлах сети.

• Завершение работы подчиненного сервера (корректное) также является безопасным, поскольку при этом отслеживаются события начиная от момента остановки сервера. Но в случае некорректного отключения сервера могут возникать проблемы, особенно, если дисковый кэш не был синхронизирован перед ``смертью'' системы. Для того чтобы значительно повысить эффективность своей системы обеспечения отказоустойчивости, целесообразно приобрести хороший UPS (источник бесперебойного питания).

• Если головной сервер слушает нестандартный порт, это нужно будет указать также в параметре master-port в файле `my.cnf'.

• В версии 3.23.15 все таблицы и базы данных могут быть реплицированы. Начиная с версии 3.23.16 появилась возможность ограничить репликацию набором баз данных при помощи директив replicate-do-db в файле `my.cnf'; можно также исключить набор баз данных из репликации при помощи директив replicate-ignore-db. Обратите внимание,: в версиях MySQL до 3.23.23, имелась ошибка, из-за которой команда LOAD DATA INFILE выполнялась некорректно, если она применялась к базе данных, исключенной из репликации.

• Начиная с версии 3.23.16 команда SET SQL_LOG_BIN = 0 будет выключать ведение записей о репликации в журналах (двоичных) на головном сервере, а команда SET SQL_LOG_BIN = 1 - включать такое ведение записей. Для выполнения этих команд нужно иметь привилегию SUPER (в MySQL 4.0.2 и выше) или PROCESS (в более ранних версиях MySQL).

• Начиная с версии 3.23.19 можно убрать мусор, оставшийся после неоконченной репликации (если ее выполнение пошло не должным образом), и начать все сначала, используя команды FLUSH MASTER и FLUSH SLAVE. В версии 3.23.26 эти команды переименованы в RESET MASTER и RESET SLAVE соответственно - чтобы сделать понятным их назначение. Тем не менее, старые варианты FLUSH все еще работают - для обеспечения совместимости.

• Начиная с версии 3.23.23 можно заменять головные серверы и корректировать точку положения в журнале репликации при помощи команды CHANGE MASTER TO.

• Начиная с версии 3.23.23 можно при помощи опции binlog-ignore-db уведомлять головной сервер о том, что обновления в некоторых базах данных не должны отражаться в двоичном журнале.

• Начиная с версии 3.23.26, можно использовать опцию replicate-rewrite-db для уведомления подчиненного сервера о том, что он должен применить обновления базы данных на головном сервере к базе данных с другим именем на подчиненном сервере.

• Начиная с версии 3.23.28 можно использовать команду PURGE MASTER LOGS TO 'имя-журнала', чтобы избавиться от старых журналов без завершения работы подчиненного сервера.

• Из-за того, что по своей природе таблицы MyISAM являются нетранзакционными, может случиться так, что запрос обновит таблицу только частично и возвратит код ошибки. Это может произойти, например, при вставке нескольких строк, одна из которых нарушает ограничение ключа, или в случае, когда длинный запрос обновления ``убивается'' после обновления некоторых строк. Если такое случится на головном сервере, поток подчиненного сервера завершит работу и будет ждать, пока администратор базы данных не примет решение о том, что делать в этом случае (если только код ошибки не является легитимным и в результате выполнения запроса не будет сгенерирована ошибка с тем же кодом). Если такой способ проверки правильности кода ошибки нежелателен, начиная с версии 3.23.47, некоторые (или все) ошибки могут быть замаскированы при помощи опции slave-skip-errors.

• Отдельные таблицы могут исключаться из репликации при помощи опции replicate-do-table/replicate-ignore-tab или опции replicate-wild-do-table/replicate-wild-ignore-table. Однако в настоящее время наличие определенных конструктивных неточностей в некоторых довольно редких случаях может приводить к неожиданным результатам. Протокол репликации явно не уведомляет подчиненный сервер о том, какие таблицы должны быть изменены запросом, поэтому подчиненному серверу требуется анализировать запрос, чтобы узнать это. Чтобы избежать лишнего синтаксического анализа, для которого требуется прерывать выполнение запросов, исключение таблицы в настоящее время реализуется путем посылки запроса к стандартному анализатору MySQL для упрощенного синтаксического анализа. Если анализатор обнаружит, что таблица должна игнорироваться, выполнение запроса будет остановлено и выдано сообщение об успехе. Этот подход несколько неэффективен, при его применении чаще возникают ошибки и, кроме того, имеются две известные ошибки в версии 3.23.49. Первая может возникнуть из-за того, что поскольку анализатор автоматически открывает таблицу при анализе некоторых запросов, игнорируемая таблица должна существовать на подчиненном сервере. Другая ошибка заключается в том, что при частичном обновлении игнорируемой таблицы поток подчиненного сервера не заметит, что таблица должна игнорироваться, и приостановит процесс репликации. Несмотря на то что вышеупомянутые ошибки концептуально очень просто исправить, для этого придется изменить достаточно много кода, что поставит под угрозу состояние стабильности ветви 3.23. Если описанные случаи непосредственно имеют отношение к вашему приложению (а это довольно редкий случай) - используйте опцию slave-skip-errors, чтобы дать указание серверу продолжать репликации, игнорируя эти ошибки.

Лабораторная работа 3. Распределенные запросы.

Создать запросы

Первый вариант. БД поликлиники

1. Выдать список всех пациентов и номера их полисов.
2. Выдать всю информацию по медицинской карте заданного пациента
3. Рассчитать сколько посещений обслужил каждый врач.
4. Какие назначения были выполнены заданным врачом.
5. Какие назначения были выполнены пациентом. Задать фио пациента и дату посещения.
6. Сколько посещений выполнил заданный пациент в марте 2017 года.
7. Какие назначения приписывались больным гриппом
8. Сколько пациентов больных гриппом посетили поликлинику
9. Выбрать пациента с максимальным количеством посещений.
10. Выбрать врача с минимальным количеством посещений.

Второй вариант. БД Строительная компания

1. Выдать всех заказчиков Строительной компании и их телефоны.
2. Выдать всех исполнителей, которые вели работы в мае 2017.
3. Для заказа рассчитать стоимость работ.
4. Рассчитать стоимость всех заказов
5. Выдать всех исполнителей по заданному проекту.
6. Рассчитать стоимость всех проектов заданного заказчика.

7. Для заказа выдать все работы и исполнителей этих работ.
8. Выявить исполнителей, которые не работали в летние месяцы.
9. По исполнителям рассчитать количество работ.
10. Выявить заказ с максимальной стоимостью.

Третий вариант. БД склад

1. Выдать остаток товаров на складе по количеству и стоимости.
2. Выдать отчет о поступлении товаров в сентябре 2017.
3. Выдать отчет о продаже товаров.
4. По заданному товару сформировать справку о движении.
5. По заданному поставщику рассчитать стоимость поставки.
6. Выбрать покупателя с максимальной суммой .
7. Выбрать товар, который плохо продается.
8. Выдать прайс лист
9. Какие товары поставляет заданный поставщик.
10. Какие товары были куплены заданным покупателем летом.

Лабораторная работа 5. Интерфейс доступа к данным

Основные сведения

СУБД MS Access можно использовать как замкнутую систему, однако, для реализации интегрированных информационных систем, использующих для хранения данных файлы различных форматов – других баз данных, электронных таблиц или текстовых файлов, в MS Access 2000 поддерживается импорт, экспорт и связывание внешних данных. Для этого используются либо встроенные драйверы, либо драйверы ODBC (Open DataBase Connectivity).

Стандарт ODBC , также как и интерфейсы (DAO Data Access Objects), Remote Data Objects (RDO), ActiveX Data Objects (ADO) и Object Linking andEmbedding DataBase (OLE DB) относится к наиболее популярным интерфейсам, входящим в семейство общего интерфейса доступа к данным (API - Application Programming Interface), позволяющего иметь дело с несколькими системами баз данных и существенно упрощающего процесс разработки приложения.

Интерфейс прикладного программирования ODBC API предоставляет общие методы доступа на основе языка баз данных SQL как к реляционным, так и к нереляционным источникам данных.

Интерфейс ODBC API реализован как набор расслоенных DLL-функций для Windows. Динамическая библиотека ODBC.DLL - это основная библиотека управления драйверами ODBC, которая содержит функции вызовов специализированных драйверов для разных поддерживаемых системой баз данных. Каждый драйвер совместим со своим уровнем CLI (Call Level Interfase) и относится к одной из двух категорий: одноуровневые или многоуровневые драйверы.

Одноуровневые драйверы предназначены для использования при работе с теми источниками данных, которые не могут быть прямо обработаны с использованием ANSI SQL. Обычно это локальные базы данных на персональных компьютерах, такие как dBase, Paradox, FoxPro и Excel. Драйверы, соответствующие этим базам данных, производят компиляцию ANSI SQL в наборы инструкций более низкого уровня, которые непосредственно обрабатывают составляющие базу данных файлы.

Технология ODBC разрабатывалась как общий, независимый от источников данных, способ доступа к данным. Применение технологии должно было также обеспечить переносимость приложений в среду различных баз данных без необходимости переработки самих приложений. В этом смысле технологияODBC уже стала промышленным стандартом, ее поддерживают практически все производители СУБД и средств разработки.

Однако универсальность стоит дорого. Если при разработке приложений одним из основных критериев является переносимость на различные СУБД, то использование ODBC является оправданным. Для увеличения производительности и эффективности приложения активно применяют специфические для данной СУБД расширения языка SQL, используют хранимые на сервере про-

цедуры и функции. В этом случае теряется роль ODBC как общего метода доступа к данным. Тем более, что для разных СУБД драйверы ODBC поддерживают разные уровни совместимости. Поэтому многие производители средств разработки, помимо поддержки ODBC, поставляют "прямые" драйверы к основным СУБД.

Для того, чтобы выбрать тип доступа приложения к данным из других баз, следует сравнить импорт и связывание.

Импорт предпочтительнее в следующих случаях:

- необходимый файл сравнительно невелик и пользователи исходной базы данных редко меняют его содержимое;
- параллельно с нашим приложением с данными исходного файла не будут работать приложения других пользователей;
- старое приложение меняется на новое и прежний формат данных больше использоваться не будет;
- необходимо обеспечить максимальную эффективность работы с данными другой СУБД, для этого Access должен работать со своей копией данных и в своем формате.

Связывание эффективнее, когда:

- необходимый файл превышает максимально возможный размер базы данных Access (2 Гбайт);
- данные часто меняются пользователями исходной базы данных;
- данные будут использоваться в режиме коллективного доступа;
- разрабатываемое приложение будет распространяться среди различных пользователей, при этом пользовательский интерфейс возможно будет модифицироваться. В этом случае отделение приложения от данных позволяет изменять приложение, не затрагивая пользовательские данные.

Дальнейшим развитием концепции ODBC является система объектов данных ADO (ActiveX Data Objects) корпорации Microsoft, являющаяся «универсальным интерфейсом» для баз данных, независимо от того поддерживают они ODBC или нет.

Задание

1. Познакомиться с концепцией ODBC и технологией импорта, экспорта и связывания внешних данных.
2. Выбрать таблицу в приложении Access (mdb-файл), которая будет использоваться с импортированными данными.
3. Определить наличие драйверов подключения к базам данных, в частности, драйвера Microsoft SQL Server.
4. В соответствии с технологией работы импортировать SQL -таблицу в приложение Access.
5. В соответствии с технологией работы связать SQL-таблицу с приложением Access.
6. Создать форму, демонстрирующую использование связанных таблиц и возможность редактирования исходной SQL -таблицы.
7. Создать приложение «только для исполнения» (mde-файл)

Лабораторная работа 6. Технологии ASP и ADO

Технологии BDE и ODBC существуют уже довольно давно и постепенно утрачивают свою популярность. Технология BDE более не развивается фирмой производителем, хотя все еще поддерживается. Технология ODBC все еще признается в качестве отраслевого стандарта доступа к базам данных, однако также не развивается, поэтому разработчики программного обеспечения все чаще обращают свое внимание на современные и, возможно, более эффективные технологии.

С конца 90-ых годов и до настоящего времени фирма Microsoft пытается создать универсальную платформу доступа к разнородным хранилищам данных для семейства своих операционных систем (Windows). Эта платформа получила название MDAC, что означает Microsoft Data Access Components (Компоненты Microsoft для доступа к данным). Платформа MDAC представля-

ет собой набор взаимосвязанных технологий доступа к данным, обеспечивающих разработчиков программного обеспечения удобным и единообразным механизмом разработки приложений для работы с данными различных типов. Источниками данных в данном случае могут являться не только SQL-сервера Баз Данных, но и иные хранилища структурированных и частично структурированных данных (БД на базе иерархических моделей, службы каталогов, хранилища документов и др.).

За более, чем десятилетнюю историю своего существования, состав и содержание платформы MDAC неоднократно изменялись. На сегодняшний день официально распространяется версия MDAC 2.8 SP 1. В составе операционной системы Windows Vista платформа MDAC получила название Windows DAC.

Архитектура платформы MDAC имеет уже знакомую нам трехуровневую структуру. На верхнем уровне располагаются компоненты программного интерфейса, на среднем – компоненты, отвечающие за трансляцию запросов источникам данных и возвращение данных в приложение, на нижнем – компоненты, непосредственно взаимодействующие с БД или иным источником данных.

На сегодняшний день основными компонентами MDAC являются:

- Компоненты ODBC. Данная технология все еще остается широко используемой и востребованной и будет еще довольно долгое время поддерживаться производителем. Тем не менее, специалисты Microsoft предупреждают, что ее дальнейшего развития не будет и предлагают использовать ее только в тех случаях, когда информационная система не имеет иных источников данных, кроме реляционных СУБД.
- Прикладной программный интерфейс OLE-DB (Object Linking and Embedding). Этот программный интерфейс был разработан как замена прикладному программному интерфейсу ODBC. Оба интерфейса предполагают возможность универсального доступа к данным из программных приложений. Основными отличиями интерфейса OLE DB от ODBC являются:
  - Технология OLE DB изначально разрабатывалась как средство доступа к хранилищам данных (data stores) самой разной природы, а не только к реляционным базам данных. Специалисты Microsoft называют в качестве возможных хранилищ данных любые информационные хранилища, которые могут быть представлены в виде набора строк и столбцов – от текстового файла или электронной таблицы – до промышленной СУБД с сотнями таблиц и миллионами записей.
  - Интерфейсы OLE DB описаны не на одном из языков программирования, а с использованием специальных языков определения интерфейсов модели COM, что обеспечило им большую переносимость и универсальность.
  - Компоненты OLE DB не только обеспечивают пользователям доступ к данным, но и предлагают дополнительные функциональные возможности по их обработке: сортировку или фильтрацию записей, построение иерархических моделей данных, проведение многомерного анализа (доступ к сервисам OLAP).

В терминах технологии OLE DB приложения, которым необходим доступ к данным, носят названия потребителей (consumers), а компоненты, которые реализуют интерфейс OLE DB и обеспечивают доступ к источникам данных, – поставщиков (providers). Так же как и драйвера ODBC, компоненты-поставщики OLE DB могут разрабатываться сторонними производителями для доступа к своим СУБД или иным хранилищам данных. В состав компонентов MDAC входят поставщики данных OLE DB для таких СУБД, как Microsoft SQL Server и Oracle. Кроме того, сюда же входит поставщик OLE DB для драйверов ODBC, позволяющий при необходимости интегрировать две указанных технологии.

- Компоненты ADO (ActiveX Data Objects) – представляют собой высокоуровневый программный интерфейс (объектную модель) доступа к данным посредством технологии OLE DB. Разработчики программного обеспечения, в принципе, могут обойтись и без этого дополнительного уровня между своим приложением и источником данных и использовать интерфейс OLE DB напрямую. Это положительно скажется на производительности программного приложения, однако потребует от программиста гораздо более детального описания механизма взаимодействия между его программой и поставщиком данных. Использование компонентов ADO позволяет раз-

работчику в меньшей степени задумываться о том, как именно устроен источник данных, и каким образом с ним нужно взаимодействовать, но сосредоточиться непосредственно на вопросах отображения и обработки данных. Компоненты ADO незаменимы при разработке Web-приложений, а также приложений, написанных на интерпретируемых языках программирования (Visual Basic, Microsoft JScript и др.). Как и интерфейс OLE DB, объектная модель ADO является независимой от языка программирования (language neutral).

- Технология ADO.NET. Является развитием классической модели ADO, в которую были специально добавлены дополнительные возможности, отвечающие принципам платформы .NET. Особый акцент в данном случае был сделан на возможность разработки распределенных, многозвенных приложений с портфельным способом обработки данных.

На рисунке показана архитектура платформы MDAC.

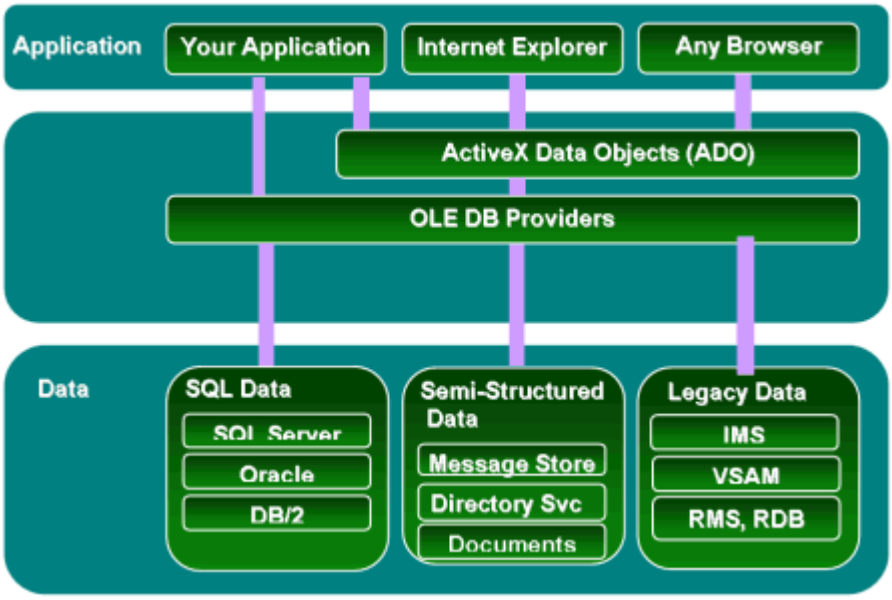


Рис. Архитектура платформы MDAC

Другие производители программного обеспечения также предлагают программистам собственные или совместимые с платформой MDAC технологии доступа к данным. Уже почти 5 лет компания Borland предлагает следующую концепцию разработки БД-приложений в своих средах программирования (Borland Delphi, Borland C++ Builder, Borland Developer Studio):

- Для связи с локальными базами данных устаревших форматов (Paradox, dBASE) рекомендуется использовать технологию BDE.
- Для сравнительно небольших проектов, в которых в качестве источников данных выступают, например, таблицы Excel или БД Access, рекомендуется использовать технологию ADO. Для этого в библиотеку компонентов Borland включен целый ряд невизуальных компонентов, основными из которых являются:
  - TADOConnection – для установки и поддержания соединения с источником данных;
  - TADOTable – аналог компонента TTable, использующийся для работы с набором данных одной таблицы.
  - TADOQuery – аналог компонента TQuery, использующийся для работы с набором данных некоторого запроса.
  - TADOStoredProc – аналог компонента TStoredProc, использующийся для работы с хранимыми процедурами.

В Borland C++ Builder 6 указанные компоненты располагаются на вкладке ADO палитры компонентов. В поздних версиях RAD-продуктов Borland эта вкладка была переименована в dbGo (из-за запрета фирмы Microsoft использовать сокращение ADO другими производителями).

- Для разработки БД-приложений с использованием СУБД Interbase (которая также относится к продуктам Borland) рекомендуется использовать специальный набор компонентов InterbaseExpress (IBX).

- Для разработки БД-приложений с использованием промышленных СУБД других производителей рекомендуется использовать либо технологию Borland dbExpress, которая пришла на смену SQL-соединений BDE, либо компоненты сторонних производителей (например, Oracle Data Access Components для работы с СУБД Oracle).

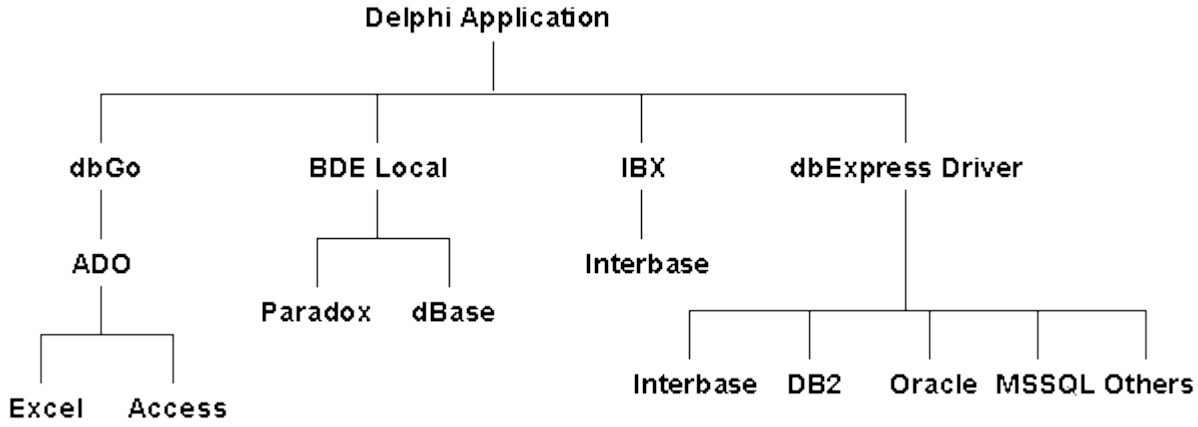


Рис. Концепция использования технология доступа к данным при разработке БД-приложений средствами Borland

Использование технологии ADO для создания приложений для работы с базами данных в среде Borland C++ Builder 6 практически не отличается от ранее рассмотренного использования компонентов BDE. Чтобы использовать наборы данных TADOTable, TADOQuery или TADOStoredProc, достаточно корректно настроить компонент TADOConnection, отвечающий за соединение с источником данных. Основными свойствами этого компонента являются:

- Свойство ConnectionString – содержит строку инициализации соединения, то есть список необходимых параметров. Исходя из вышесказанного важнейшим параметром соединения посредством технологии ADO является наименование поставщика данных OLE DB. Кроме того, необходимо указать наименование источника (базы) данных, а также имя пользователя и пароль. Для быстрого формирования строки инициализации соединения необходимо перейти в режим редактирования свойства ConnectionString в инспекторе объектов и щелкнуть по появившейся в правой части строки редактирования кнопке. В появившемся окне следует выбрать способ определения связи с данными путем формирования строки соединения («Use Connection String») и нажать кнопку Build….

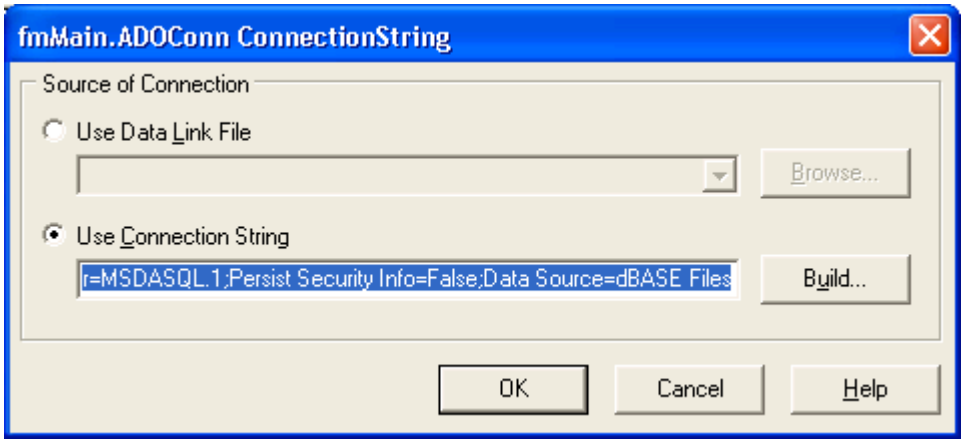


Рис. Редактор настройки соединения ADO

Свойства связи с источником данных, которые войдут в строку инициализации соединения, задаются в окне, изображенном на рисунке. Важнейшим свойством, как уже было сказано, является наименование поставщика (провайдера) данных OLE DB. Для файлов Microsoft Access можно выбрать, например, поставщика OLE DB для драйверов ODBC (Microsoft OLE DB Provider for ODBC Driver).

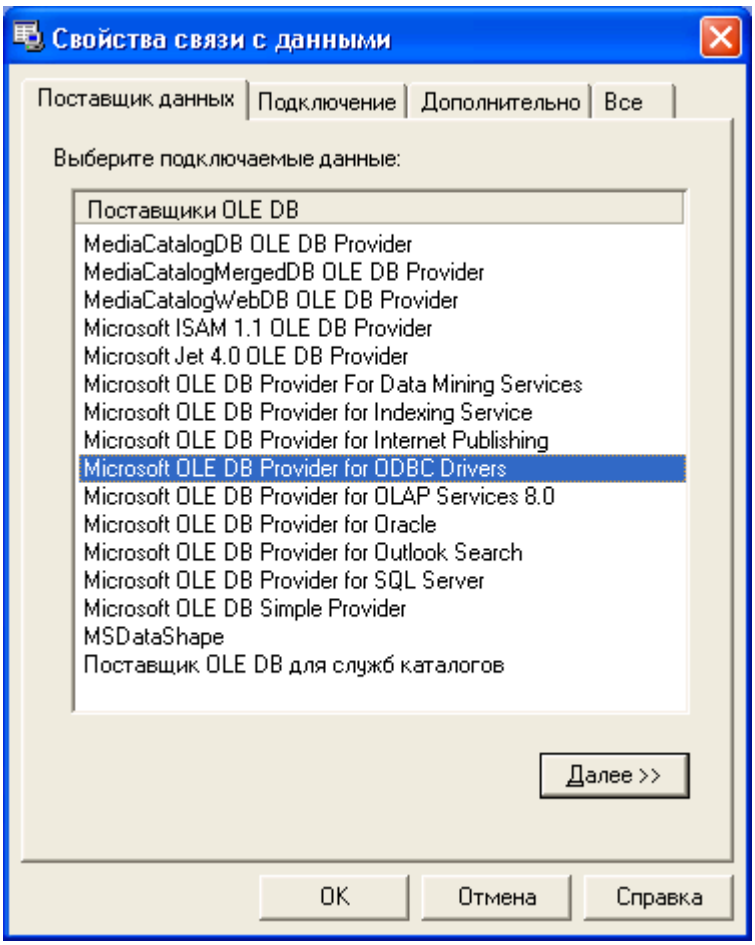


Рис. . Диалоговое окно настройки параметров соединения ADO.

В этом случае на вкладке «Подключение» необходимо будет указать следующие параметры подключения к источнику данных через указанного поставщика данных:

- Наименование источника данных ODBC (можно выбрать из списка существующих источников данных);
- Имя пользователя и пароля (обычно для клиент-серверных СУБД);
- Наименование начального каталога (например, наименование базы данных, с которой устанавливается соединение; для файлов Microsoft Access этот параметр можно оставить пустым).

На вкладке «Подключение» можно сразу же проверить корректность введенных значений параметров, для чего используется кнопка «Проверить подключение». В случае, если параметры определены правильно, появится сообщение «Проверка подключения выполнена».

Параметры соединения, которые можно задавать на других вкладках рассматриваемого диалогового окна являются необязательными и в рамках данной работы не рассматриваются.

После настройки всех параметров соединения необходимо нажать кнопку «OK». Сформированная строка инициализации будет отображаться в окне редактора настройки соединения ADO.

- Свойство Connected, указывающее состояние соединения (true – «установлено» и false – «не установлено»).
- Свойство LoginPrompt, включающее/выключающее режим автоматического запроса имени и пароля пользователя при попытке установить соединение (true – «включен» и false – «выключен»).

Остальные компоненты ADO (TADOTable, TADOQuery, TAdOStoredProc и др.) могут обращаться к источнику данных напрямую, либо через компонент TADOConnection. В первом случае для этих компонентов необходимо формировать строку параметров соединения для свойства ConnectionString точно так же, как это было рассмотрено выше. Во втором случае наименование компонента TADOConnection следует указать в качестве значения свойства Connection. Все свойства и

методы работы с наборами данных, рассмотренные в лабораторных работах №№ 3-5, могут быть использованы и при работе с компонентами наборов данных ADO.

Задания

1. Создать новую базу данных в Microsoft Access, состоящую из единственной таблицы DUMMY с двумя полями: uid (счетчик) и Value (текст).
2. Создать поименованный источник данных ODBC для работы с новой базой данных.
3. Поместить на форму нового приложения C++ Builder пару компонентов TDatabase и TTable. Настроить параметры компонентов таким образом, чтобы через набор данных TTable можно было работать с таблицей DUMMY.
4. Поместить на форму пару компонентов TADOConnection и TADOTable. Открыть диалоговое окно настройки параметров соединения . Убедиться в наличии поставщика данных Microsoft OLE DB Provider for ODBC Drivers. Проверить наличие поставщика данных Microsoft Jet OLE DB Provider.
5. Используя указания к лабораторной работе, сформировать для компонента TADO-Connection строку соединения ADO с созданной ранее БД через поставщика данных Microsoft OLE DB Provider for ODBC Drivers.
6. Настроить компоненту TADOTable для работы с таблицей DUMMY.
7. Если поставщик данных Microsoft Jet OLE DB Provider имеется в списке доступных поставщиков, то поместить на форму еще одну пару компонентов TADOConnection и TADOTable и повторить шаги 6 и 7 для их настройки. Использовать для соединения с БД поставщик Microsoft Jet OLE DB Provider.
8. Измерить скорость обращения к данным БД с использованием настроенных компонентов. Для этого с каждым набором данных формы выполнить следующие операции:

а. Циклическое добавление в пустую таблицу большого количества записей (несколько десятков тысяч). Значение поля Value генерировать как строку из 100-200 символов случайным образом (дублирование строк не допускается).

б. Открывание заполненной таблицы и выполнение фильтрации набора данных по определенному критерию.

в. Открывание заполненной таблицы с упорядочением записей по возрастанию значений неиндексированного поля Value и выполнение фильтрации набора данных по критерию, выбранному в предыдущем пункте.

г. Открывание заполненной таблицы с упорядочением записей по возрастанию значений индексированного поля Value и выполнение фильтрации набора данных по критерию, выбранному в пункте б.

Замерить время выполнения каждой операции для каждого набора данных.

4. Методические указания к самостоятельной работе студентов

Самостоятельная работа по дисциплине «Распределенные системы обработки информации», направлена на углубление и закрепление знаний студента, на развитие практических умений и включает в себя следующие виды работ:

– работа с лекционным материалом, учебниками и учебными пособиями;
– изучение тем, вынесенных на самостоятельную проработку;
– подготовка к практическим занятиям;
– выполнение домашних индивидуальных заданий;
– подготовка к лабораторным работам;
– подготовка к текущему и итоговому контролю.