Распределенные системы
Базовые концепции

Распределенные системы:определения

- “...система нескольких автономных вычислительных узлов, взаимодействующих для выполнения общей цели.”

Распределенные системы:определения

- Система, чьи компоненты размещены на различных узлах взаимодействующие и управляемые только посредством передачи сообщений.
  Существуют системы с разделяемой памятью (или с разделяемым временем)

Распределенные системы:определения

- ”Система, состоящая из набора двух или более независимых узлов которые координируют свою работу посредством синхронного или асинхронного обмена сообщениями.”

Распределенные системы:определения

- “распределенная система это набор независимых узлов (компьютеров), которые представляются пользователю как один компьютер.” [Tanenbaum]
- “распределенная система это собрание независимых компьютеров соединенных сетью с программным обеспечением, обеспечивающим их совместное функционирование.”

Последствия...

- Параллельность
  - Независимые процессы
    - ✓ Синхронизация
  - Необходимость разделения ресурсов
    - ✓ Данные
    - ✓ Сервисы
    - ✓ Устройства
  - Типичные проблемы
    - ✓ Deadlocks
    - ✓ Ненадежные коммуникации (проблема освобождения ресурсов)

Последствия...

- Нет “глобального” времени
  - Асинхронная передача сообщений -
  - Ограниченная точность синхронизации часов
- Нет состояния системы
  - Нет ни одного процесса в распределенной системе, который бы знал текущее глобальное состояние системы
    - ✓ Следствие параллелизма и механизма передачи данных

Последствия...

- Сбои
  - Процессы выполняют автономно, изолированно
  - Неудачи отдельных процессов могут остаться необнаруженными
  - Отдельные процессы могут не подозревать о общесистемном сбое
  - Сбои происходят чаще чем в централизованной системе
  - Новые причины сбоев (которых не было в монолитных системах)
  - Сетевые сбои изолируют процессы и фрагментируют систему

Принципы разделения

- **Функциональное разделение**: узлы выполняют различные задачи
  - Клиент / сервер
  - Хост / Терминал
  - Сборка данных/ обработка данных
    - ✓ Решение - создание разделяемых сервисов
- **Естественное разделение** (определяемое задачей)
  - Система обслуживания сети супермаркетов
  - Сеть для поддержки коллективной работы

Принципы разделения

- **Распределение нагрузки/балансировка**: назначение задачи на процессора так, чтобы оптимизировать общую загрузку системы.
- **Усиление мощности**: различные узлы работают над одной задачей
  - Распределенные системы содержащие набор микропроцессоров, по мощности могут приближаться к суперкомпьютеру
  - 10000 CPU, каждый 50 MIPS, вместе 500000 MIPS -> команда выполняется за 0.002 nsec -> свет проходит 0.6 mm -> любой существующий чип - больше!

Принципы разделения

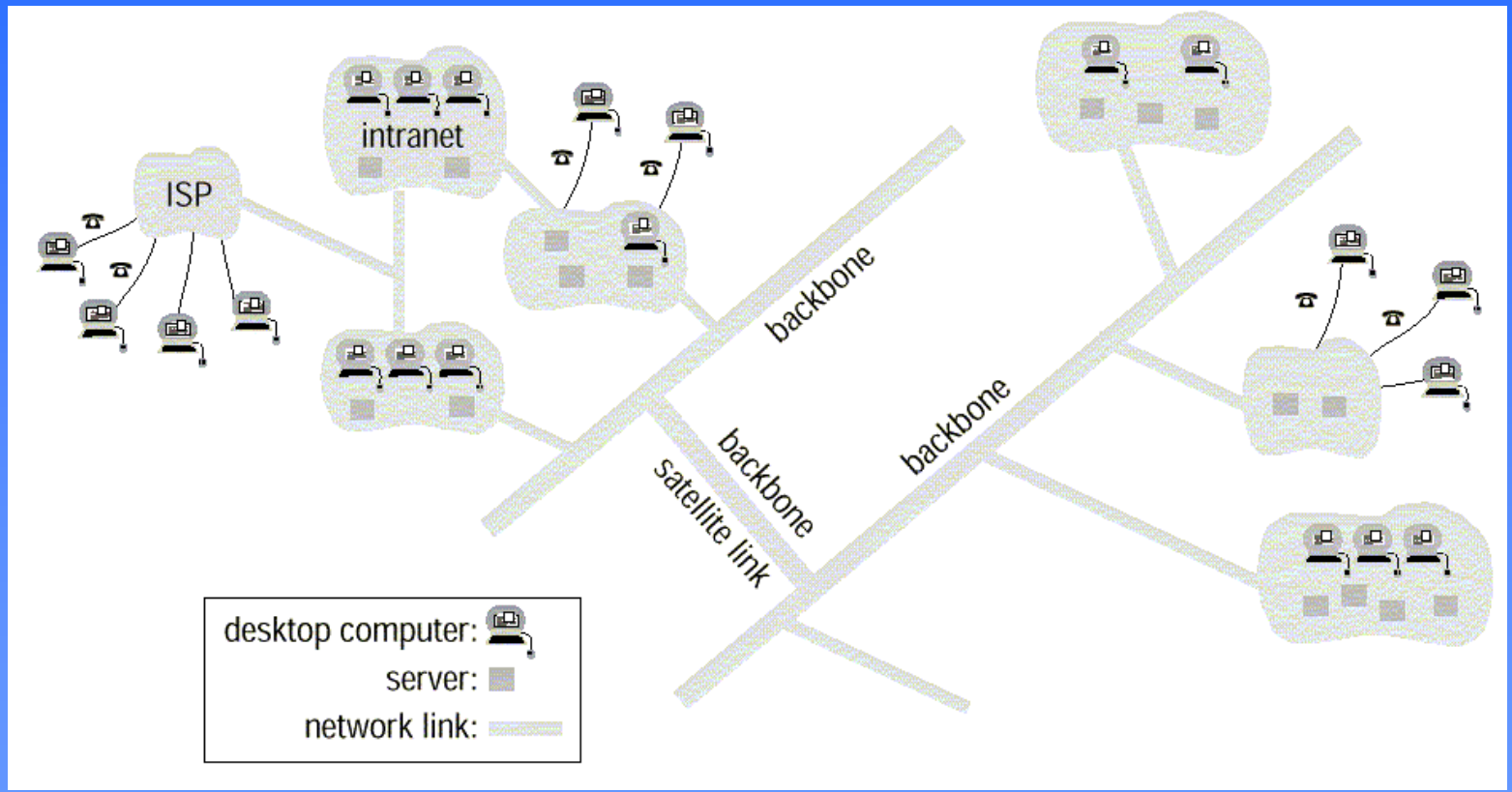
- **Физическое разделение**: система строится в предположении, что узлы физически разделены (требования к надежности, устойчивости к сбоям).
- **Экономические**: набор дешевых чипов может обеспечить лучшие показатели отношения цена/производительность, чем мэйнфрэйм
  - Мэйнфрэйм: 10 раз быстрее, 1000 раз дороже

Примеры распределенных систем

- Internet (?)
- Intranet
- Вычислительные кластера
- …

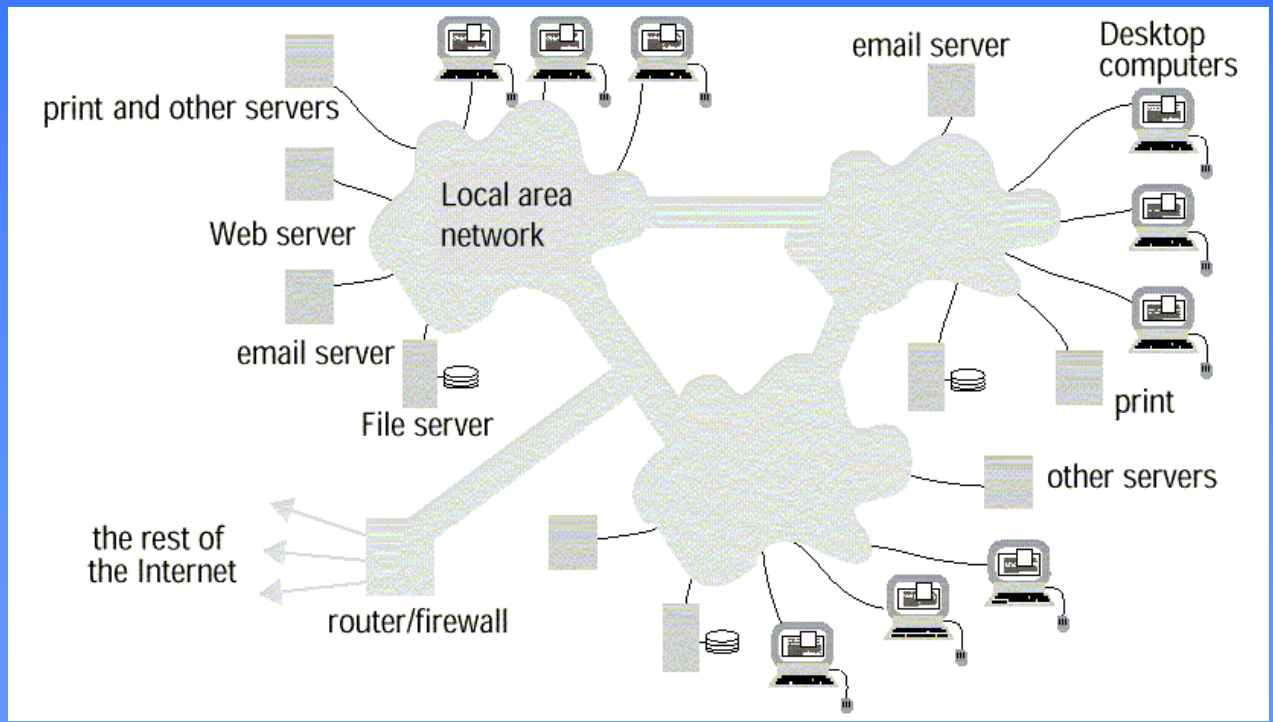
Пример: Internet

- Гетерогенная сеть компьютеров и приложений
- Реализация взаимодействия - IP стек



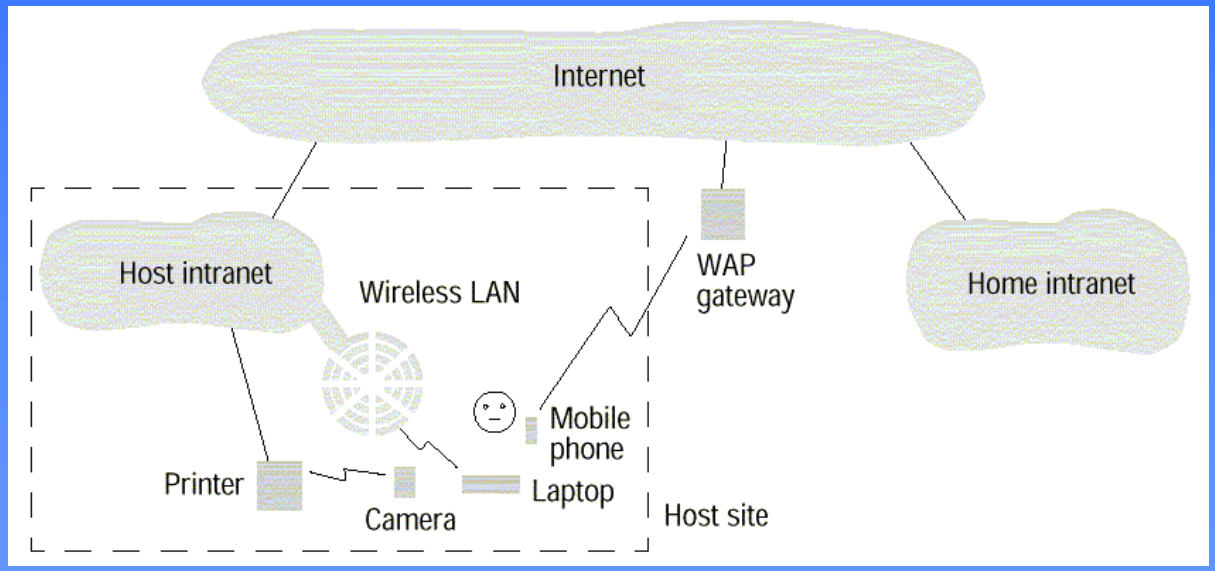
Пример: Intranet

- Администрируется локально
- Взаимодействие с Internet
- Обеспечивает сервисами (внутренних и внешних пользователей)



Пример: Wireless Information Devices

- Система сотовой связи (GSM)
  - Ресурсы разделяемы (радио частота, время передачи на частоте,...)
- Laptop (подключаются к Wireless LAN)
- Handheld, PDAs etc.



Другие примеры

- Системы управления аэропортом
- Автомобильные управляющие системы

  Mercedes S класса сегодня имеет более 50 автономных встроенных процессоров соединенных общей шиной

Примеры...

- Телефонные системы
- Сложные сети предприятий
- Сетевые файловые системы
- WWW
- и многое другое...

Разделение ресурсов

- Разделение ресурсов часто является одной из причин разработки распределенной системы
  - Уменьшается стоимость, (file и print сервера)
  - Разделение данных между пользователями (совместная работа над проектом)
- Сервисы
  - Управляют набором ресурсов
  - Представляют услуги пользователям

Разделение ресурсов

- Сервер используется для предоставления сервисов
  - Принимает запросы на обслуживание от клиентов
    - ✓ вызов операции
  - Прием сообщения/ответ на сообщение
    - ✓ полная реализация - удаленный вызов
  - Роли клиента и сервера меняются от вызова к вызову
    - ✓ один и тот же процесс может быть как клиентом, так и сервером
  - Терминология Клиент/Сервер применяется к процессам, а не к узлам!!!

Проблемы

- Распространение приложения
- Гетерогенность
- Открытость
- Безопасность
- Масштабируемость
- Обработка ошибок и восстановление после сбоев
- Параллелизм
- Прозрачность
- Управляемость

Распространение приложения

- Фрагментация
  - разделение приложения на модули для распространения
- Конфигурация
  - Связь модулей друг с другом (зависимости)
- Размещение
  - выгрузка модулей на целевую систему
  - Распределение вычислительных модулей между узлами (статическое или динамическое)

Гетерогенность

- Гетерогенные = разные
- Различные
  - сетевые инфраструктуры,
  - hardware&software (пример Intel & Motorolla, UNIX sockets & Winsock calls),
  - языки программирования (и представления данных!!!)
- Различия должны быть скрыты

Гетерогенность

- Интерфейсы и реализация могут быть разными
  - Базовые концепции обычно неизменны
- Необходимы стандарты

Гетерогенность

- Middleware: промежуточный программный слой
  - позволяет гетерогенным узлам взаимодействовать
  - Определяет однородную вычислительную модель
  - Поддерживает один или несколько языков программирования
  - Обеспечивает поддержку распределенных приложений
    - ✓ Вызов удаленных объектов
    - ✓ Удаленный вызов SQL
    - ✓ Распределенная обработка транзакций
- Примеры: CORBA, Java RMI, Microsoft DCOM

Гетерогенность

- Мобильный код: код разработан для миграции между узлами
  - Необходимо преодолевать аппаратные различия (разные наборы инструкций)
- Виртуальные машины
  - Компилятор «изготавливает» байт-код для VM
  - VM реализована для всех аппаратных платформ (Java)
- Методы грубой силы
  - Портируем код под каждую платформу...

Открытость

- Гарантирует расширяемость
- Возможность повторного использования
- Важные факторы:
  - Наличие четких спецификаций
  - Наличие полной документации
  - Опубликованные интерфейсы
  - Тестирование и проверка на многих платформах

Безопасность

- Три компонента:
  - Защищенность
  - Целостность
  - Доступность
- Задача: посылка значимой информации по сети безопасно и эффективно

Безопасность

- Сценарий 1: Доступ к результатам тестирования по NFS
  - Откуда мы знаем, что пользователь - преподаватель, имеющий доступ к данным?
  - Авторизация
- Сценарий 2: Посылка номера кредитной карты в интернет-магазин
  - Никто кроме получателя не должен прочитать данные
  - Криптография

Безопасность

- Нерешенные проблемы
  - Атаки типа DoS (отказы в обслуживании)
  - Безопасность мобильного кода
    - ✓ Непредсказуемые эффекты
    - ✓ Может вести себя подобно троянскому коню...

Масштабируемость

- Распределенная система масштабируема, если она остается эффективной при увеличении числа обслуживаемых пользователей или ресурсов
- Проблемы:
  - Контроль стоимости ресурсов
  - Контроль потерь производительности

Масштабируемость

- Стоимость физических ресурсов
  - Растет,при увеличении числа пользователей
  - Не должна расти быстрее, чем $O(n)$, где n = количеству пользователей
- Потери производительности
  - Увеличиваются с ростом размера данных (и количества пользователей)
  - Время поиска не должно расти быстрее, чем $O(\log n)$, где n = размер данных

Масштабируемость

- Существуют естественные ограничения
  - Некоторые определяются легко
  - Другие труднее
- Обход узких мест
  - Децентрализация алгоритмов
    - ✓ Пример - Domain Name Service
  - Тиражирование и кэширование данных

Обработка сбоев

- Сбои более частые, чем в централизованных системах, но обычно локальные
- Обработка сбоев включает
  - Определение факта сбоя (может быть невозможно
  - Маскирование
  - Восстановление

Обработка сбоев

- Диагностика
  - Может быть возможна (ошибки передачи - контрольная сумма)
  - Может быть невозможна (удаленный сервер не работает или просто очень загружен?)

Обработка сбоев

- Маскирование
  - Многие сбои могут быть скрыты
  - Может быть невозможно (все диски повреждены)
  - Не всегда хорошо

Параллелизм

- Контроль параллелизма
  - Обращение нескольких потоков к ресурсу
    - ✓ Правильное планирование доступа в параллельных потоках (устранение взаимоисключений, транзакции)
  - Синхронизация (семафоры)
    - ✓ Безопасно, но уменьшают производительность
  - Разделяемые объекты(ресурсы) должны работать корректно в многопоточной среде

Прозрачность

- Сокрытие гетерогенной и распределенной структуры системы так, чтобы пользователю система представлялась монолитной

Прозрачность

Прозрачность доступа: доступ к локальным и удаленным ресурсам посредством одинаковых вызовов

Прозрачность расположения: доступ к ресурсам вне зависимости от их физического расположения

Прозрачность параллелизма: возможность нескольким процессам параллельно работать с ресурсами, не оказывая влияния друг на друга

Прозрачность репликации: возможность нескольким экземплярам одного ресурса использоваться без знания физических особенностей репликации.

Прозрачность обработки ошибок: Защита программных компонентов от сбоев, произошедших в других программных компонентах. Восстановление после сбоев

Прозрачность мобильности:Возможность переноса приложения между платформами, без его переделки

Прозрачность производительности: возможность конфигурации системы с целью увеличения производительности при изменении состава платформы выполнения

Прозрачность масштабируемости: возможность увеличения производительности без изменения структуры программной системы и используемых алгоритмов

Прозрачность

- Очень важна для распределенных систем
  - Прозрачность доступа и физического расположения
  - Имеет критическое значения для должного использования распределенных ресурсов

Управляемость

- Распределенные ресурсы не имеют центральной точки управления
- Локальная оптимизация не всегда означает глобальную оптимизацию
  - Нужен глобальный взгляд на проблему
  - Не всегда возможен (есть системы, никому конкретно не принадлежащие)

Итоги

- Распределенная система:
  - Автономные (но соединенные средой передачи данных) узлы
  - Взаимодействие посредством передачи сообщений
- Много примеров того, что распределенные системы нужны и их нужно уметь строить
- Распределенные системы существуют и их нужно уметь развивать и поддерживать

Модели архитектуры

- Модель архитектуры распределенной системы должна содержать решение двух проблем:
  - Физическое размещение компонентов между узлами
  - Взаимодействие между компонентами.

Уровни

- Приложения, сервисы
- Middleware
- Операционная система
- Аппаратура

Возможные архитектуры

- Клиент - сервер
- Модель предоставления услуг пулом серверов
- Модель прокси – и кэш - серверов
- Модель равных процессов

Вариации на тему Клиент-сервер

- Мобильный код
- Мобильные агенты
- Network computers
- Тонкие клиенты
- X - window

Требования к дизайну

- Требования, накладываемые обеспечением требуемой производительности
- Использование кэширования и репликации
- Требование надежности

Требования к производительности

- Время отклика
- Производительность.
- Балансировка нагрузки

Использование кэширования и репликации

- Очень многие проблемы производительности системы могут быть решены путем кэширования данных .

Модели

- Модель взаимодействия
- Модель защиты от сбоев
- Модель безопасности

Модель взаимодействия

- Производительность линий связи
- Время и события
- Асинхронный и синхронный обмен

Модели распределенных систем

- Модели архитектуры
  - Размещение компонентов
  - Связи между компонентами
  - Пример Уровни протоколов, модель клиент-сервер, одно-ранговая модель
- Модели функционирования
  - Формальное описание свойств системы, общих во всех моделях архитектуры
  - Взаимодействие, устойчивость к сбоям, модели безопасности
  - …

Модели архитектуры

- Архитектура: структура системы и отдельно определенных компонент
- Цель: Структура должна отвечать требованиям
  - Надежности
  - Управляемости
  - Адаптируемости
  - Рентабельности

Модели архитектуры

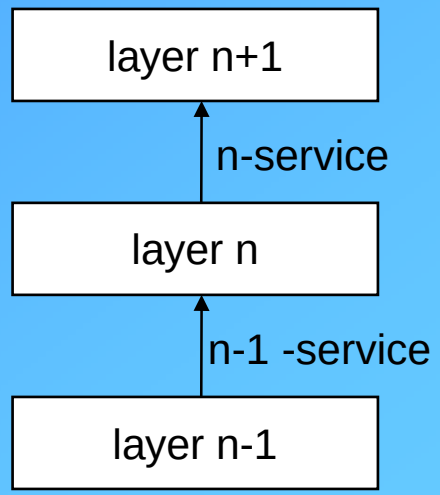
- Функции индивидуальных компонентов не интересны
  - Рассмотрим на практике
- Рассмотрим на лекции
  - Размещение (по узлам сети)
  - Модели для распределения данных и рабочей нагрузки
  - Модели взаимодействия

Модели архитектуры

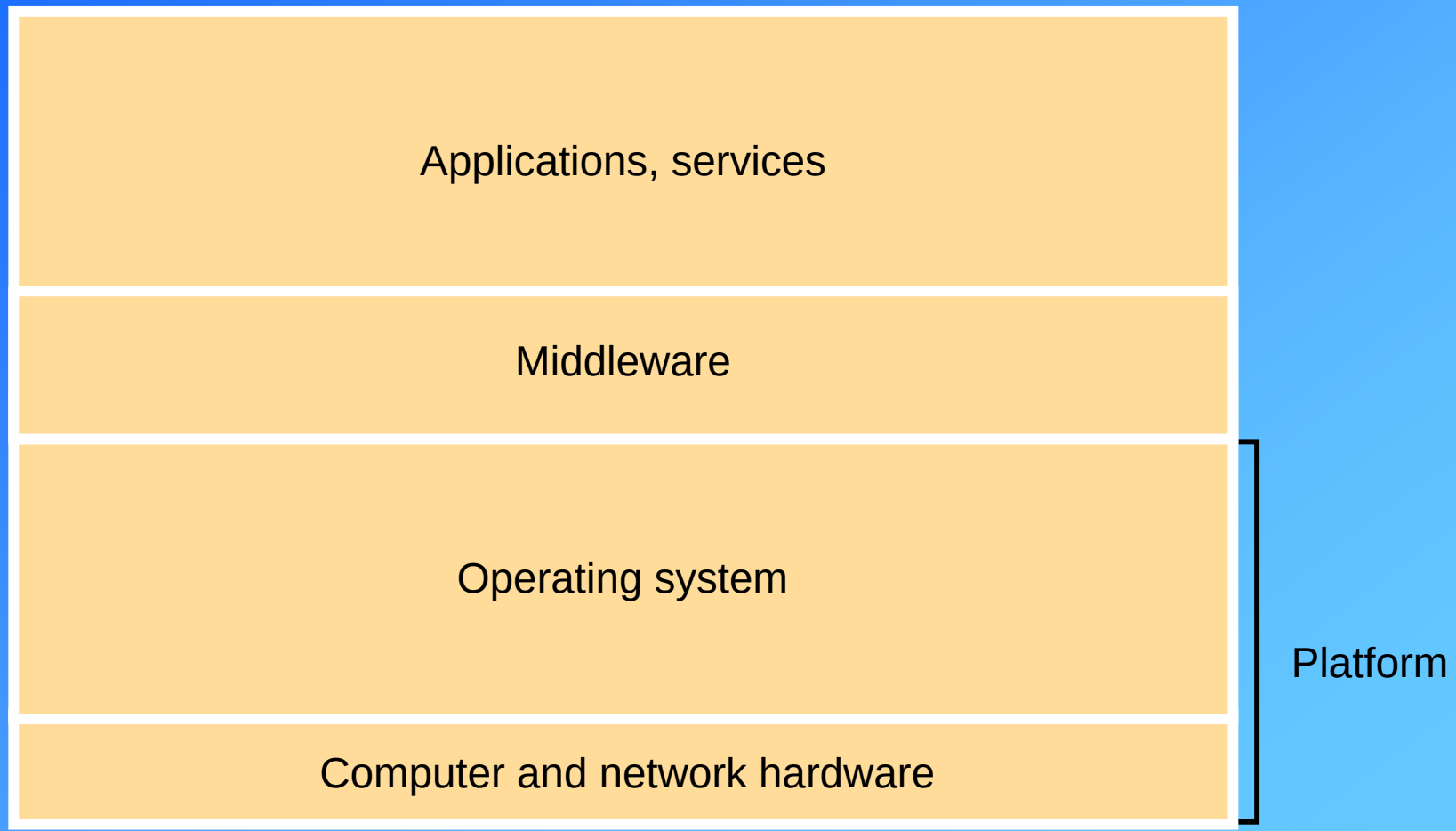
- Различие между клиентом, сервером и моделью равных процессов
  - Оценка рабочей нагрузки
  - Анализ сбоев
- Модели архитектуры могут использоваться, чтобы определить размещение компонентов
- Большинство систем могут быть построены как вариации модели клиент-сервер
  - Перемещение кода
  - Добавление/удаление узлов
    - ✓ Перемещение сервисов

Уровни

- Основная идея - уменьшать сложность систем, разделяя их на слои и сервисы
  - уровень: группа сильно связанных и закрытых элементов, реализующих одну функциональность
  - service: функциональность, обеспечиваемая для вышестоящего слоя
- Примеры разделения на уровни
  - Операционные системы (ядро, утилиты)
  - Архитектура сетевых протоколов



Уровни


Applications, services
Middleware
Operating system
Computer and network hardware
Platform

Уровни: Платформа

- Платформа: Операционная система и оборудование
  - Предоставляет интерфейс системного программирования

Уровни: Middleware

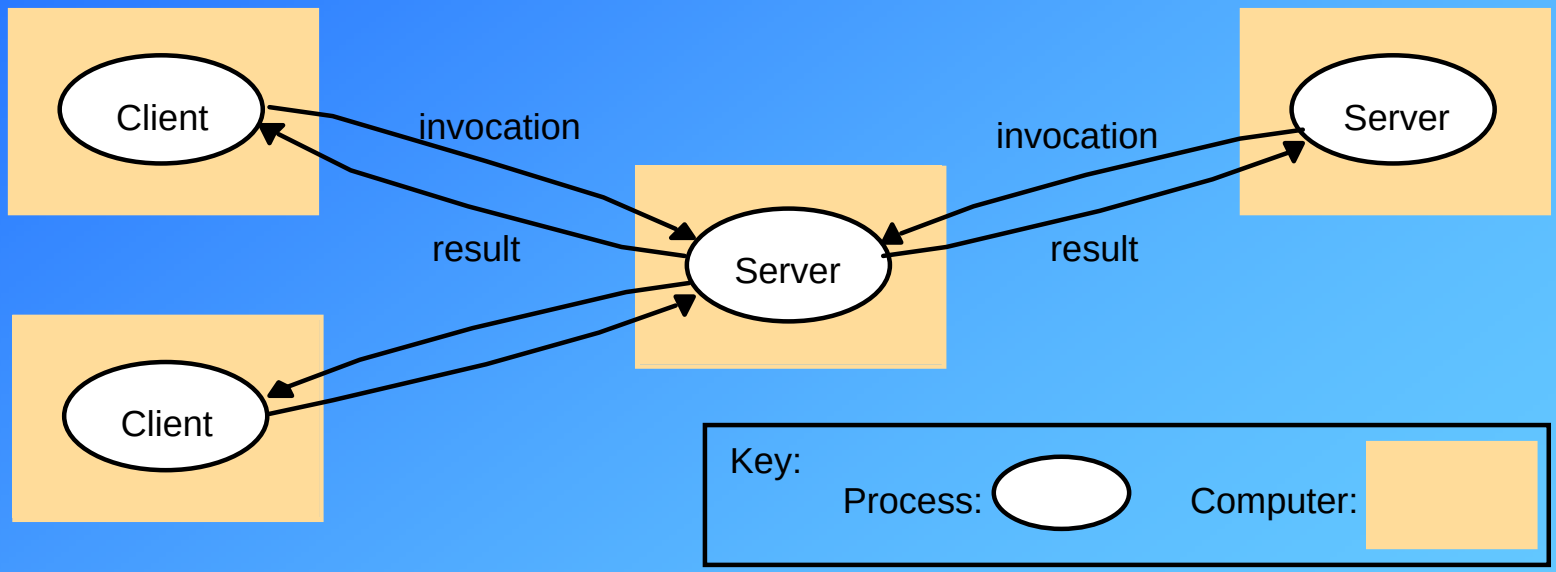
- Middleware: “Слой программного обеспечения, чья цель состоит в том, чтобы скрыть неоднородность и обеспечивать удобную модель программирования для разработчиков”
  - Предоставляет прикладной интерфейс программирования
  - Примеры
    - ✓ CORBA (OMG)
    - ✓ DCOM (Microsoft)
    - ✓ Remote Procedure Call (Sun)
    - ✓ Java Remote Method Invocation (Sun)
  - Может также предоставлять услуги(сервисы)

Уровни: Middleware

- Сервисы
  - Именования
  - Безопасности
  - Транзакций
  - Долговременного хранения
  - Уведомления о событиях
  - …
- Ограничения
  - Некоторые функции могут быть правильно выполнены только при наличии информации, полученной с прикладного уровня

Модель Клиент-Сервер

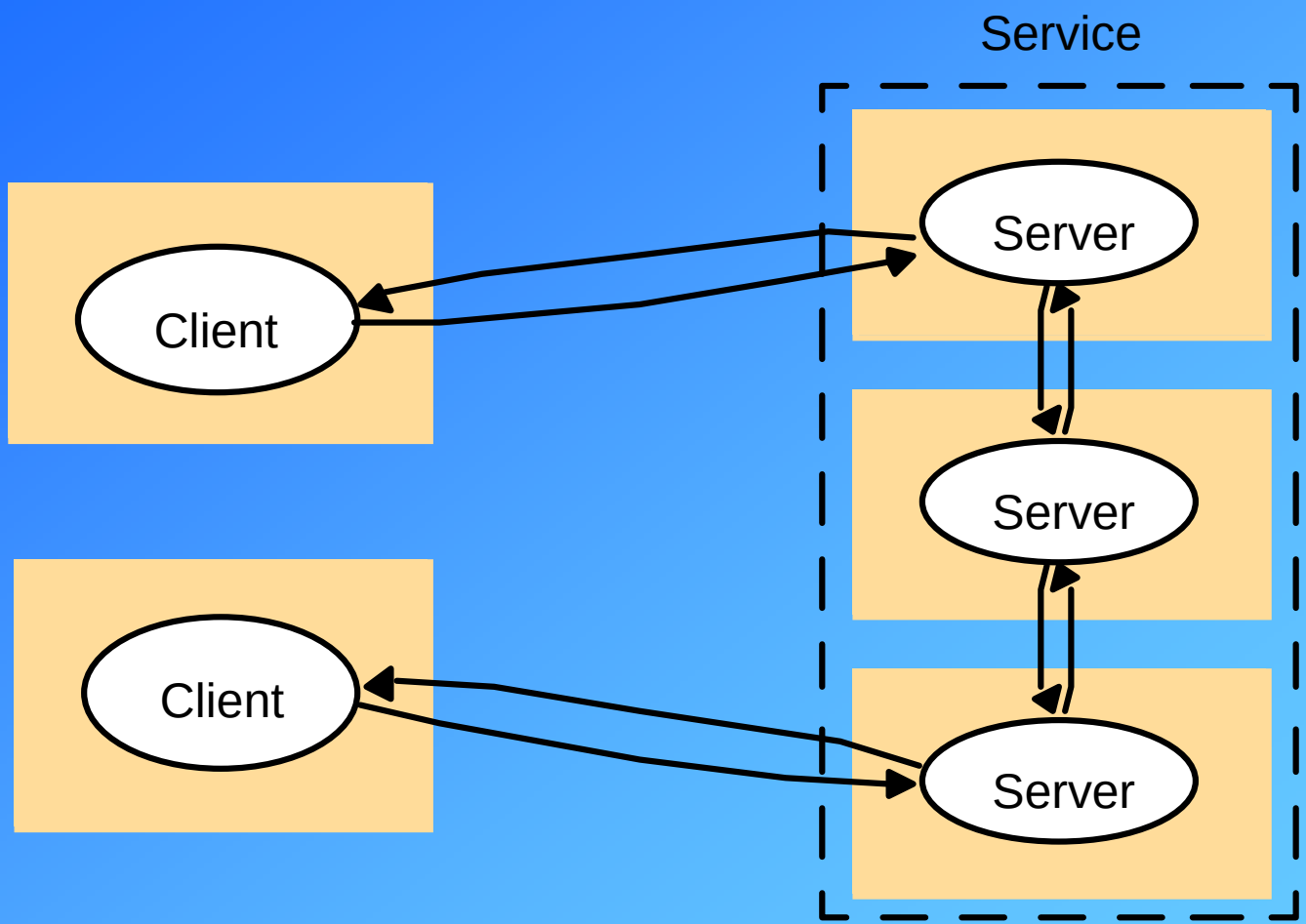
- Наиболее широко используемый вариант



Модель Клиент-Сервер

- Клиент: Процесс, желающий получить доступ к данным, использует ресурсы и/или выполняет действия на удаленном узле
- Сервер: Процесс, управляющий данными и всеми другими разделяемыми ресурсами, обеспечивающий клиентам доступ к ресурсам и производящий вычисления
- Взаимодействие: пары запрос / результат
- Пример
  - ✓ Http сервер: клиент (броузер) запрашивает страницу, сервер поставляет страницу

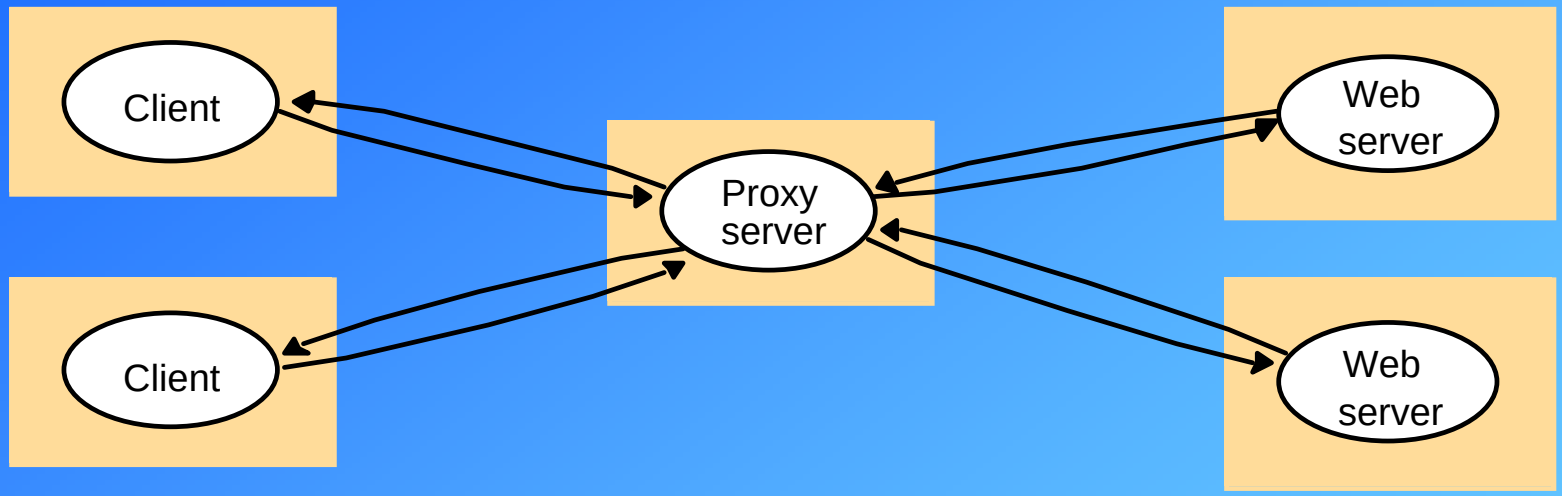
Пул Серверов



Пул Серверов

- Услуги могут обеспечиваться многими серверами
- Распределенные между серверами объекты
- Реплицированные объекты
  - Увеличение производительности, доступности и отказоустойчивости
- Но требуют координации копий / консистентности представления
- Например высодоступные серверы (порталы, диллинговые центры), информационные службы
- Серверы, обслуживающие распределенную БД

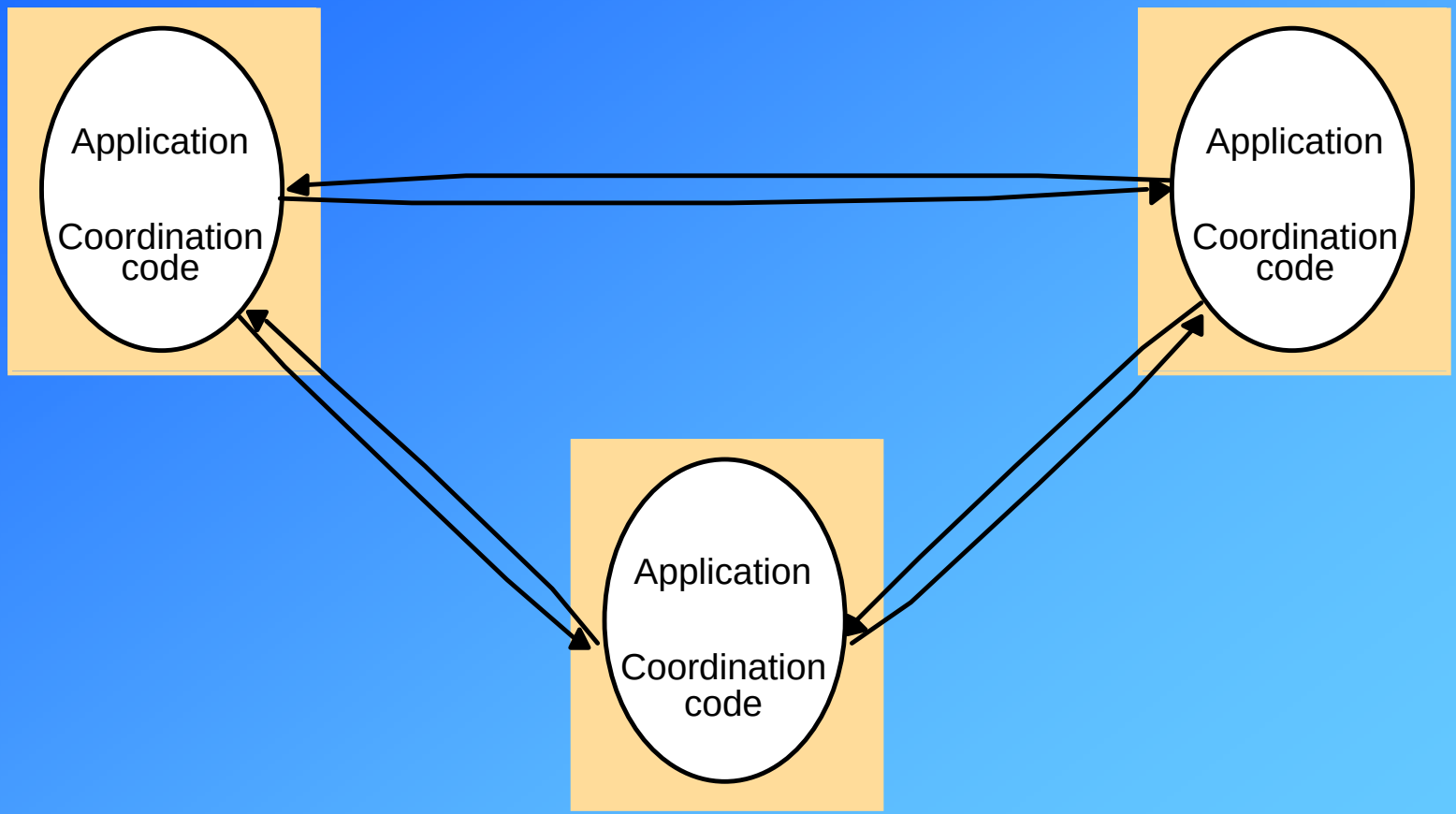
Прокси - сервера



Прокси - сервера

- Кэш: «близкая» копия, наиболее часто используемых данных
  - ЗНАЧИТЕЛЬНО повышает производительность большинства приложений
  - Но требует усилий по поддержанию когерентности
- Прокси-сервер: разделяемый кэш ресурсов
  - Еще сложнее, чем простой Кэш…
  - Обычно используется (и хорошо подходит) для доступа к веб - ресурсам

Равноправные процессы



Равноправные процессы

- Равные процессы: процессы, которые играют равные роли
  - Никакого абсолютного различия между клиентом / сервером
  - Роли клиента и сервера различаются от вызова к вызову (или со временем)
- Увеличивает задержку при обращении к локальным объектам
- Увеличивает устойчивость к сбоям и масштабируемость
- Трудности с координацией
- Примеры: роутинг, распределенное вычисление, серверы новостей

Вариации модели клиент-сервер

- Вариации возможны по следующим параметрам:
  - Связь инициируется сервером
  - Использование мобильного кода или мобильных агентов
  - Легкие клиенты базирующиеся на потребности пользователей в дешевых компьютерах и простом управлении (тонкий клиент)

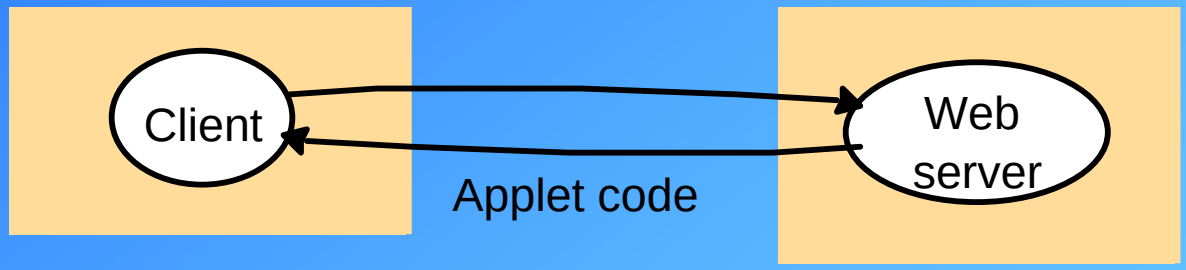
Вариации модели клиент-сервер

- Модель сервера - инициатора
  - Сервер начинает диалог
  - “Помещает” (заталкивает) информацию клиенту
  - Клиент просматривает посылки сервера

Вариации модели клиент-сервер

- Мобильный код: Код, посланный процессу клиента, чтобы выполнить определенную задачу

a) client request results in the downloading of applet code

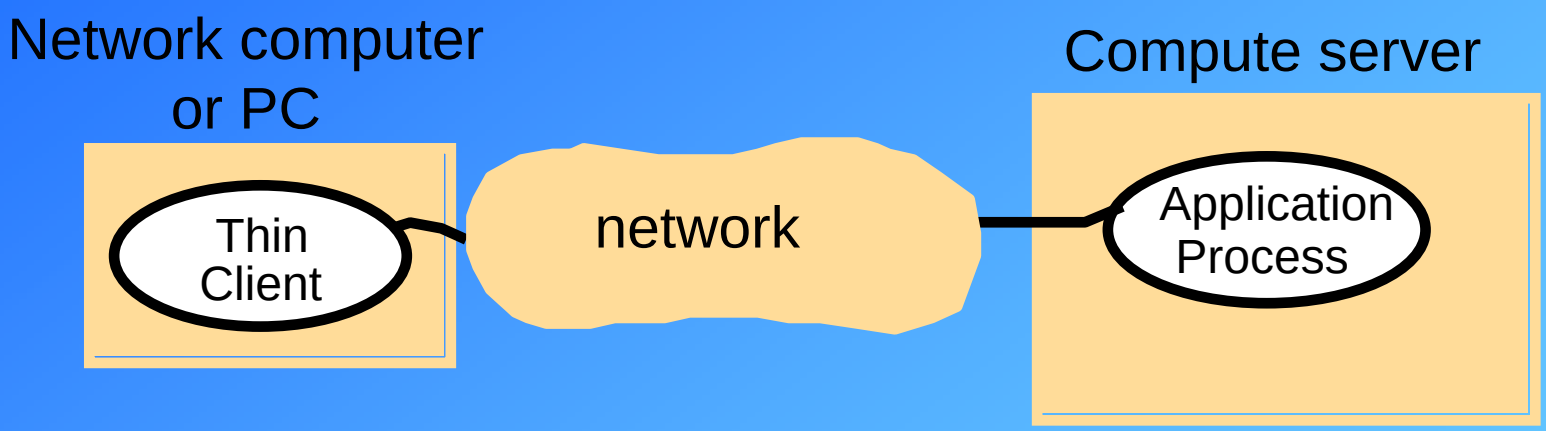


b) client interacts with the applet

Вариации модели клиент-сервер

- Мобильные агенты
  - Выполнение программы (код + данные) которая перемещается между узлами в сети
    - ✓ Выполняет автономную задачу обычно под управлением некоторого другого процесса
    - ✓ Имеет внутреннее знание и цели
  - Преимущество: всюду доступ локальный!!!
    - ✓ Снижает затраты на коммуникации
  - Потенциальная угроза безопасности
    - ✓ Ограниченная применимость
  - Например, сбор данных из многих источников, установка программ, программы типа червей (электронная почта)

Вариации модели клиент-сервер



Вариации модели клиент-сервер

- Network computer
  - Все файлы сохраняются на удаленном носителе
  - Минимум локального программного обеспечения
  - Любой локальный диск используется только под кэш
- Тонкий клиент
  - Выполняет интерфейс пользователя на локальном компьютере
  - Программы работают на мощном вычислительном сервере
  - Citrix WinFrame, X11 , …

Примитивы передачи данных

- Распределенные системы нуждаются в обмене данными и синхронизации между автономными распределенными процессами
  - Interprocess communication (IPC)
    - ✓ Разделяемые переменные
    - ✓ Передача сообщений
- Синхронизация
  - Процессы на различных компьютерах выполняются с различными скоростями
- Примитивы передачи данных
  - **send** *expression_list* **to** *destination_designator*
  - **receive** *variable_list* **from** *source_designator*
- Вопросы
  - Как определелять адреса (как указывать процесс, которому передаются данные)?
  - Как добиваться синхронизации при передаче?

Примитивы передачи данных

- Указание назначения(адресата)
  - Прямое наименование: имена процессов получателя и отправителя используются в качестве адресатов (пара имен однозначно определяет канал)

```
send cur_status to monitor
receive message from handler
```

    - ✓ Просто реализовать и использовать
    - ✓ Позволяет процессу легко контролировать когда получать какое сообщение с какого процесса
    - ✓ Используется для реализации клиент-серверных приложений
      - ❑ *хорошо подходящий, чтобы реализовать схему клиент/сервер если есть один клиент и один сервер*
      - ❑ *в противном случае, сервер должен уметь принимать запросы от любого клиента в любое время,и клиент должен уметь вызвать много сервисов в одно время, если доступно больше одного сервера*

Примитивы передачи данных

- Указание назначения(адресата)
  - глобальные имена или почтовые ящики: независимое имя процесса-приемника может использоваться процессами - источниками
    - ✓ сообщения посланные в почтовый ящик могут получаться любым процессом
    - ✓ чтобы реализовать концепцию клиент-сервера
      - *клиенты посылают сообщения в почтовый ящик, освободившийся сервер их обрабатывает*
    - ✓ недостаток: дорогая реализация
      - *сообщение послано в ящик*
      - *если один процесс решил получить сообщение, он должен его заблокировать*
      - *взаимное исключение при параллельном доступе*
  - порты: почтовый ящик, но только одному процессу разрешаются получать сообщения
    - ✓ легко осуществимо - принимать может только один процесс - не нужна блокировка
    - ✓ подходит если один сервер и много клиентов
  - Указание адресов при программировании в интернет
    - ✓ гибридная прямая схема присваивания имен/порта
    - ✓ порты соответствуют концепции много процессов посылают, один принимает

Примитивы передачи сообщений

- Семантика примитивов передачи сообщений
  - Блокировка
    - ✓ не блокирующие: вызов не задерживает вызывающий процесс
    - ✓ блокирующий: вызов не возвращает управление до завершения
  - Синхронизация
    - ✓ синхронные нет никакой буферизации
      - *процессы синхронизируются по любому сообщению*
      - *источник блокируется до тех пор, пока получатель не будет готов к приему*
      - *приемщик блокируется, пока процесс - источник не будет готов к передаче*
    - ✓ асинхронные сообщения передаются с использованием неограниченного буфера
      - *передающий процесс выполняет передачу неограниченное число раз*
      - *передающий никогда не блокируется*
      - *принимающий блокируется на пустой очереди*
    - ✓ буферизованные буферизация с ограниченным буфером
      - *передающий процесс может передавать до тех пор, пока не переполнился буфер*
      - *передающий блокируется при переполненном буфере*
      - *принимающий блокируется на пустой очереди*

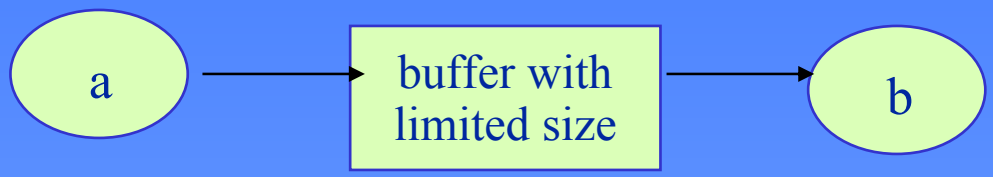
Примитивы передачи сообщений

- Не блокирующие примитивы для асинхронной или буферизированной передачи
  - прием
    - ✓ фоновый вариант:процесс продолжает выполняться, при приеме получает прерывание
      - ❑ *иногда сложно для реализации*
    - ✓ программные опросы
  - посылка
    - ✓ посылающий процесс ожидает освобождения буфера, или удаляет из него не посланные сообщения

Именованные каналы

• **Каналы** самые ранние и наиболее простые механизмы связи между процессами

• **Каналы** позволяют двум процессам взаимодействлвать через конечный буфер, реадизуемы й ОС данные сохраняются в порядке(FIFO)



• **Каналы** существуют только в момент передачи данных; процессы блокируются при полном буфере

Сокеты

• Сокеты возможно наиболее мощный механизм для IPC из-за уровня управления предоставленного программисту; но программист оказывается ответственным за многие детали связи

• Сокеты имеет две оконечные точки;

• Существует, покажив процесс

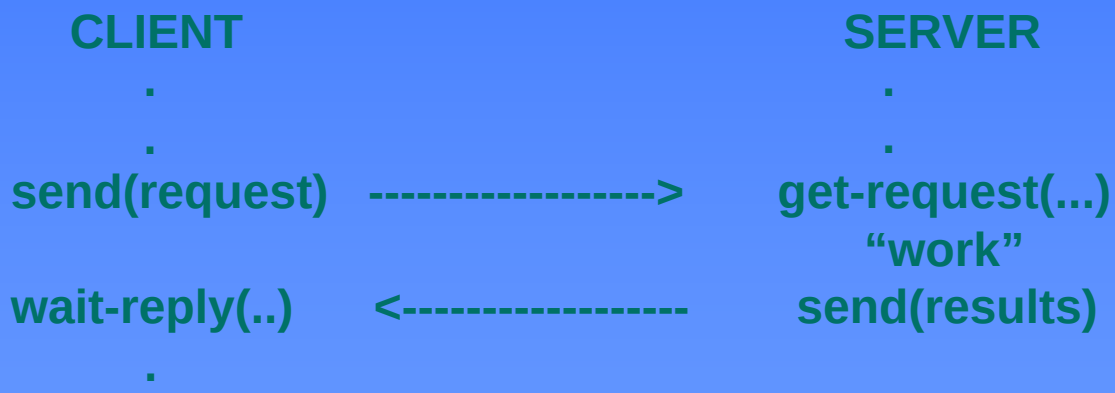
• Для его создания нужно выполнить ряд шагов

Сокеты

- **Создание сокета**
- **Связывание сокета с портом**
- **Прослушивание порта**
- **Приняти запроса от удаленного сокета. Создание соединения**
- **Обмен сообщениями**
- **Разъединение (закрытие сокета)**

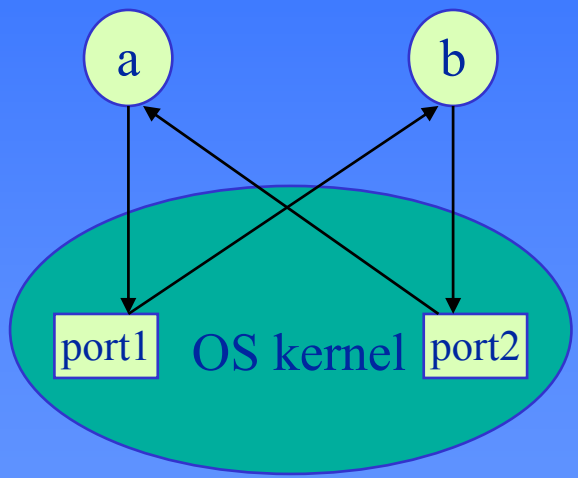
Модель взаимодействия клиента и сервера

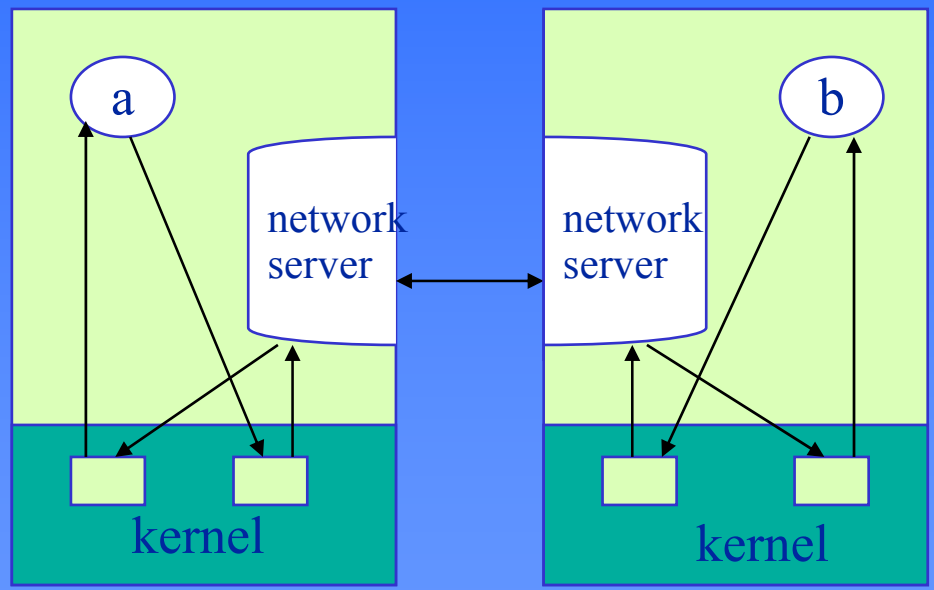
Типичное взаимодействие: Клиент посылает сообщение на сервер и блокируется, ожидая ответа; сервер принимает сообщение и обрабатывает его, передавая клиенту результат, что приводит к его разблокированию



Взаимодействие через порты

Порты контролируются ОС





Типы сообщений

C: клиент, S:сервер

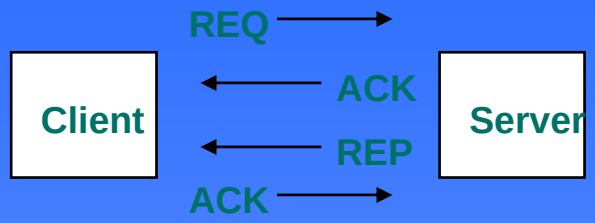
	Code	Type	From	To	Remarks
1.	REQ	request	C	S	Client wants service
2.	REP	reply	S	C	Reply from server to client
3.	ACK	ack	S/C	C/S	Previous msg arrived
4.	AYA	are you alive?	C	S	Enq. msg to check if server is functioning
5.	IAA	I am alive	S	C	Server responding that it is functioning
6.	TA	try again	S	C	Server is busy, has no room (e.g. no buffer space)
7.	AU	address unknown	S	C	No process is using this address

Типы сообщений (продолжение)

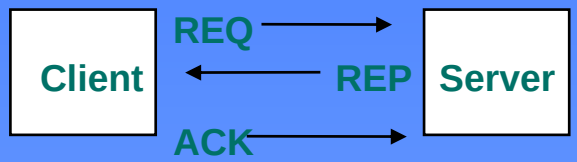
1-2типы 1 и 2 самые главные
3: для повышения надежности
4-7: не обязательны, но добавляют дополнительную функциональность
Необходимость в 4-5: Предположим, что клиент послал запрос. Что, если нет ответа, за приемлемое временя? Сервер все еще работает или сервер потерпел крах?
Клиент использует AYA сообщение для проверки сервера если IAA (или REP) сообщение получено, значит все в порядке; Иначе, если после нескольких AYA сообщений, нет обратных IAA/REP, клиент может предположить, что сервер недоступен.

Примеры

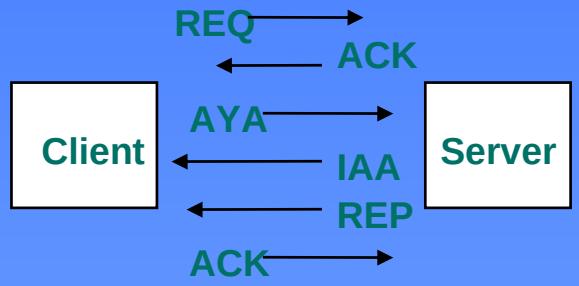
SIMPLE PROTOCOL WITH NO ACK



PROTOCOL WITH ACK FOR EACH MSG



PROTOCOL WHERE REP ALSO ACTS AS ACK



PROTOCOL WITH ACK AND PROBES

Механизм сокетов

- IPC базирующийся на UDP
  - Свойства UDP: нет гарантии порядка сообщений, сообщения теряются и дублируются
  - Необходимые шаги
    - ✓ создание сокета
    - ✓ связывание сокета с портом
      - *клиент: производный свободный порт*
      - *сервер: порт сервера*
  - Метод приемки: возвращает интернет адрес и порт отправителя + сообщение
  - Размер сообщения: IP разрешает сообщения до to $2^{16} = 65536$ bytes
    - ✓ Большинство реализаций ограничивают 8 Kb
    - ✓ Большие сообщения: увеличиваю производительность передачи
    - ✓ Если передаваемое сообщение слишком велико, оно усекается
  - Посылка – не блокирующая
  - Приемка блокирующая

Механизм сокетов

- Java API для UDP
  - ✓ Процесс создает сокет, посылает сообщение на порт 6789, и ждет получения отклика

```
import java.net.*;
import java.io.*;
public class UDPClient{
   public static void main(String args[]){
                // args give message contents and destination hostname
                try {
                                DatagramSocket aSocket = new DatagramSocket();      // create socket
                                byte [] message = args[0].getBytes();
                                InetAddress aHost = InetAddress.getByName(args[1]); // DNS lookup
                                int serverPort = 6789;
                                DatagramPacket request =
                                                new DatagramPacket(message,  args[0].length(), aHost, serverPort);
                                aSocket.send(request);                                          //send message
                                byte[] buffer = new byte[1000];
                                DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
                                aSocket.receive(reply);                                         //wait for reply
                                System.out.println("Reply: " + new String(reply.getData()));
                                aSocket.close();
                } catch (SocketException e){ System.out.println("Socket: " + e.getMessage()); // socket creation failed
                } catch (IOException e){ System.out.println("IO: " + e.getMessage()); // can be caused by send
                }
   }
}
```

Механизм сокетов

- Java API для UDP
  - ✓ Сервер создает слушает порт 6789, принимает данные и отправляет ответ

```
import java.net.*;
import java.io.*;
public class UDPServer{
    public static void main(String args[]) {
        DatagramSocket aSocket = null;
        try{
            aSocket = new DatagramSocket(6789); // create socket at agreed port
            byte[] buffer = new byte[1000];
            while(true){
                DatagramPacket request = new DatagramPacket(buffer, buffer.length);
                aSocket.receive(request);
                DatagramPacket reply = new DatagramPacket(request.getData(), request.getLength(),
                request.getAddress(), request.getPort());
                aSocket.send(reply);
            }
        } catch (SocketException e){System.out.println("Socket: " + e.getMessage()); // socket creation failed
        } catch (IOException e) {System.out.println("IO: " + e.getMessage());
        } finally {if(aSocket != null) aSocket.close();}
    }
}
```

Механизм сокетов

- IPC основанный на TCP
  - Абстрактный сервис:поток байт принимается и получается
  - возможности
    - ✓ Размер сообщения: нет ограничений, TCP решает, когда послать сообщение транспортного уровня, состоящее из нескольких прикладных сообщений, непосредственная(немедленная) передача может принудительный
    - ✓ Ориентированный на соединение
    - ✓ Основанный на таймаутах механизм отслеживания потерянных сообщений
    - ✓ Очередь на приемщике
    - ✓ Блокирование при приеме
    - ✓ Отслеживание переполнения буферов
    - ✓ Сервер должен создавать новый поток для каждого принятого соединения
- API для потоков
  - Создание соединения
    - ✓ клиент: запрос коннекта
    - ✓ сервер: слушает порт и принимает запросы на соединение
    - ✓ Принимает соединение. Создает новый поток для соединения

Механизм сокетов

- Java API для TCP
  - Классы
    - `ServerSocket` сокет на стороне сервера
    - `Socket` класс для работы с соединением (клиент и сервер)
      - *конструктор для создания сокета и соединения с удаленным узлом и портом*
      - *Методы для работы с входными и выходными потоками*

TCP-клиент

```
import java.net.*;
import java.io.*;
public class TCPClient {
    public static void main (String args[]) {
        // arguments supply message and hostname
        Socket s = null;
        try {
            int serverPort = 7896;
            s = new Socket(args[1], serverPort);
            DataInputStream in = new DataInputStream( s.getInputStream());
            DataOutputStream out =new DataOutputStream( s.getOutputStream());
            out.writeUTF(args[0]); // UTF is a string encoding
            String data = in.readUTF(); // read a line of data from the stream
            System.out.println("Received: "+ data) ;
        } catch (UnknownHostException e) {System.out.println("Socket:"+e.getMessage()); // host cannot be resolved
        } catch (EOFException e) {System.out.println("EOF:"+e.getMessage()); // end of stream reached
        } catch (IOException e) {System.out.println("readline:"+e.getMessage()); // error in reading the stream
        } finally {if(s!=null) try {s.close();} catch (IOException e) {System.out.println("close:"+e.getMessage());}}
    }
}
```

– TCP-сервер

```
import java.net.*;
import java.io.*;
public class TCPServer {
   public static void main (String args[]) {
      try {
         int serverPort = 7896; // the server port
         ServerSocket listenSocket = new ServerSocket(serverPort); // new server port generated
         while(true) {
            Socket clientSocket = listenSocket.accept(); // listen for new connection
            Connection c = new Connection(clientSocket); // launch new thread
         }
      } catch(IOException e) { System.out.println("Listen socket:"+e.getMessage());
      }
   }
}
```

```
class Connection extends Thread {
        DataInputStream in;
        DataOutputStream out;
        Socket clientSocket;
        public Connection (Socket aClientSocket) {
                try {
                        clientSocket = aClientSocket;
                        in = new DataInputStream( clientSocket.getInputStream());
                        out = new DataOutputStream( clientSocket.getOutputStream());
                        this.start();
                } catch(IOException e){System.out.println("Connection:"+e.getMessage());
                }
        }
        public void run() { // an echo server
                try {
                        String data = in.readUTF(); // read a line of data from the stream
                        out.writeUTF(data); // write a line to the stream
                        clientSocket.close();
                } catch (EOFException e){System.out.println("EOF:"+e.getMessage());
                } catch (IOException e) {System.out.println("readline:"+e.getMessage());
                }
        }
}
```

Передача в гетерогенных системах

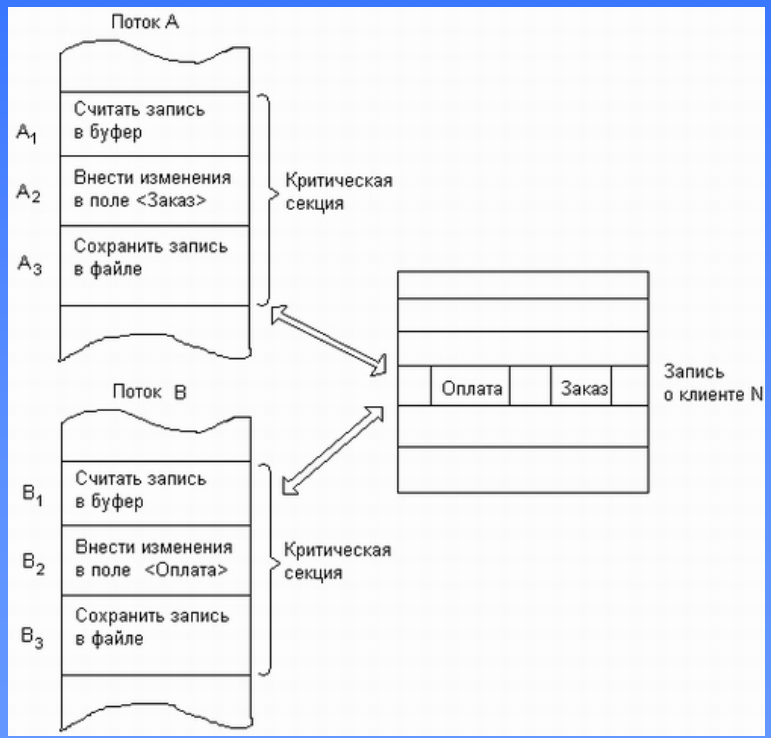
- Проблема передачи данных
  - Информация, представленная как данные определяется внутри процесса
  - Информация в сообщениях состоит только из последовательностей байтов
  - Разные платформы по разному представляют примитивные типы
    - ✓ integers (big-endian &little-endian)
    - ✓ floating-point numbers
    - ✓ characters (ASCII & Unicode)
  - » Данные должны быть упакованы перед передачей и восстановлены по прибытию
- Хорошо, если это делает слой middleware

Передача в гетерогенных системах

- Представление данных
  - Решение проблемы представления данных
    - ✓ Соглашение об использовании внешнего представления – два преобразования
    - ✓ Используется формат источника или приемщика – одно преобразование
  - Передача структурированных типов
    - ✓ Типы данных могут не изменяться при передаче
    - ✓ Использование упакованных форматов (структуры «сплющиваются»)
  - Форматы представления данных
    - ✓ SUN Microsystems XDR (eXternal Data Representation)
    - ✓ CORBA CDR (Common Data Representation)
    - ✓ ASN.1 (OSI layer 6)
  - marshalling/unmarshalling
    - ✓ marshalling: преобразование исходных данных к виду, удобному для передачи
    - ✓ unmarshalling: восстановление исходных данных
    - ✓ Обычно лежит на middleware

Параллельные потоки (процессы)

Гонка потоков (результат зависит от порядка выполнения операций)



Сценарий 1:
A1,A2,A3,B1,B2,B3
Сценарий 2:
A1,B1,B2,B3,A2,A3

Параллельные потоки (процессы)

- Условия возникновения:
  - Одновременное выполнение нескольких процессов (например, в случае многопоточного сервера, наличие двух и более одновременно выполняемых клиентских запросов)
  - Доступ к одним и тем же данным (объектам)
- Синхронизация процессов (потоков)
  - Критические секции
- Блокирование ресурсов (данных)
- Понятие транзакционности при обработке данных
  - Принципы АСИД

Критические секции

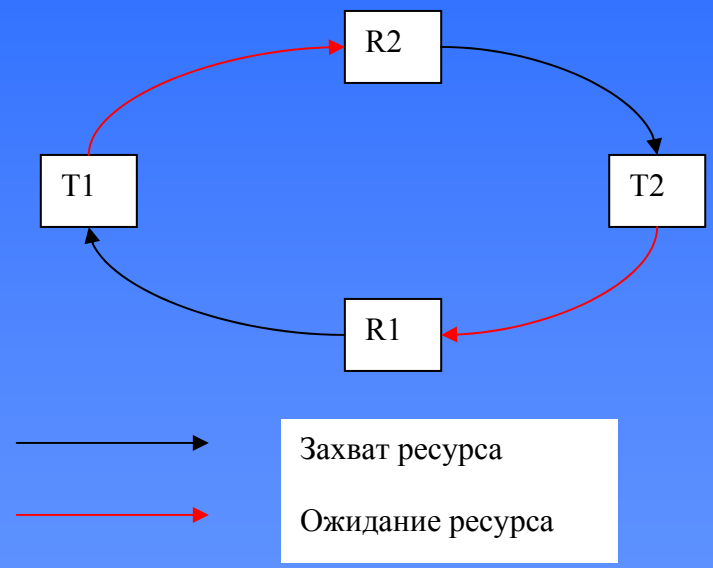
- Части кода приложения, для которых не допускается одновременное выполнение несколькими потоками
- Реализация
  - Внешние (ОС) примитивы синхронизации (напр - семафоры)
  - Внутренние примитивы синхронизации (поддержка не на уровне ОС, а на уровне прикладных библиотек)

Блокировка ресурсов

- Ресурсы, допускающие совместное использование не более чем n процессами (часто n=1), *захватываются* процессами. При успешном захвате ресурса n уменьшается на 1. При освобождении ресурса n увеличивается на 1 (семафор)
- При попытке захватить ресурс, у которого n = 0, процесс блокируется. Выход из блокировки – при увеличении n (при доступности ресурса)
- Реализация
  - Внешние (ОС) примитивы синхронизации (напр - семафоры)
  - Внутренние примитивы синхронизации (поддержка не на уровне ОС, а на уровне прикладных библиотек)

Проблема тупиков

- Возникают при использовании критических секций и блокировок ресурсов.



Проблема тупиков - решение

- Ответственность на программисте
  - Аккуратное программирование. Один из методов – соблюдение одной и той же последовательности наложения блокировок во ВСЕХ процессах.
- Ответственность на менеджере блокировок (ОС, …)
  - Обнаружение: использование графа ожидания ресурсов (пример – на предыдущем слайде). Блокировка = цикл в графе
  - Устранение:
    - ✓ Метод 1: отменить (снять, прервать) один из процессов (любой), участвующих в блокировке
    - ✓ Метод 2:отслеживается начало каждой блокировки и устанавливается ее максимальная продолжительность.

Транзакции:

- Транзакция - единая неделимая последовательность действий, представляющую собой некоторую операцию в системе и удовлетворяющая принципам АСИД:
  - **Атомарность**. Транзакции атомарны (выполняются все действия, входящие в транзакцию, или ни одно из них)
  - **Согласованность**. Транзакции защищают данные согласованно. Это означает, что транзакции переводят одно согласованное состояние системы в другое без обязательной поддержки согласованности во всех промежуточных точках.
  - **Изоляция**. Транзакции отделены одна от другой (не оказывают друг на друга влияния).
  - **Долговечность**. Когда транзакция выполнена, ее обновления сохраняются, даже если в следующий момент произойдет сбой системы.

Транзакции: типичная семантика

- Типичная реализация механизма транзакций обеспечивает, по меньшей мере, следующие операции:
  - операция начала транзакции (beginTransaction)
  - операция фиксации транзакции (commit)
  - операция отмены транзакции (rollback)
- для реализации предложенного подхода необходимо наличие координатора (менеджера) транзакций – специального компонента, обеспечивающего управление транзакциями. Типичным является размещение этого компонента на стороне сервера.

Транзакции: примеры использования

- Банковский перевод
- Перевод представляет собой согласованную последовательность ДВУХ действий
  - Снять сумму со счета – источника
  - Положить сумму на счет – приемник
- Важные замечания:
  - В случае ошибки при выполнении ЛЮБОЙ операции из этой последовательности, должны быть отменены (откат транзакции) все выполненные ранее операции (понятие обратимости операции)
  - Откат транзакции может быть вызван как нарушением логики системы (счет – приемник закрыт), так и отказом (в том числе - оборудования).
  - Два состояния – перед началом транзакции и после ее окончания являются согласованными. Состояние «в процессе выполнения» транзакции является не консистентным для системы.

Транзакции: примеры использования

- Процедура подсчета суммы средств на счетах:

	t1	t2
Иванов	120	20
Петров	150	150
Сидоров	240	240
…		
…		
Яковлев	400	500
…		

- Вывод – с введением понятия транзакции исчезают старые проблемы и появляются новые

Транзакции: возможные проблемы

Проблема потери результатов обновления

№ по порядку	Транзакция 1	Транзакция 2
1	beginTransaction	beginTransaction
2	B:= Иванов.ballance	
3		B:= Иванов.ballance
4	B:=B+100;	B:=B+200;
5	Иванов.ballance:=B;	
6	Commit;	Иванов.ballance:=B;
7		Commit;

В результате выполнения этого сценария, изменения, сделанные первой транзакцией оказались потерянными

Проблема незафиксированной зависимости

№ по порядку	Транзакция 1	Транзакция 2
1	beginTransaction	beginTransaction
2		Иванов.ballance+=100;
3	B:= Иванов.ballance	
4	B:=B+100;	
5	Иванов.ballance:=B;	
6	Commit;	
7		Rollback;

В результате выполнения этого сценария, транзакция1 внесла изменения, рассчитанные на основании данных, которых «не существует» (транзакция2, которая эти данные внесла, отменила свои изменения)

Проблема несовместимого анализа

	t1	t2
Иванов	120	20
Петров	150	150
Сидоров	240	240
…		
…		
Яковлев	400	500
…		

Проблема несовместимого анализа возникает в том случае, если одна из транзакций в силу длительности выполнения застает часть данных в одном состоянии (неизмененными), а часть в другом (уже измененными).

Транзакции: уровни изоляции

- Уровень изоляции транзакций определяет влияние, которое оказывают друг на друга параллельно выполняющиеся транзакции (имеет смысл только в случае параллельно выполняющихся процессов). Уровень изоляции показывает, какие из «типичных» проблем возможны при его использовании, а какие – нет.

Основные проблемы, возникающие при параллельной обработке данных следующие:

- проблема потери результатов обновления
- проблема незафиксированной зависимости
- проблема несовместимого анализа.

Транзакции: уровни изоляции

- Уровни изоляции транзакций определены относительно некоторых эффектов параллельной обработки: Уровень изоляции определяет возможность возникновения этих эффектов

Уровень изоляции	Грязное чтение	Неповторяемое считывание	Фиктивные элементы
Грязное чтение (Dirty read)	да	да	да
Завершенное считывание (Read commited)	нет	да	да
Повторяемое чтение (Repeatable read)	нет	нет	да
Способность к упорядочиванию (Serializable)	нет	нет	нет

Транзакции: уровни изоляции

- **Неаккуратное (грязное) считывание**. Допустим транзакция T1 выполняет обновление с некоторым объектом, затем транзакция T2 извлекает состояние этого объекта, после чего выполнение T1 отменяется. В результате транзакция T2 обнаружит, что извлеченное ей состояние объекта никогда не существовало (поскольку не зафиксирована T1)
- **Неповторяемое считывание**. Допустим, транзакция T1 извлекает состояние объекта, транзакция T2 изменяет это состояние и фиксирует транзакцию, после чего T1 вновь извлекает состояние этого же объекта. Получается, что один и тот же объект для T2 в разные моменты времени будет иметь разные состояния.
- **Фиктивные элементы**. Допустим, что транзакция T1 извлекает какое-то множество элементов (процесс продолжителен во времени), транзакция T2 добавила в это множество элемент и зафиксировала транзакцию. T1, продолжая извлекать элементы, обнаружит элемент, которого раньше во множестве не было.

Транзакции: откат

- **Важнейшая задача транзакционной обработки – атомарность – предполагает возможность отката сделанных изменений в случае, если какое то из действий, составляющих транзакцию, закончилось неудачей.**
- Возможные реализации:
  - изменения, которые делает транзакция в процессе работы, записываются не в долговременную память, а во временную. в случае фиксации транзакции происходит перенос данных из временной памяти в основную, а в случае отката временная память просто очищается
  - *журналирование* операций, выполняемых транзакцией. Все изменения, которые производит транзакция записываются в журнал операций. В случае отката транзакций часть журнала, относящегося к прерванной транзакции, может быть просто уничтожена (часто в этом случае в журнал операций записывается специальная операция отката транзакции), а в случае фиксации транзакции – операции из журнала должны быть «проиграны» на основной долговременной памяти.

Вложенные и распределенные транзакции

- Транзакции могут затрагивать «не локальные системы» (несколько таких систем) – быть рапределенными
- Транзакции могут включать в себя другие транзакции (образовывать деревья транзакций) – содержать вложенные транзакции
- Пример: бронирование тура
  - Поиск подходящего отеля и бронирование в нем места
  - Поиск подходящего авиарейса и бронирования места на нем
  - Поиск подходящего обратного авиарейса и бронирование места на нем
  - Заказ необходимых дополнительных услуг (экскурсий, и т.д.)
  - Оплата через банк всего пакета услуг (проживание, перелет, экскурсии)
- Вовлечены несколько независимых систем
- Каждая из операций – может быть транзакцией в терминах своей системы

Поддержка распределенных транзакций

- Двух-фазный протокол фиксации транзакций (2PC)
  - В системе выделяется менеджер транзакций (управляющий компонент; во многих реализациях он называется координатором транзакций)
  - Фиксация изменений транзакции происходит в два этапа. На первом этапе изменения верифицируются. Производятся все необходимые проверки, которые гарантируют, что внесенные изменения полностью готовы к переносу в долговременную память (некоторые реализации уже на этой фазе переносят изменения в долговременную память, помечая их как «незафиксированные»). Вторая фаза начинается только после того, как все узлы, вовлеченные в транзакцию, отрапортуют координатору об успешном завершении первой фазы. Вторая фаза состоит в простом переносе уже подготовленных изменений в долговременную память

Протокол координатора (пример)

- Фаза 1
  - Координатор посылает сообщение CanCommit на все узлы, вовлеченные в транзакцию
  - Когда узел получает сообщение CanCommit он отвечает координатору Yes или No. Перед тем, как ответить Yes, узел выполняет подготовку к фиксации изменений, записывая объекты в долговременной памяти. В случае ответа No (или в том случае, когда при подготовке ответа Yes произошла ошибка) узел немедленно откатывает свои изменения.
- Фаза 2
  - Координатор собирает ответы от всех узлов
    - ✓ Если все узлы ответили Yes, координатор фиксирует транзакцию и рассылает всем узлам сообщения doCommit
    - ✓ В противном случае, координатор откатывает транзакцию, рассылая всем узлам сообщение doAbort
  - Узлы, которые ответили Yes на запрос CanCommit ожидают сообщения doCommit или doAbort от координатора. При получении одного из этих сообщений, узел выполняет фиксацию или откат своих изменений, после чего пересылает координатору сообщение HaveEnded

Блокировки в распределенных системах

- В распределенных системах блокировки являются механизмом борьбы с нежелательными эффектами параллельной обработки.
- Для их поддержки часть вводится диспетчер блокировок, который управляет блокировками на уровне системы.
- Разрешение тупиков может быть реализовано методом анализа графа ожидания ресурсов и применением одного из рассмотренных ранее алгоритмов

Репликация данных

- Требование повышения производительности
- Требование повышения надежности системы путем резервного копирования части критичных данных

Пассивная (primary - backup) репликация

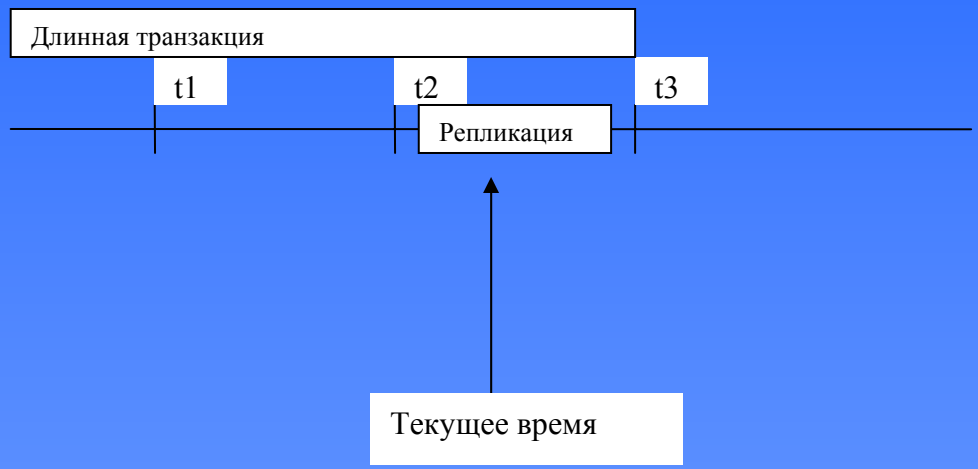
- Описание: изменение данных допускается только на одном узле (данные, размещенные на этом узле, называются мастер - копией) а на всех других узлах данные доступны только для чтения
- Применение: может использоваться как для резервного копирования данных, так и в большинстве информационных систем, в которых операции изменения данных сравнительно редки по отношению к операциям извлечения данных (при этом извлекаются данные большого объема) и могут быть сведены к операциям над мастер - копией

Пассивная репликация: реализация

- Весь массив данных по некоторому регламенту копируется на удаленные узлы. При этом подходе предполагается, что данные изменяются нечасто и на удаленных узлах не важна их высокая актуальность
- Второй подход называется репликацией на временных метках (timestamp - replication). Он состоит в том, что любая транзакция, изменяющая данные, изменяет также и временную метку (timestamp) этих данных. Таким образом, у данных, которые изменялись позднее, будет большая временная метка, чем у данных, которые изменялись раньше. Зная время последнего удачного обмена данными, менеджер репликаций «снимает» все данные, измененные позднее и передает на удаленный узел только их .
- Третий подход: перенос не данных, а журналов изменений

Пассивная репликация: замечания

- Процесс, «снимающий» измененные данные **должен** работать на уровне изоляции Serializable



  - Нужны механизмы борьбы

Активная репликация с блокировкой

- Описание: главное отличие от пассивной репликации состоит в том, что более чем один узел системы может претендовать на то, чтобы изменить объект.
- Реализация: предполагается, что прежде чем изменять объект, узел запрашивает разрешение на это у центрального координатора (фактически это является блокировкой объекта, поскольку как только такое разрешение получено, остальные запросы на этот же объект координатором будут отвергаться). После того, как объект изменен, это изменение реплицируется на все узлы, которые потенциально могут изменять этот объект и только после этого блокировка объекта снимается

Активная репликация без блокировки

- Описание: рассматривается система, в которой несколько узлов могут одновременно выполнять операции над одним и тем же объектом, не блокируя его и не дожидаясь репликаций о других изменениях
- Ограничения: При применении такого метода возможны разнообразные эффекты типа «потерянного обновления» и т.д. Возможен эффект невозможности приема репликаций - например, мы разрешаем ведение справочника покупателей на двух разных узлах, при этом накладываем ограничение, что название покупателя должно быть уникально в пределах справочника.
- Применение: системы, где нет редактирования сущностей – только добавление (большинство банковских платежных систем).