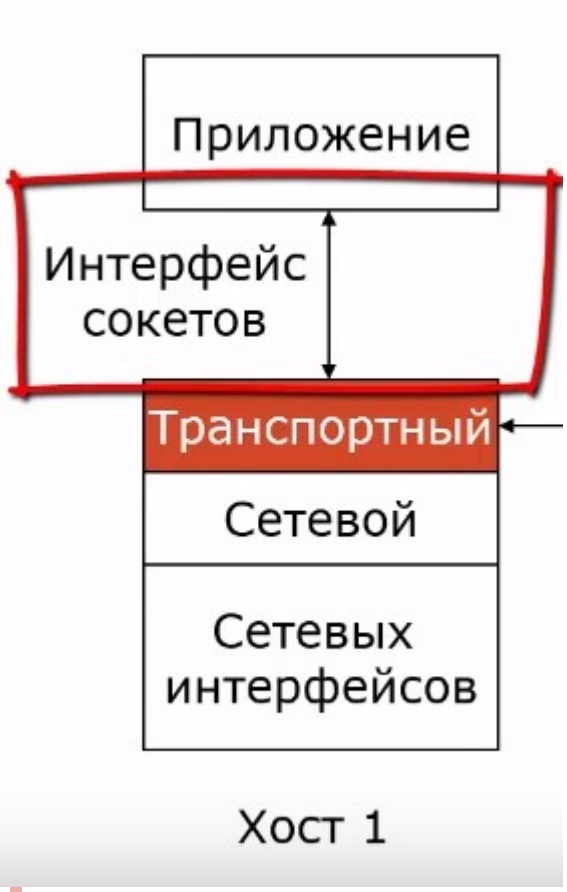
КОММУНИКАЦИИ ОРИЕНТИРОВАННЫЕ НА ПЕРЕДАЧУ СООБЩЕНИЙ (MESSAGE ORIENTED COMMUNICATION)

MESSAGE ORIENTED COMMUNICATION

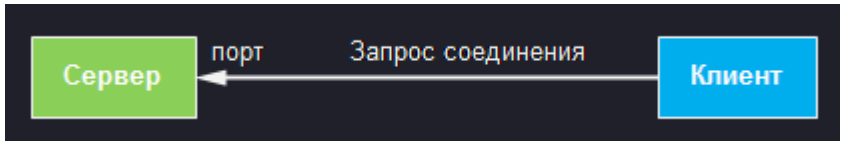
- Несмотря на то, что RPC и RMI поддерживают прозрачность доступа, однако не всегда это делается походящим способом.
- Например:
  - когда принимающая сторона в момент выдачи запроса клиентом, выполняет необходимы альтернативные услуги связи, связанные с запросом от другого клиента, что приводит к увеличению времени выполнения, а то и к потере запроса.
  - внутренняя синхронная природа RPC, из-за которой клиент блокируется до тех пор, пока его запрос не будет обработан, может потребоваться заменить чем-то другим.
- Такой заменой могут быть коммуникации ориентированные на обмен сообщениями, которые обладают большей гибкостью.
- Эти коммуникации основываются на уровне транспортных протоколов (транспортный уровень OSI.)
- Стандартизованными интерфейсами к транспортному уровню OSI являются **сокеты**:
  - (Berkeley UNIX);
  - XTI (X/Open Transport Interface), ранее известные как TLI (AT&T model) – двойник сокетов Berkeley.

ПРОСТОЙ НЕРЕЗИДЕНТНЫЙ ОБМЕН СООБЩЕНИЯМИ С ИСПОЛЬЗОВАНИЕМ СОКЕТОВ

СОКЕТЫ



- Сокеты впервые появились в операционной системе (ОС) Berkeley Unix 4.2 BSD в 1983 году.
- Концептуально **сокет** - это конечная точка связи, в которую приложение может записывать данные, которые должны быть отправлены по базовой сети, и из которой могут быть прочитаны входящие данные.
- Сокет **образует абстракцию** над фактическим **портом,** который используется локальной операционной системой для определенного **транспортного протокола**.
- Связь между конечными точками используется приложениями для записи/чтения в/из сети.
- Сокеты являются абстракцией реальных коммуникаций с конечными точками обслуживания средствами локальной ОС.
- Адрес сокета: Адрес IP + номер порта



ОПЕРАЦИИ СОКЕТОВ БЕРКЛИ

- Механизм сокетов предоставляет набор базовых примитивов (функций) для выполнения операций по передаче и приему данных в/из конечных точек.

Примитив (функция)	**Значение**
socket ()	Создать новую конечную точку связи
bind ()	Привязать локальный адрес к сокету
listen ()*	Готовность принимать соединения (неблокирующая)
accept ()	Блокировать вызывающего абонента до тех пор, пока не поступит запрос на подключение
connect ()	Активная попытка установить соединение
send ()	Отправить данные по соединению
receive ()	Получить данные по соединению
close ()	Закрыть соединение

ОПЕРАЦИИ СОКЕТОВ БЕРКЛИ

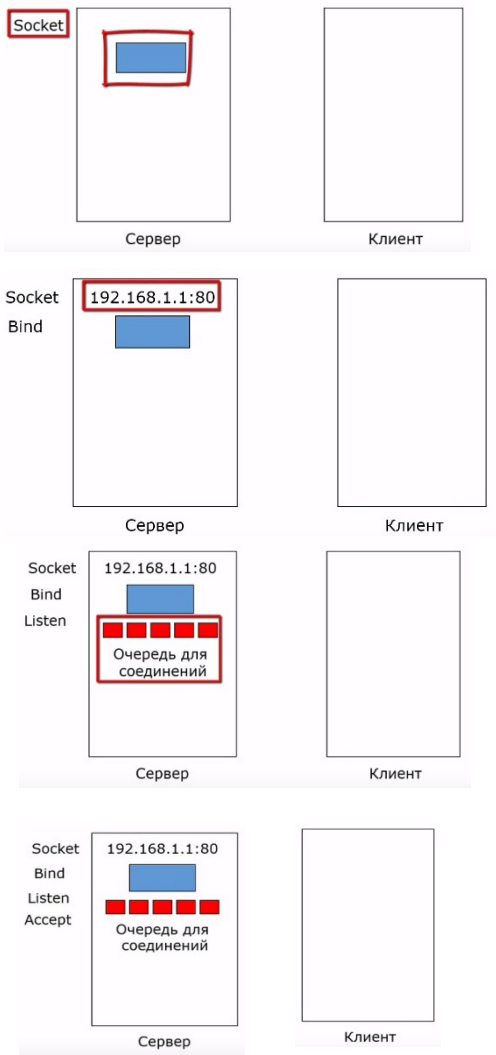
- Операции сокетов Беркли делятся на несколько типов.
  - Первый тип это создание сокетов: **Socket, Bind, Listen**.
  - Второй, установка соединения: **Connect и Accept**.
  - Третья, передача данных **Send, Receive.**
  - Четвёртое, закрытие соединения **Close.**

КОММУНИКАЦИИ НА ОСНОВЕ СОКЕТОВ

- Используя сокеты, **клиенты и серверы могут устанавливать между собой связь** в рамках сессии ориентированной на соединение (connection-oriented communication session).
- **На серверах** используются первые четыре функции (**socket, bind, listen, accept**), в то время как на **клиентах** используются примитивы (функции) – **socket и connect**.
- Затем на клиенте и сервере поочередно выполняются операции записи-чтения (**client/write, server/read, server/write, client/read**) а в случае закрытия соединения операция close, закрывающая соединение.

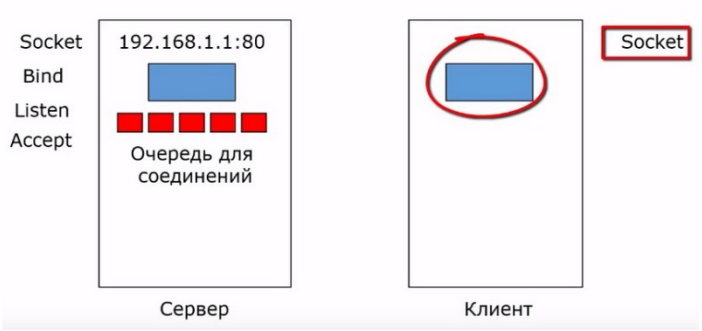
РАБОТА СОКЕТОВ В РАМКАХ МОДЕЛИ КЛИЕНТ-СЕРВЕР
СЕРВЕР.

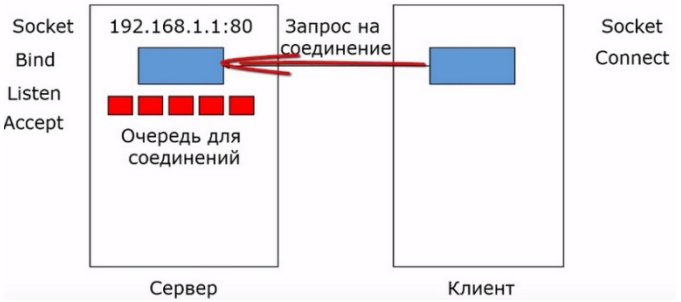
1. На сервере выполняется вызов **soket**. Создается объект — сокет, в простейшем случае, это просто файл специального вида:
2. Затем вызывается метод **bind**, который используется для присоединения сокета к определенному ip адресу и порту:
3. Вызов **listen** говорит о том, что сокет готов принимать соединение по сети, сокет слушает. При вызове listen создаётся очередь для соединений, в вызове необходимо указать размер этой очереди:
4. Затем сервер вызывает метод сокета **accept**, это говорит о том, что сервер готов принимать соединения и он переходит в режим пассивного ожидания, ждет установку запросов на соединение от клиентов:

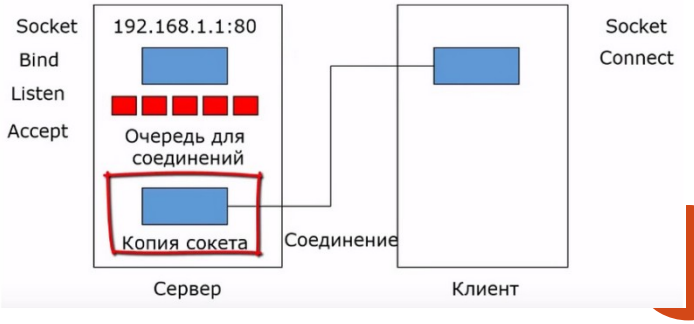


РАБОТА СОКЕТОВ В РАМКАХ МОДЕЛИ КЛИЕНТ-СЕРВЕР. КЛИЕНТ.

1. Клиент со свой стороны, сначала вызывает метод **socket**, для создания сокета, как правило для клиента не имеет значение, какой ip адрес и какой порт используется, номер порта назначается операционной системой. Поэтому **метод bind** на клиентском сокете обычно **не вызывается**.
2. Сразу после создания сокета, вызывается метод **connect**, в котором указывается ip адрес и порт. В параметрах метода connect указываются ip адрес сервера и порт с которыми нужно установить соединение. Отправляется запрос на соединение.
3. Для того, чтобы другие клиенты могли соединяться с этим сервером на этом ip адресе и на этом же порту, **создаётся копия сокета**. И соединение устанавливается не с исходным сокетом, который принимает соединения, а с копией сокета. Данные передаются через копию сокета.

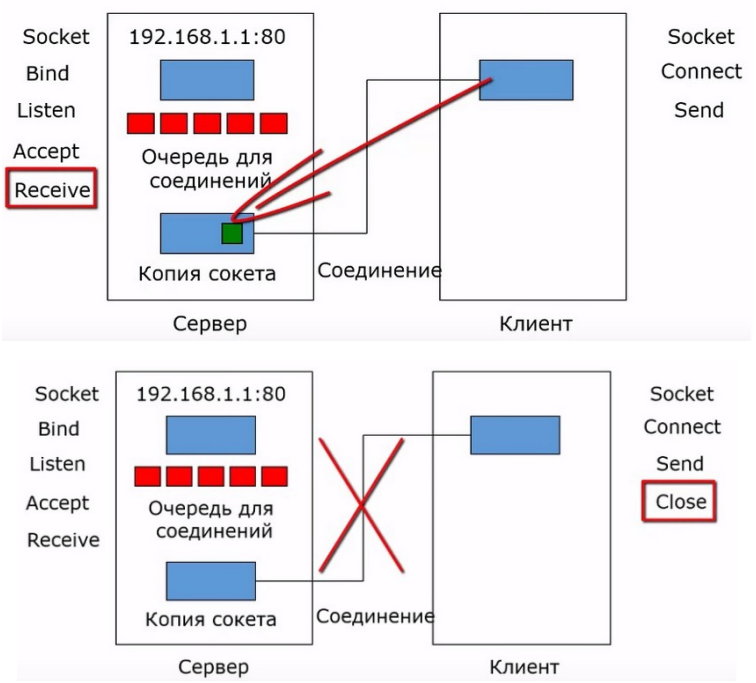






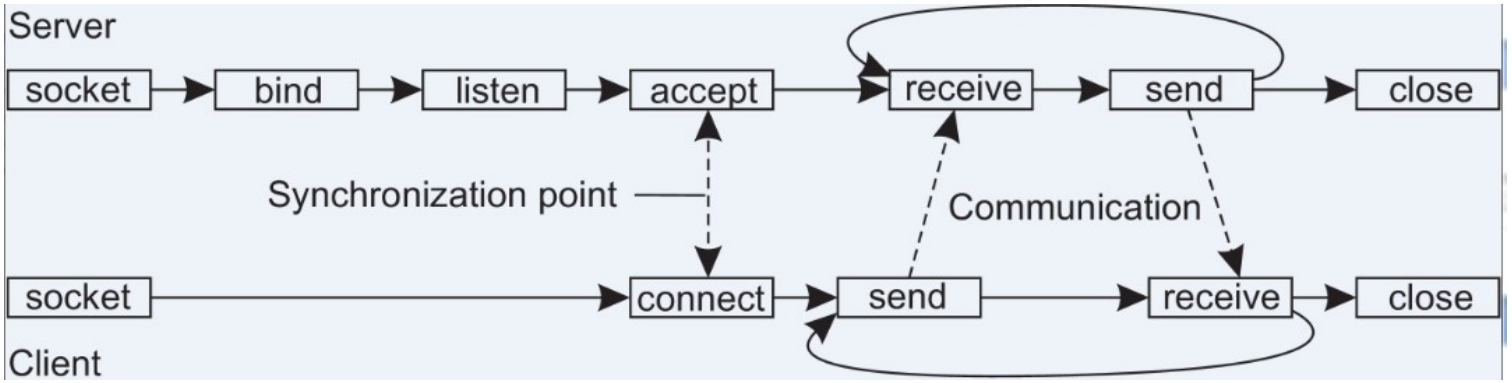
РАБОТА СОКЕТОВ В РАМКАХ МОДЕЛИ КЛИЕНТ-СЕРВЕР. ОБМЕН СООБЩЕНИЯМИ И ЗАКРЫТИЕ СОЕДИНЕНИЯ.

1. Клиент подготавливает порцию данных, вызывает метод **send.** Данные передаются по сети и сервер может их прочитать с помощью метода **receive**. Дальше сервер и клиент могут обмениваться между собой несколькими порциями данных.
2. После того, как все данные переданы, клиент вызывает метод close. После чего происходит разрыв соединения.



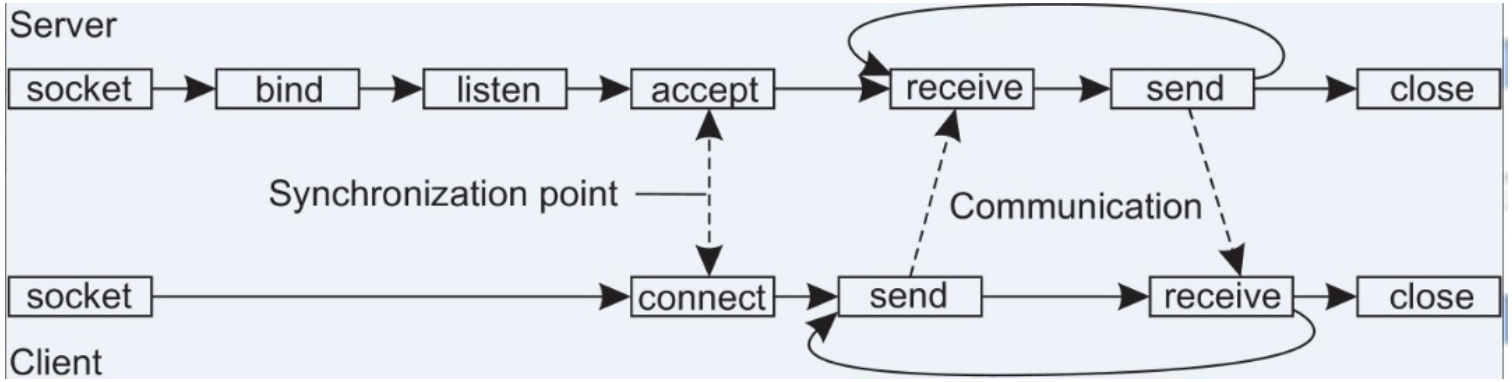
Общая схема взаимодействия клиента и сервера с использованием сокетов. Сервер.

- Общая схема взаимодействия клиента и сервера с использованием сокетов для коммуникаций, ориентированных на соединение:
- **Сервер:**
  - Примитив **bind** выполняет привязку локального адреса к только что **созданному сокету**. Например, сервер должен связать IP-адрес своей машины с номером порта (возможно, общеизвестным) сокета. Привязка сообщает операционной системе, что сервер намерен получать сообщения только на указанные адрес и порт.
  - Примитив **listen** применяется только для коммуникаций, ориентированных на соединение. Это **неблокирующий вызов**, требующий от локальной операционной системы зарезервировать буфер для определенного максимального количества соединений, которое вызывающий процесс намерен поддерживать.
  - Вызов примитива *accept* блокирует вызывающий процесс до прихода запроса на соединение. Когда этот запрос придет, локальная операционная система создаст новый сокет с теми же свойствами, что и у базового, и возвратит его вызывающему процессу.



ОБЩАЯ СХЕМА ВЗАИМОДЕЙСТВИЯ КЛИЕНТА И СЕРВЕРА С ИСПОЛЬЗОВАНИЕМ СОКЕТОВ. КЛИЕНТ.

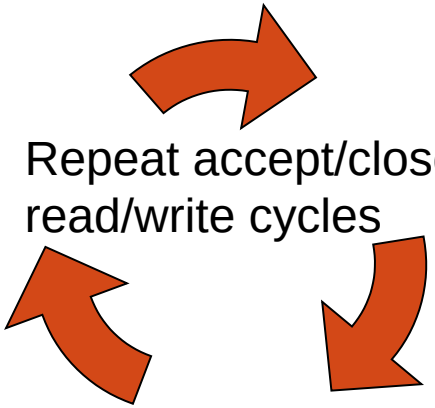
- **На стороне клиента** все начинается с создания сокета при помощи примитива socket, однако в явной привязке сокета к локальному адресу нет необходимости, поскольку операционная система может динамически выделить порт при установлении соединения.
- Примитив **connect** требует, чтобы вызывающий процесс указал **адрес транспортного уровня**, на который будет отправлен запрос на соединение.
- **Клиент блокируется** до тех пор, **пока соединение не будет установлено**.
- После установления соединения стороны начинают обмениваться информацией при помощи примитивов **write** и **read,** предназначенных для посылки и приема данных соответственно.
- Наконец, **закрытие** соединения при использовании сокетов симметрично и может быть осуществлено как клиентом, так и сервером путем вызова примитива **close.**



Использование сокетов сервером (How a Server Uses Sockets)

Internetworking with TCP/IP, Douglas E. Comer & David L. Stevens, Prentice Hall, 1996

Системный вызов:	Что означает:
• Socket	• Создать дескриптор сокета.
• Bind	• Привязать локальный IP адрес/порт к сокету.
• Listen	• Перейти в пассивный режим, создать очередь запросов.
• Accept	• Получить следующее сообщение
• Read	• Читать данные поступающие из сети
• Write	• Писать данные в сеть.
• Close	• Разорвать соединение

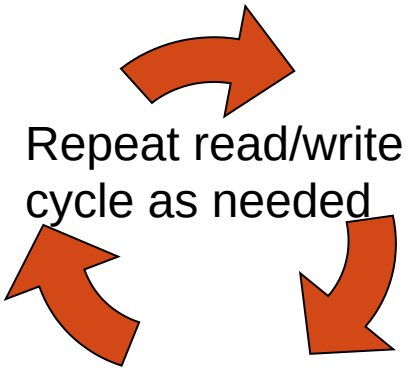


Использование сокетов клиентом
How a Client Uses Sockets

Internetworking with TCP/IP, Douglas E. Comer & David L. Stevens, Prentice Hall, 1996

Системный вызов:

- Socket
- Connect
- Write
- Read
- Close



Что означает:

- Создать дескриптор сокета.
- Подключиться к удаленному серверу
- Писать данные в сеть.
- Читать данные поступающие из сети
- Разорвать соединение

РАСШИРЕННЫЙ НЕТРАНЗИТНЫЙ ОБМЕН СООБЩЕНИЯМИ

- ZeroMQ
- Message-Passing Interface (MPI)

НЕДОСТАТКИ СТАНДАРТНОГО МЕХАНИЗМА СОКЕТОВ

- Стандартный подход на основе сокетов к реализации нетранзитному обмену сообщениям очень прост и, как таковой, довольно хрупок.
  - **Легко сделать ошибку**.
  - Более того, сокеты по существу поддерживают **только TCP или UDP**, а это означает, что любые дополнительные средства для обмена сообщениями должны быть реализованы отдельно программистом приложения.
- На практике нам часто требуются более продвинутые подходы к коммуникациям, ориентированным на сообщения, с целью:
  - **упростить** сетевое программирование,
  - **расширить** функциональные возможности, предлагаемые существующими сетевыми протоколами,
  - лучше **использовать локальные ресурсы** и т. д.
- Один из подходов к **упрощению сетевого программирования** основан на наблюдении, что многие приложения для обмена сообщениями или их компоненты могут быть эффективно организованы в соответствии с несколькими простыми **шаблонами связи.** Посредством последующего усовершенствования сокетов для каждого из этих шаблонов можно упростить разработку сетевого распределенного приложения.

ИСПОЛЬЗОВАНИЕ ШАБЛОНОВ ОБМЕНА СООБЩЕНИЯМИ: ZEROMQ

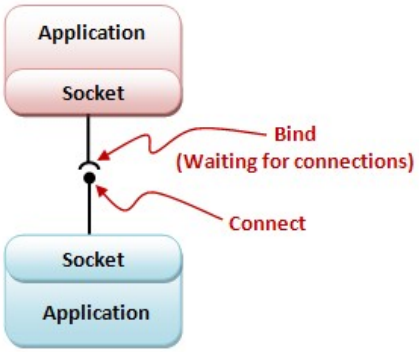
- Как и в случае сокетов Беркли, ZeroMQ также предоставляет **сокеты**, через которые происходит вся связь.
- Фактическая передача сообщений обычно происходит **через TCP-соединения**, и вся связь по существу ориентирована на соединение. Однако установка и поддержание соединений в основном скрыта ОС: прикладному программисту не нужно беспокоиться об этих проблемах.
- Чтобы еще больше упростить ситуацию, сокет ZeroMQ может быть привязан к **нескольким адресам**, что позволяет серверу обрабатывать сообщения из самых разных источников через единый интерфейс.
  - Например, сервер может прослушивать несколько портов, используя одну операцию приема с блокировкой. Таким образом, сокеты ZeroMQ могут поддерживать связь **«многие к одному»,** а **не только «один к одному»,** как в случае со стандартными сокетами Беркли.
- Сокеты ZeroMQ также поддерживают связь «**один-ко-многим»**, то есть многоадресную (групповую) рассылку (multicast).
- Существенным для ZeroMQ является то, что связь является **асинхронной**:
  - **отправитель** обычно **продолжает работу после отправки сообщения** в базовую подсистему связи.
- Сочетание **асинхронной** связи со связью, **ориентированной на соединение**, является то, что процесс может запросить настройку соединения и впоследствии отправить сообщение, **даже если получатель еще не запущен** и не готов принимать входящие запросы на соединение.

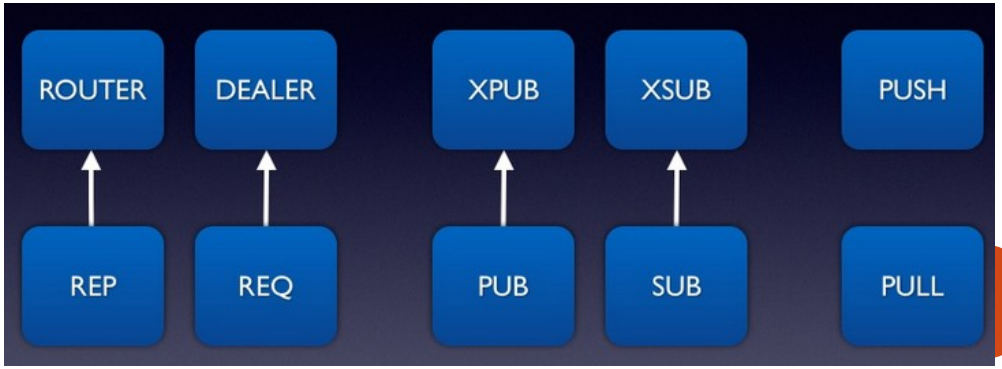
О БИБЛИОТЕКЕ ZEROMQ

- ZeroMQ (также обозначаемый как ØMQ, 0MQ или ZMQ) – это программное обеспечение с **открытым** исходным кодом **для организации очередей сообщений**.
- ZeroMQ не является полноценной системой организации очередей как, например RabbitMQ или ActiveMQ. ZeroMQ - всего лишь достаточно простой программный интерфейс, позволяющий создать свою собственную MQ-систему.
- Но работает ZeroMQ как **фреймворк для многопоточного** программирования. Она дает вам сокеты, которые **атомарно** передают сообщения через **различные транспорты** внутри процесса, между процессами, по TCP или мультикастом.
- У ZeroMQ **нет автономного** сервера; сообщения отправляются напрямую от приложения к приложению.
- ZerpMQ проста в изучении и внедрении. Он состоит из одной библиотеки под названием **libzmq.dll**, написанной на C ++, которую можно связать с любым приложением.
- Чтобы использовать его в среде .NET, нам нужна оболочка для этой библиотеки, которая называется **clrzmq.dll**, написанная на C #.
- ZeroMQ можно запустить в Windows, OS X и Linux. Для реализации приложений с использованием ZeroMQ можно использовать несколько языков, включая C, C ++, C #, Java, Python…(всего **15 языков** программирования). Это дает возможность взаимодействовать с различными приложениями на разных платформах.

СОКЕТЫ ZEROMQ

- ZeroMQ сокет - это не традиционный сокет, но это сокет, который обеспечивает уровень абстракции **поверх традиционного API сокетов**, что освобождает от сложности и повторяющихся задач, которые характерны при использовании традиционные сокеты.
- ZeroMQ поддерживает несколько типов сокетов (тип сокета определяется как **значение атрибута** в самом сокете), например:
  - **REQ, REP, ROUTER, DEALER, PUB, SUB**, и т.д. От типа сокета на обоих концах зависит его поведение.
- Различные комбинации типов сокетов на отправляющей и принимающей сторонах позволяют реализовать разные шаблоны связи:
  - Конкретный тип сокета, используемый для отправки сообщений, соединяется с соответствующим типом сокета для получения сообщений.
  - Каждая пара типов сокетов соответствует шаблону связи.





АСИНХРОННАЯ СВЯЗЬ ZEROMQ

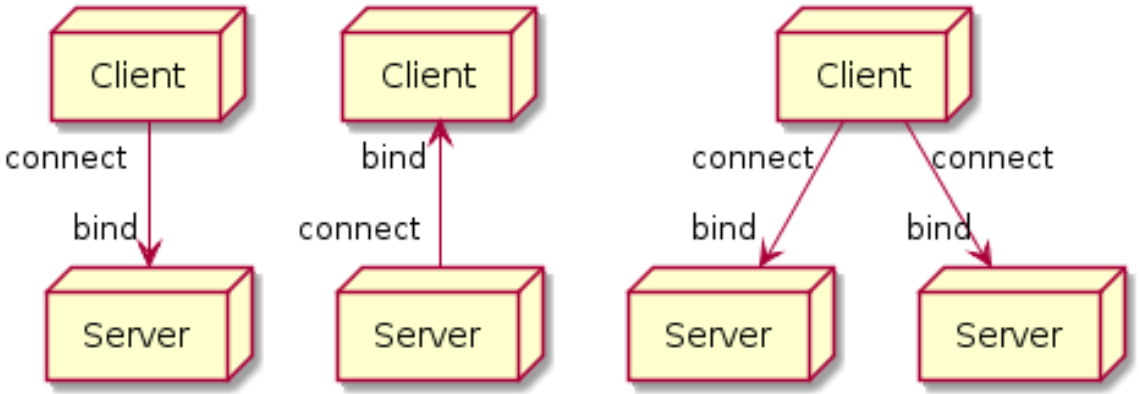
- Связь, осуществляемая ZeroMQ, **осуществляется асинхронно**. Это означает, что наше **приложение не будет заблокировано** во время установки или закрытия сокетного соединения, повторного подключения и доставки сообщений.
- Эти операции управляются самим ZeroMQ в **фоновых потоках** и параллельно с обычной обработкой, выполняемой нашим приложением.
- При необходимости он **автоматически ставит сообщения в очередь** (на стороне отправителя или получателя). Он делает это разумно, проталкивая сообщения как можно ближе к получателю перед их постановкой в очередь.

ТРАНСПОРТНЫЕ ПРОТОКОЛЫ ZEROMQ

- ZeroMQ поддерживает 4 типа транспортных протоколов. Каждый транспорт определяется адресной строкой, состоящей из двух частей:
  - `transport://endpoint.`
- Транспортная часть определяет базовый транспортный протокол для использования, а часть конечной точки определяется в соответствии с используемым протоколом следующим образом:
  - **TCP** (`tcp://hostname:port`): стандартная однонаправленная сетевая передача данных с использованием TCP-протокола;
  - **INROC** (`inproc://name`): передача сообщений внутри процесса, между потоками
  - **IPC** (`ipc:///tmp/filename`): передача сообщений между процессами в пределах одного хоста
  - **MULTICAST** – групповая (многоадресная) сетевая передача сообщений с гарантированной доставкой, реализованная посредством инкапсуляции прикладных данных непосредственно в IP-пакет (`pgm://interface;address:port`) или с использованием стандартного UDP-протокола (`epgm://interface;address:port`).

ВИДЫ ПОДКЛЮЧЕНИЙ В ZEROMQ

- Направление подключения не зависит от роли компонент.
- В TCP как правило клиент подключается к серверу: **сервер слушает** порт, а **клиент подключается** к нему.
- В ZMQ можно все наоборот:
  - клиент слушает порт, а сервер подключается к нему.
- К тому же можно подключаться сразу к нескольким слушающим сокетам, тогда сообщения будут распределены между ними, чаще всего поочередно (round-robin).
- Поэтому для слушания **выбираем не тот компонент**, который **сервер**, а тот, чей **сетевой адрес будет реже меняться** или будет более известен.

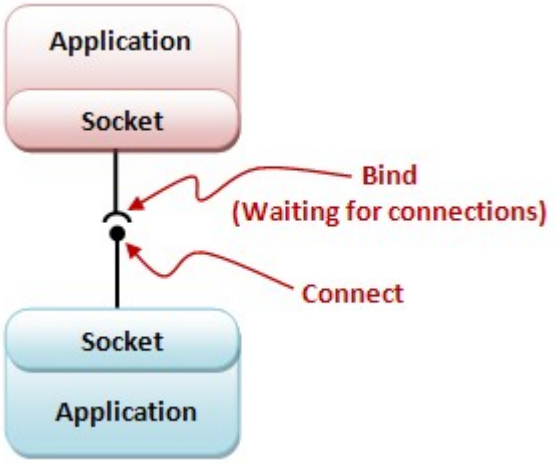


ОТЛИЧИЯ СОКЕТОВ ZEROMQ ОТ TCP СОКЕТОВ

Признак различий	TCP сокеты	ZeroMQ сокеты
Направление связи	Сервер слушает порт, а клиент подключается к нему	1. Можно все наоборот: клиент слушает порт, а сервер подключается к нему. 2. Можно подключаться сразу к нескольким слушающим сокетам, тогда сообщения будут распределены между ними (обычно по кругу (round-robin). 3. Можно «подключиться» и начать посылать сообщения, когда TCP связности между узлами еще нет. (благодаря использованию очередей на обеих сторонах).
Формат обмена	TCP «доставляет» потоки.	1. ZMQ доставляет сообщения. Сообщения — просто наборы байт. 2. Гарантированной доставки нет. О гарантированной доставке нужно заботиться самостоятельно с помощью нумерации сообщений, дополнительных запросов, таймаутов и т.д и т.п.
Наличие разделителей фрагментов сообщений	В HTTP (работает поверх TCP) для разделения заголовков от тела запроса используется пустая строка	1. В ØMQ для подобного разделения не нужно изобретать специальных разделителей, достаточно использовать фрагменты. 2. Фрагменты — это просто удобный способ разбиения сообщения на части.

ШАБЛОНЫ

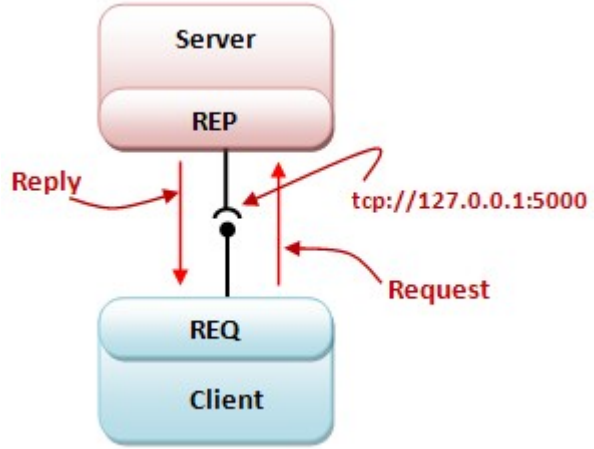
- Шаблон связи - это набор подключенных сокетов, которые определяют поток сообщений.
- Три наиболее важных шаблона коммуникации, поддерживаемые ZeroMQ:
  - запрос-ответ,
  - публикация-подписка и
  - конвейер.



ШАБЛОН «ЗАПРОС-ОТВЕТ»

- Этот шаблон используется в традиционном взаимодействии клиент-сервер, как и те, которые обычно используются для удаленных вызовов процедур.
- Клиентское приложение использует **сокет запроса** (типа **REQ**) для отправки сообщения запроса на сервер и ожидает, что последний ответит соответствующим ответом. Предполагается, что сервер использует **сокет ответа** (типа **REP**).
  - **Клиент отправляет** запрос и **получает** ответ, в то время как сервер получает запрос и отправляет ответ.
  - Это позволяет **одному клиенту** подключаться **к одному или нескольким сер**верам. В этом случае запросы распределяются по **циклическому алгоритму** между всеми серверами (Reps), один запрос отправляется на один сервер, а следующий запрос отправляется на следующий сервер и так далее.
  - Шаблон **на основе состояния**: клиент должен **получить ответ** на свой запрос перед отправкой другого, а сервер должен **отправить ответ** перед получением другого запроса.
- Шаблон «запрос-ответ» упрощает задачу для разработчиков, **избавляя от необходимости** вызывать операцию **прослушивания** (**listen** в сокетах Беркли), а также операцию **принятия** (**accept** в сокетах Беркли).

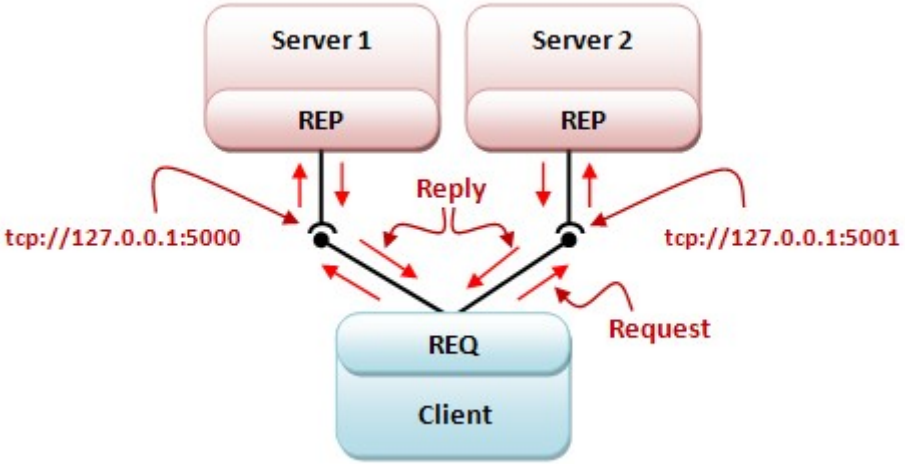
1. ОДИН КЛИЕНТ – ОДИН СЕРВЕР



- Асинхронный характер ZeroMQ позволяет запускать клиента перед запуском сервера.
- Более того, **когда сервер получает сообщение**, последующий **вызов** для отправки **автоматически нацелен на исходного отправителя**.
- Аналогичным образом, **когда клиент вызывает операцию recv** (для получения сообщения) **после отправки сообщения**, ZeroMQ предполагает, что клиент **ожидает ответа от исходного получателя**.
- Давайте рассмотрим этот шаблон, используя два случая соединений клиент-сервер.

Рассмотрим шаблон «запрос-ответ», используя два случая соединений клиент-сервер.

2. ОДИН КЛИЕНТ – НЕСКОЛЬКО СЕРВЕРОВ

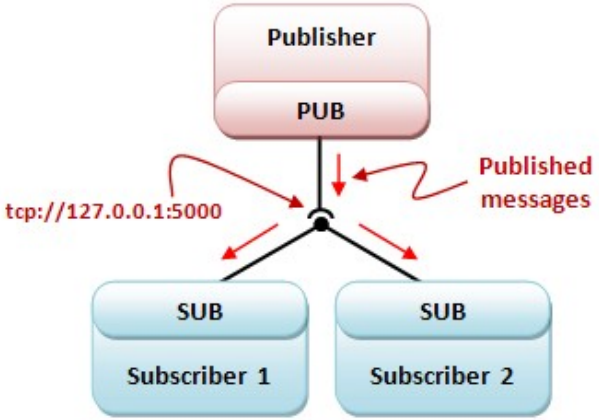


ШАБЛОНА ПУБЛИКАЦИИ-ПОДПИСКИ

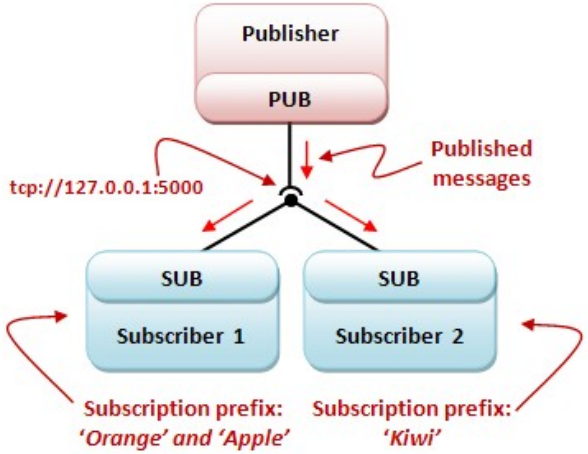
- Предполагается, что сервер использует **сокет типа PUB**, в то время как каждый клиент должен использовать **сокеты типа SUB**. Каждый клиентский сокет подключен к сокету сервера.
- У одного издателя может быть один или несколько подписчиков.
- Один подписчик может быть подключен к одному или нескольким издателям.
- Клиенты **подписываются** на все (с помощью метода **SubscribeAll**) или только определенные сообщения (с помощью метода **Subscribe**), которые **публикуются** серверами. Фактически, **будут передаваться** только **сообщения**, на которые **подписан клиент**. Если сервер публикует сообщения, на которые **никто не подписался**, эти сообщения будут **потеряны**.
- Подписчик может **отказаться от подписки** либо **на все сообщения** издателя, используя метод **UnsubscribeAll**, либо **на конкретное сообщение**, используя метод **Unsubscribe** и указав префикс сообщения в качестве параметра метода.
- В своей **простейшей форме** этот шаблон устанавливает **многоадресную** рассылку сообщений от сервера нескольким клиентам.
- **По умолчанию клиент не подписывается** ни на какое конкретное сообщение. Это означает, что пока явная подписка не предоставлена, клиент **не получит сообщение**, опубликованное сервером

ВАРИАНТЫ ИСПОЛЬЗОВАНИЯ ШАБЛОНА «ИЗДАТЕЛЬ/ПОДПИСЧИК»

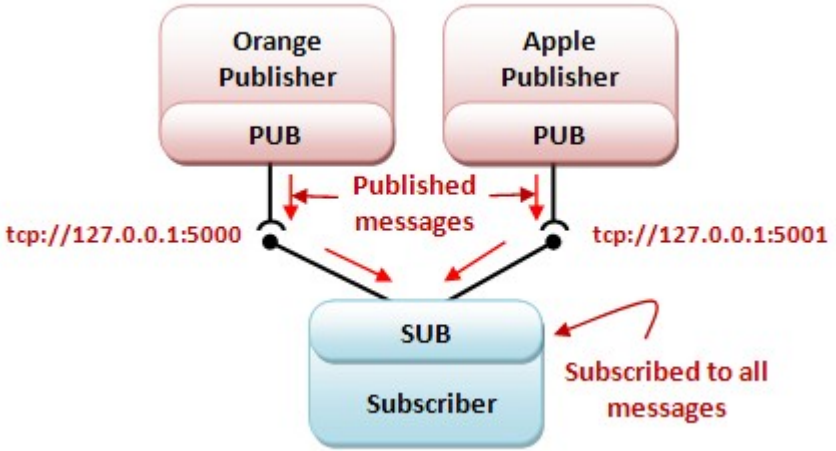
1. ОДИН ИЗДАТЕЛЬ – ДВА ПОДПИСЧИКА (НА ВСЕ СООБЩЕНИЯ)



2. ОДИН ИЗДАТЕЛЬ – ДВА ПОДПИСЧИКА (НА ОТДЕЛЬНЫЕ СООБЩЕНИЯ)



3. ДВА ИЗДАТЕЛЯ – ОДИН ПОДПИСЧИК

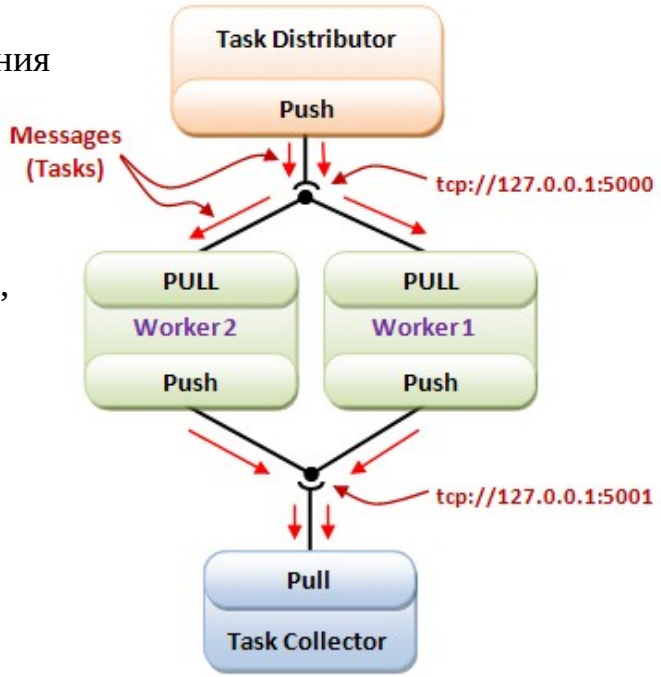


Шаблон конвейера (push/pull)

- Этот шаблон обычно используется, когда есть необходимость в **параллельной обработке данных**.
- Этот шаблон характеризуется тем фактом, что процесс хочет **выдать (протолкнуть) (push out)** свои результаты, предполагая, что есть другие процессы, которые хотят **получить (извлечь) (pull in)** эти результаты.
- Суть шаблона конвейера состоит в том, что **процессу проталкивания** на самом деле не важно, какой другой процесс извлекает свои результаты: **подойдет первый доступный**.
- Точно так же любой процесс, **извлекающий** результаты из нескольких других процессов, **будет делать это с первого процесса отправки**, который сделает свои результаты доступными.
- Таким образом, видно, что цель шаблона конвейера - поддерживать работу как можно большего числа процессов, **продвигая результаты через конвейер** процессов **как можно быстрее**.

СЦЕНАРИЙ ИСПОЛЬЗОВАНИЯ ШАБЛОНА КОНВЕЙЕРА

- Сценарий использования шаблона конвейера выглядит следующим образом:
  - Предположим, что у нас есть процесс **распределитель задач (Task Distributer)**, который циклически отправляет сообщения (**tasks**) исполнителям (каждого исполнителя (**worker**) разные задачи).
  - Когда исполнитель получает сообщение, он обрабатывает его, а затем отправляет его в своего рода сборщик задач (**Task Collector**), который получает сообщения (задачи). Сообщения, полученные сборщиком, ставятся в очередь между всеми подключенными рабочими.
- Этот шаблон имеет следующие характеристики:
  - Распределитель задач использует сокет типа **PUSH**. Он связывается со своей конечной точкой и ожидает получения соединений от исполнителей;
  - У исполнителя есть два сокета:
    - один сокет типа **PULL**, подключенный к сокету распределителя задач,
    - а другой сокет типа **PUSH**, подключенный к сокету сборщика.
  - Сборщик задач имеет сокет типа **PULL**. Он связывается со своей конечной точкой и ожидает получения соединений от рабочих.



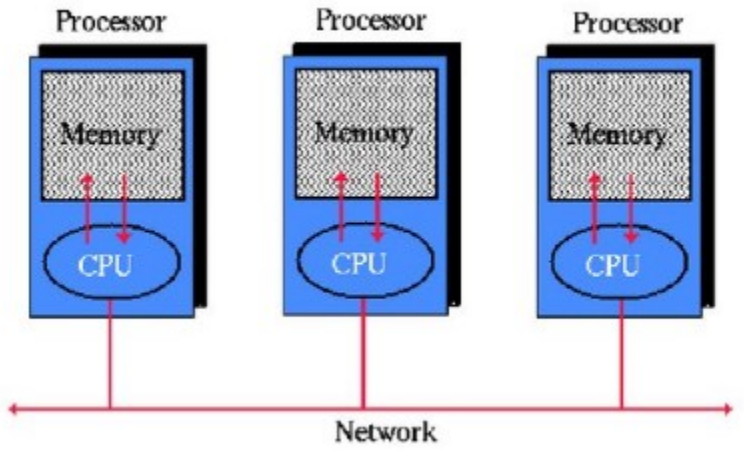
MESSAGE-PASSING INTERFACE (MPI) ИНТЕРФЕЙС ПЕРЕДАЧИ (ОБМЕНА) СООБЩЕНИЙ

СОКЕТЫ И HPC (ВЫСОКОПРОИЗВОДИТЕЛЬНЫЕ ВЫЧИСЛЕНИЯ)

- Сокеты предоставляют низкоуровневый интерфейс доступа к глобальным (TCP/IP-based) сетям.
- Сокеты были признаны недостаточными для приложений HPC по двум причинам.
  - Во-первых, они находились на неправильном уровне абстракции, поддерживая только простые операции отправки и получения.
  - Во-вторых, сокеты были разработаны для связи между сетями с использованием стеков протоколов общего назначения, таких как TCP / IP.
- Сокеты не считаются подходящими для проприетарных протоколов, разработанных для высокоскоростных межсетевых соединений, например, используемых в высокопроизводительных серверных кластерах.
- Эти протоколы требовали интерфейса, который мог бы обрабатывать более продвинутые функции, такие как различные формы буферизации и синхронизации.
- Распределенные системы, работающие на высокоскоростных сетях в высокопроизводительных кластерных системах требуют более эффективных протоколов обмена сообщениями, обеспечивающих эффективные варианты буферизации и синхронизации.

ИНТЕРФЕЙС ПЕРЕСЫЛКИ СООБЩЕНИЙ (MESSAGE-PASSING INTERFACE - MPI)

- Часто высокопроизводительные мультикомпьютерные вычислительные системы имеют свои собственные коммуникационные библиотеки. Необходимость в платформе обмена сообщениями независящей от аппаратной платформы привела к созданию стандарта обмена сообщениями – MPI.
- Необходимость быть независимой от оборудования и платформы в конечном итоге привела к определению стандарта для передачи сообщений, называемого просто **интерфейсом передачи сообщений** или **MPI** (Message-Passing Interface).
- MPI разработан для параллельных приложений и, как таковой, адаптирован к переходным процессам связи. Он напрямую использует базовую сеть. Кроме того, предполагается, что серьезные сбои, такие как сбои процессов или сетевых разделов, являются фатальными и не требуют автоматического восстановления.

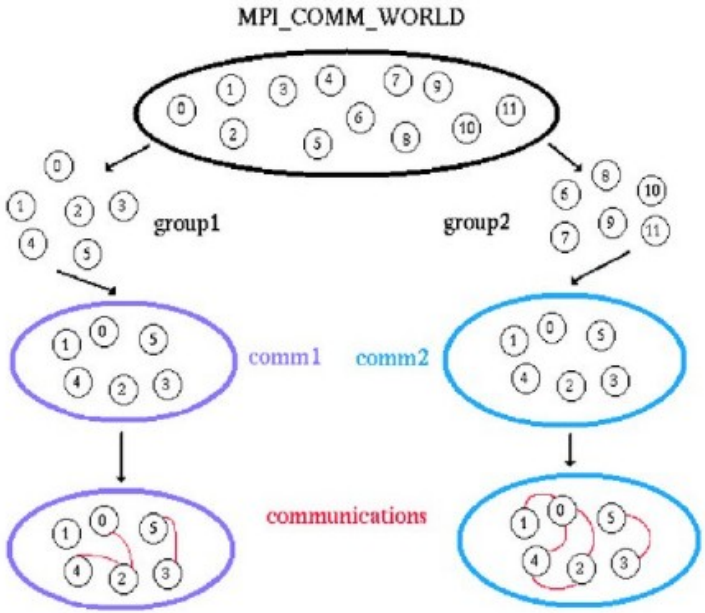


MPI (MESSAGE-PASSING INTERFACE)

- MPI - это библиотека спецификаций обмена сообщениями, предложенная в качестве стандарта (is a library specification for message-passing, proposed as a standard by) комитетом производителей, реализаторов и пользователей.
- MPI разработан для обеспечения обмена сообщениями между узлами вычислительного кластера на основе асинхронной сохранной (persistent) связи сообщений при выполнении параллельных вычислений. Но поддерживаются и синхронные взаимодействия.
- MPI используется во многих системных окружениях, включая как вычислительные кластеры, так и гетерогенные сети.
- MPI является платформо-независимым решением.
- MPICH2 - это популярная реализация стандарта MPI .

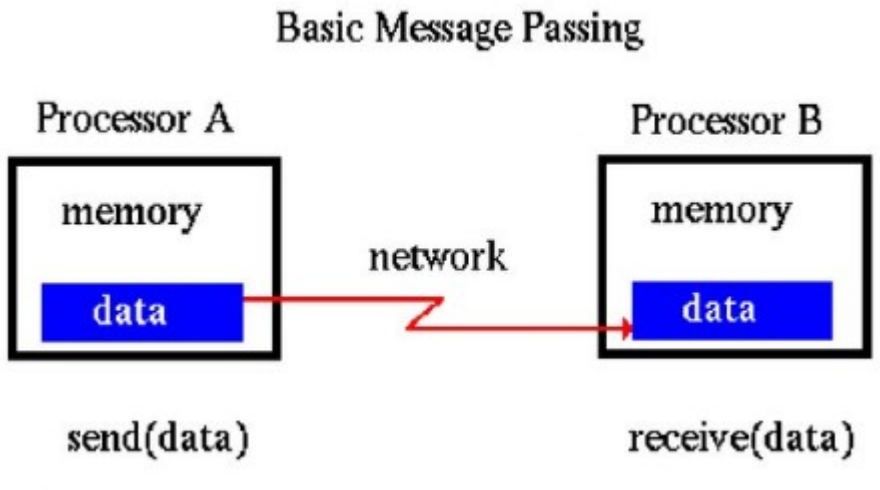
КОММУНИКАЦИИ В MPI

- MPI предполагает использование базовых сетей и не предусматривает ничего, напоминающего коммуникационные серверы. Кроме того, он предусматривает, что серьезные сбои в системе, такие как аварии процессов или участков сети, фатальны и не могут быть восстановлены автоматически.
- Коммуникации предполагаются между группой процессов известных друг другу.
- Для группу прцессов назначается свой уникальный идентификатор - groupID . Для каждого процесса включенного в группу назначается свой идентификатор процесса – processID.
- Пара (groupID, processID) выполняют роль адреса процесса при коммуникациях на основе MPI.



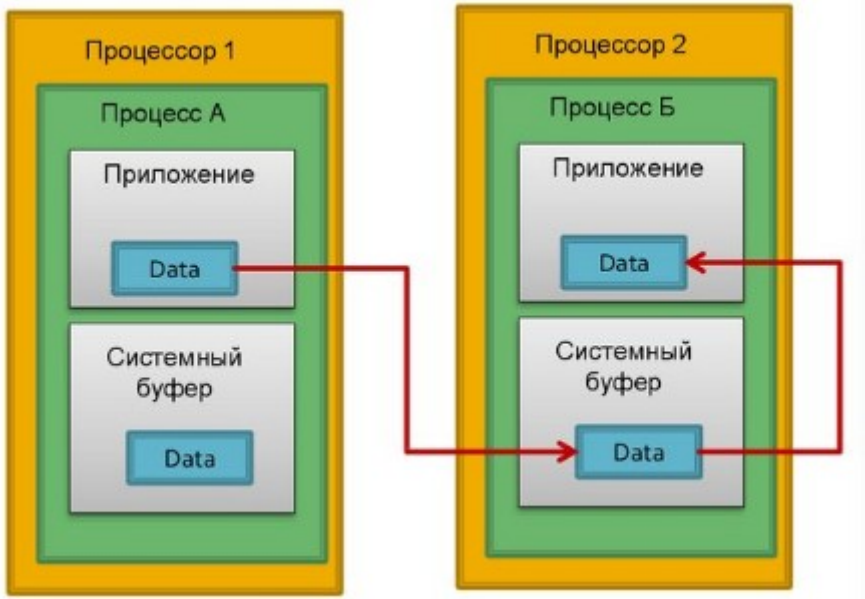
КОММУНИКАЦИОННАЯ СХЕМА MPI

- Коммуникационные функции MPI предоставляют вычислительным процессам MPI-программы работающей на узлах кластера следующие способы взаимодействия:
  - Индивидуальные;
  - Групповые.



КОММУНИКАЦИИ ТОЧКА-ТОЧКА

- Различают следующие виды индивидуальных коммуникаций:
  - Блокируемые/неблокируемые
  - Синхронные/асинхронные
  - Буферизованные
  - Пересылка по готовности



НЕКОТОРЫЕ ИЗ НАИБОЛЕЕ ИСПОЛЬЗУЕМЫХ ПРИМИТИВОВ MPI

Примитив	Назначение
MPI_bsend	Поместить исходящее сообщение в локальный буфер отсылки
MPI_send	Послать сообщение и ожидать, пока оно не будет скопировано в локальный или удаленный буфер
MPI_ssend	Послать сообщение и ожидать начала его передачи на обработку
MPI_sendrecv	Послать сообщение и ожидать ответа
MPI_isend	Передать ссылку на исходящее сообщение и продолжить работу
MPI_issend	Передать ссылку на исходящее сообщение и ожидать начала его передачи на обработку
MPI_recv	Принять сообщение, блокировать работу в случае его отсутствия
MPI_irecv	Проверить наличие входящих сообщений, не блокируя работы

ОПЕРАТОРЫ (ПРИМИТИВЫ) MPI ПЕРЕДАЧИ СООБЩЕНИЙ. АСИНХРОННАЯ СВЯЗЬ.

- Оператор MPI_bsend: обеспечивает **асинхронную сохранную** связь между процессами.
  - После выполнения этой операции, передающий процесс (sender) продолжает выполнение кода, одновременно с копирование сообщения в локальный буфер для последующей доставки процессу получателю (bsend = buffer send).
  - Сообщение будет скопировано в буфер на принимающей машине и позднее будет получено процессом в ответ на выдачу оператора (примитива) receive.

ОПЕРАТОРЫ (ПРИМИТИВЫ) MPI ПЕРЕДАЧИ СООБЩЕНИЙ. СИНХРОННАЯ СВЯЗЬ

- MPI_send: блокирует передачу до тех пор пока сообщение не будет скопировано в локальный или удаленный буфер (синхронная связь). Семантика зависит от реализации.
  - MPI_ssend: передающий процесс блокируется до тех пор пока запрос не будет принят приемником.
  - MPI_sendrecv: отправляет сообщение и ожидает ответа (по сути аналогично RPC)
- В целом MPI поддерживает большое число функций (более 100), предназначенных для организации сетевых взаимодействий в многомашинных вычислительных системах (вычислительных кластерах).

ОПЕРАТОРЫ (ПРИМИТИВЫ) MPI. ПРИЕМ СООБЩЕНИЙ.

- Операция `MPI_recv` вызывается для приема сообщения и блокирует запустивший процесс до прихода сообщения (синхронная связь).
- Существует также и асинхронный вариант этой операции под именем `MPI_irecv`, вызовом которого получатель показывает, что он готов к приему сообщений. Получатель может проверить, имеются ли пришедшие сообщения, или заблокироваться в ожидании таковых.

ПРИЧИНА ПОДДЕРЖКИ MPI БОЛЬШОГО ЧИСЛА ВАРИАНТОВ ВЗАИМОДЕЙСТВИЯ

- Причина поддержки такого разнообразия вариантов взаимодействия состоит в том, что разработчики систем MPI должны иметь все возможности для **оптимизации производительности** многомашинных вычислительных систем (MIMD – системы по классификации Флина).
- Семантика коммуникационных примитивов MPI не всегда проста, и иногда замена различных примитивов никак не влияет на правильность программы.

ПРИЛОЖЕНИЯ MPI ПРОТИВ КЛИЕНТ-СЕРВЕР

- Процессы в параллельных системах на базе MPI действуют более **независимо**, подобно пирам в одноранговых системах (p2p).
- Коммуникации могут обеспечивать обмен сообщениями **в нескольких** направлениях.
- Коммуникации в системах клиент сервер являются более структурированными.

СОХРАННАЯ СВЯЗЬ НА ОСНОВЕ СООБЩЕНИЙ.

ПРОМЕЖУТОЧНЫЙ УРОВЕНЬ ОРИЕНТИРОВАННЫЙ НА СООБЩЕНИЯ (MESSAGE-ORIENTED MIDDLEWARE - MOM)

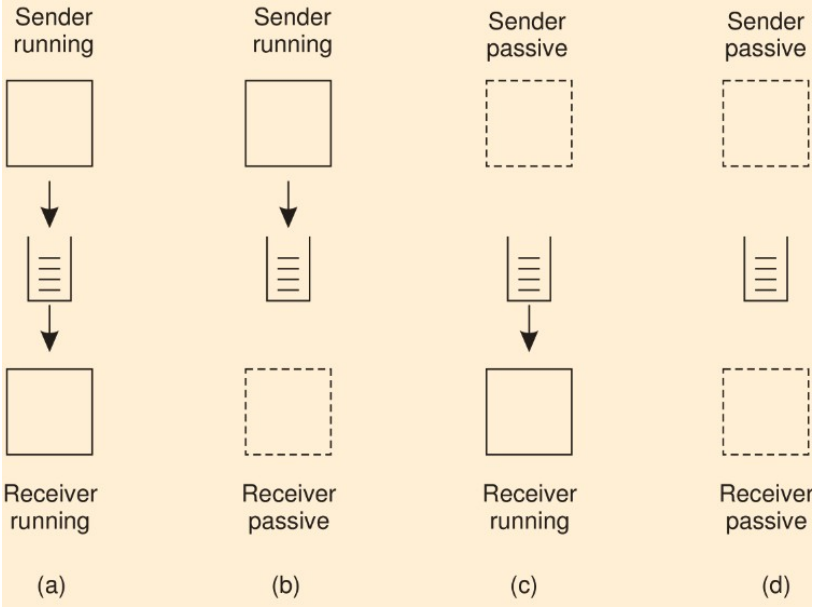
СИСТЕМЫ ОЧЕРЕДЕЙ СООБЩЕНИЙ

- Системы очередей сообщений создают расширенную поддержку асинхронной сохранной связи.
- Смысл этих систем заключается в том, что они предоставляют возможность промежуточного хранения сообщений, не требуя активности во время передачи сообщений ни от отправителя, ни от получателя.
- Их существенное отличие от **сокетов Беркли** и интерфейса **MPI** состоит в том, что системы очередей сообщений обычно предназначены для поддержки обмена сообщениями, занимающего минуты, а не секунды или миллисекунды.

МОДЕЛЬ ОЧЕРЕДЕЙ СООБЩЕНИЙ

- Парадигма очереди сообщений сродни шаблону **издатель-подписчик** Основная идея, лежащая в основе систем очередей сообщений, состоит в том, что **приложения общаются** между собой путем **помещения сообщений** в специальные **очереди**.
- Эти сообщения **передаются по цепочке коммуникационных серверов** и в конце концов достигают места назначения, даже в том случае, если получатель в момент отправки сообщения был неактивен.
- **На практике** большинство коммуникационных серверов **напрямую соединены друг с другом**. Другими словами, сообщение обычно пересылается непосредственно на сервер получателя.
- В принципе **каждое приложение имеет собственную очередь**, в которую могут посылать сообщения другие приложения. Очередь может быть прочитана только связанным с ней приложением, при этом несколько приложений могут совместно использовать одну очередь.
- Важный момент в системах очередей сообщений **СОСТОИТ В ТОМ, ЧТО** отправитель обычно в состоянии **гарантировать** только **попадание сообщения** — рано или поздно — в **очередь получателя**. Никакие гарантии относительно того, будет ли сообщение действительно прочитано, невозможны, это полностью определяется поведением получателя.
- Подобная семантика определяет **слабосвязанное взаимодействие**. Именно поэтому у получателя нет необходимости быть активным в то время, когда сообщение пересылается в его очередь.

ВАРИАНТЫ СЛАБОСВЯЗАННЫХ ВЗАИМОДЕЙСТВИЙ С ИСПОЛЬЗОВАНИЕМ ОЧЕРЕДЕЙ



- Отправитель и получатель могут выполняться абсолютно **независимо друг от друга**.
- На самом деле, как только сообщение поставлено в очередь, оно будет оставаться в ней до удаления, независимо от того, активен его отправитель или его получатель.
- Возможны следующие варианты:

a) отправитель и получатель в ходе всего процесса передачи сообщения **находятся в активном состоянии**.

b) ***активен только*** **отправитель**, в то время как получатель отключен, то есть находится в состоянии, исключающем возможность доставки сообщения. Тем не менее отправитель все же в состоянии отправлять сообщения.

c) комбинация из **активного получателя и пассивного отправителя**. В этом случае получатель может прочитать сообщения, которые были посланы ему ранее, наличия работающих отправителей этих сообщений при этом совершенно не требуется.
d) вариант когда система сохраняет и передает сообщения, даже **при неработающих отправителе и получателе**.

БАЗОВЫЙ ИНТЕРФЕЙС ОЧЕРЕДИ

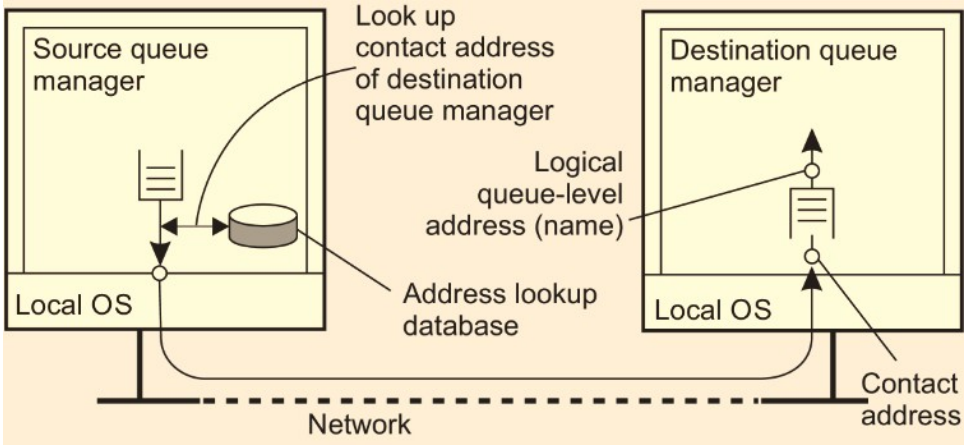
- Сообщения в принципе могут содержать любые данные. Единственно важный момент — они **должны быть правильно адресованы**.
- На практике адресация осуществляется путем предоставления уникального в пределах системы **имени очереди**, в которую направляется письмо.
- В некоторых случаях **размер сообщений** может быть **ограничен**, несмотря на то что, возможно, базовая система в состоянии **разбивать** большие сообщения **на части** и собирать их обратно в единое целое абсолютно прозрачно для приложений.
- Эффект подобного подхода — В ТОМ, что базовый интерфейс, предоставляемый приложениям, в результате можно сделать очень простым.

Примитив	Назначение
Put	Добавить сообщение в соответствующую очередь.
Get	Приостановить работу до тех пор, пока в соответствующей очереди не появятся сообщения, затем извлечь первое сообщение
Poll	Приостановить работу до тех пор, пока в соответствующей очереди не появятся сообщения, затем извлечь первое сообщение
Notify	Вставить дескриптор функции, которая будет вызвана при помещении сообщения в соответствующую очередь

ФУНКЦИИ БАЗОВОГО ИНТЕРФЕЙСА ОЧЕРЕДИ

- Примитив **`put`** вызывается отправителем для передачи сообщения базовой системе, где оно будет помещено в соответствующую очередь. Это **неблокирующий** вызов.
- Примитив **`get`** — это блокирующий вызов, посредством которого авторизованный процесс может извлечь самое старое сообщение из определенной очереди. Процесс блокируется только в том случае, если очередь пуста. Варианты этого вызова включают в себя поиск в очереди определенного сообщения, например, с использованием приоритетов или образца для сравнения.
- Неблокирующий вариант представлен примитивом **`poll`**. Если очередь пуста или искомое сообщение не найдено, вызвавший этот примитив процесс продолжает свою работу.
- И, наконец, большинство систем очередей сообщений поддерживают также процесс **вставки дескриптора функции обратного вызова** (callback function), которая автоматически вызывается при попадании сообщения в очередь (примитив **`notify`**).
- **Обратные вызовы** могут быть также использованы для автоматического запуска процесса, который будет забирать сообщения из очереди, если ни один процесс в настоящее время не запущен. Подобное поведение обычно реализуется при помощи демона на стороне получателя, который постоянно проверяет очередь на наличие входящих сообщений и поступает в соответствии с результатами проверки.

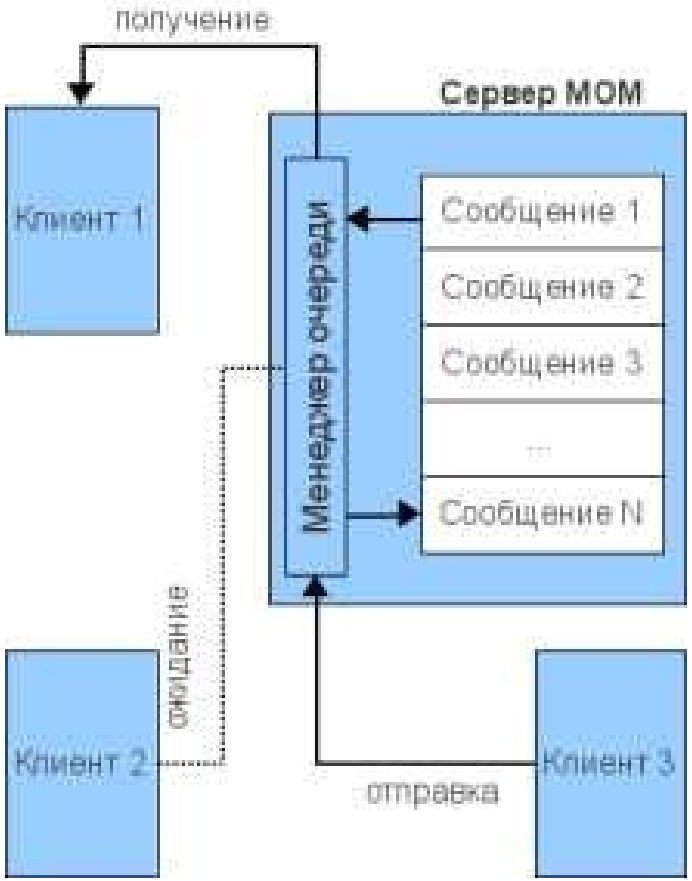
ОБЩАЯ АРХИТЕКТУРА СИСТЕМЫ ОЧЕРЕДЕЙ СООБЩЕНИЙ



- Важно понимать, что набор очередей разнесен по множеству машин.
- Соответственно, для того чтобы система очередей сообщений могла перемещать сообщения, она должна поддерживать **отображение очередей** на **сетевые адреса**.

- На практике это означает, что она должна поддерживать **базу данных** (возможно, распределенную) ***имен очередей*** *(queue names) в соответствии с их сетевым местоположением,* как показано на рис.
- Это отображение **полностью аналогично** использованию системы доменных имен (DNS) для **электронной почты** Интернета. Так, например, для отсылки почты на логический *почтовый адрес* steen@cs.vu.nl почтовая система запрашивает у DNS *сетевой адрес (то есть IP-*адрес) почтового сервера получателя, чтобы затем использовать его для передачи сообщений.

УПРАВЛЕНИЕ ОЧЕРЕДЯМИ В РАСПРЕДЕЛЕННЫХ СИСТЕМАХ



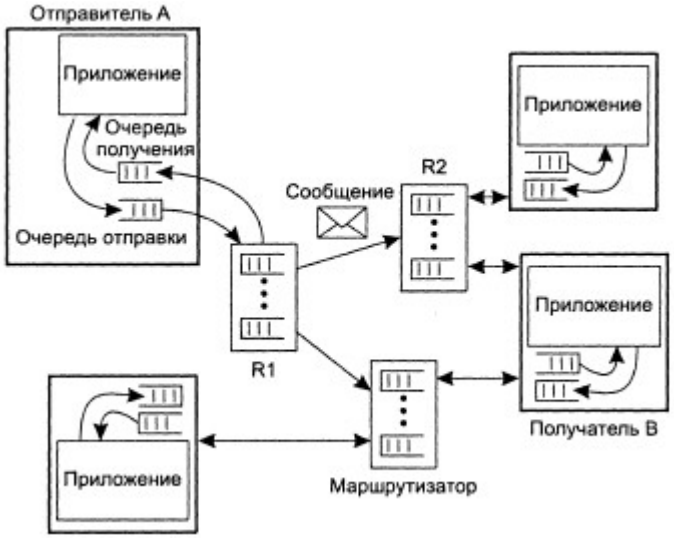
- Очереди управляются *менеджерами очередей (queue managers). Обычно менеджер* очередей взаимодействует непосредственно с отправляющими и принимающими сообщения приложениями.
- Очевидно, что при работе с очень большими системами очередей сообщений возникнет проблема масштабируемости.
- Существуют, однако, и специализированные менеджеры очередей, которые работают как:
  - маршрутизаторы, или ретрансляторы.
- Таким образом, система очередей сообщений может постепенно превратиться в законченную **наложенную (оверлейную) сеть** (**overlay network**) на уровне приложений.

РЕТРАНСЛЯТОРЫ ОЧЕРЕДЕЙ

- Ретрансляторы перенаправляют приходящие сообщения другим менеджерам очередей.
- Ретрансляторы могут быть удобны по нескольким причинам. Так, во многих системах очередей сообщений не существует общих служб именования, которые могли бы динамически поддерживать отображение имен на адреса. Вместо этого топология сети с очередями сделана статической, а каждый менеджер очередей имеет копию отображения очередей на адреса.
- В крупномасштабных системах очередей сообщений подобный подход легко может привести к трудностям в управлении сетью.
- Другой причиной использования ретрансляторов является их способность производить вторичную обработку сообщений. Так, например, в *целях безопасности* или защиты от сбоев может быть необходимо ведение журнала сообщений.
- Специальный тип ретрансляторов, в состоянии работать как шлюз, преобразуя сообщения в удобный для получателя вид.
- И, наконец, ретрансляторы могут использоваться для групповой рассылки. В этом случае входящее сообщение просто помещается в каждую из исходящих очередей.

МАРШРУТИЗАТОРЫ ОЧЕРЕДЕЙ

- Этот подход схож с ранней конструкцией системы MBone поверх Интернета, в которой обычные пользовательские процессы конфигурировались для маршрутизации групповых рассылок.
- Сегодня маршрутизаторы сами поддерживают групповые рассылки и необходимость в оверлейных групповых рассылках сильно сократилась.



✓ Однако по мере роста сети очередей сообщений становится ясно, что ручное конфигурирование невозможно при больших размерах сети.

✓ Единственным решением будет использование динамической схемы маршрутизации, такой же как в компьютерных сетях.

➢ Когда отправитель А помещает в локальную очередь сообщение для получателя В, это сообщение передается на ближайший маршрутизатор R1.

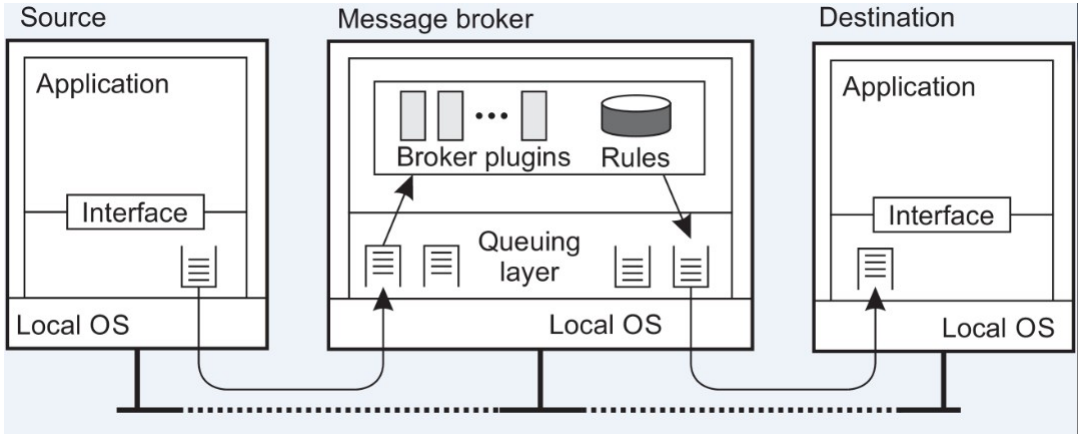
➢ При этом маршрутизатор знает, что делать с этим сообщением, и передает его в направлении В. Так, R1 может понять по имени 5, что сообщение следует передать на маршрутизатор R2.

➢Таким образом, при добавлении или удалении очередей обновление информации потребуется только маршрутизаторам, в то время как остальным менеджерам очередей достаточно будет знать только местоположение ближайшего маршрутизатора.

➢ Следовательно, маршрутизаторы могут помочь в создании масштабируемых систем очередей сообщений.

БРОКЕРЫ СООБЩЕНИЙ

- Важнейшей областью применения систем очередей сообщений является интеграция существующих и новых приложений в единые согласованные распределенные информационные системы.
- Интеграция требует, чтобы приложения понимали сообщения, которые они получают, т.е. управляемые сообщения должны иметь тот формат, которого ожидает получатель.
- Альтернатива состоит в том, чтобы принять единый формат сообщений, как это было сделано с традиционными сетевыми протоколами.
- К сожалению, этот подход для систем очередей сообщений в общем случае неприменим. Проблема состоит в уровне абстракций, которыми оперируют эти системы. Единый формат сообщений имеет смысл, только если набор процессов, использующих этот формат, в реальности достаточно однотипен.



В системах очередей сообщений преобразование производится на специальных узлах сети массового обслуживания, известных под названием *брокеров сообщений (message brokers).*

✓*Брокер сообщений работает как шлюз прикладного* уровня в системе очередей сообщений.
✓Его основная задача — преобразование входящих сообщений в формат, который понимается целевым приложением.
✓Для системы очередей сообщений брокер сообщений — это просто еще одно приложение

СЕРВИСЫ MOM

- Сервисы MOM хорошо зарекомендовали себя в сильно распределенных приложениях, используемых в гетерогенной сети с медленными и ненадежными соединениями. Это, во-многом, достигается благодаря поддержке уровней «качества обслуживания»:
  - надежная доставка сообщений (reliable message delivery) — система MOM гарантирует, что в процессе обмена ни одно сообщение не будет утеряно;
  - гарантированная доставка сообщений (guaranteed message delivery) — сообщение доставляется адресату немедленно или через заданный промежуток времени, не превышающий определенного значения (в случае, если сеть в данный момент не доступна);
  - застрахованная доставка сообщений (assured message delivery) — каждое сообщение доставляется только один раз.

ВИДЫ СЕРВИСОВ MOM

- Очереди сообщений представляют собой мощный, гибкий и в то же время простой механизм межпрограммного взаимодействия.
- Помимо приведенной, можно сказать классической, схемы с очередями, разработаны и используются сервисы MOM с *непосредственной передачей сообщений* и на основе *подписки*.
- Системы с непосредственной передачей сообщений (message passing) используют логическое сетевое соединение для обмена сообщениями между взаимодействующими приложениями. Эта схема удобна в тех случаях, когда клиенты и серверы сообщений используются в сильно связанной сетевой инфраструктуре и синхронизированы по времени.
- Сервисы MOM, обслуживающие клиентов по подписке/публикации (publish&subscribe) работают по принципу, напоминающему почтовую рассылку: одно приложение публикует информацию в сети, а другие подписываются на эту публикацию для получения необходимых данных. Взаимодействующие таким способом приложения полностью независимы друг от друга, что представляет возможности динамической реконфигурации всей распределенной системы.

API И ПРОТОКОЛЫ СИСТЕМ ОЧЕРЕДЕЙ СООБЩЕНИЙ

- Брокер сообщений – это приложение промежуточного слоя *работающее как шлюз прикладного* уровня в системе очередей сообщений.
- Как и любое приложение брокер сообщений может вызываться с помощью API – прикладного программного интерфейса.
- В тоже время к брокер сообщений должен поддерживать какой-нибудь протокол , обеспечивающий сетевое взаимодействие между вызывающей и принимающей сторонами.
- В простейшем случае это может быть уникальный протокол работающий на основе стека TCP/IP.

API С ПОДДЕРЖКОЙ MESSAGE ORIENTED MIDDLEWARE (MOM)

- Java Message Service **(JMS) API - это API с поддержкой Java Message Oriented Middleware (MOM) для отправки сообщений между двумя или несколькими клиентами**. JMS является частью платформы Java, Enterprise Edition.
- Подобные цели преследовал проект **OpenMAMA** (Open Middleware Agnostic Messaging API), который является легковесным уровнем интеграции для систем построенных на основе MOM.

ПРОТОКОЛЫ МОМ

- Среди таких протоколов можно выделить следующие.
  - **MQTT** (Message Queue Telemetry Transport) – упрощенный сетевой протокол, работающий поверх протоколов транспортного уровня,
  - **AMQP** (Advanced Message Queuing Protocol) – протокол обмена сообщениями между компонентами системы,
  - **XMPP**- (Extensible Messaging and Presence Protocol) – расширяемый протокол обмена сообщениями и данными о присутствии,
  - **STOMP** (Simple/Streaming Text Oriented Messaging Protocol) – потоковый тексто-ориентированный протокол обмена сообщениями.

РАСШИРЕННЫЙ ПРОТОКОЛ ОЧЕРЕДИ СООБЩЕНИЙ (AMQP)

НЕОБХОДИМОСТЬ В AQMP

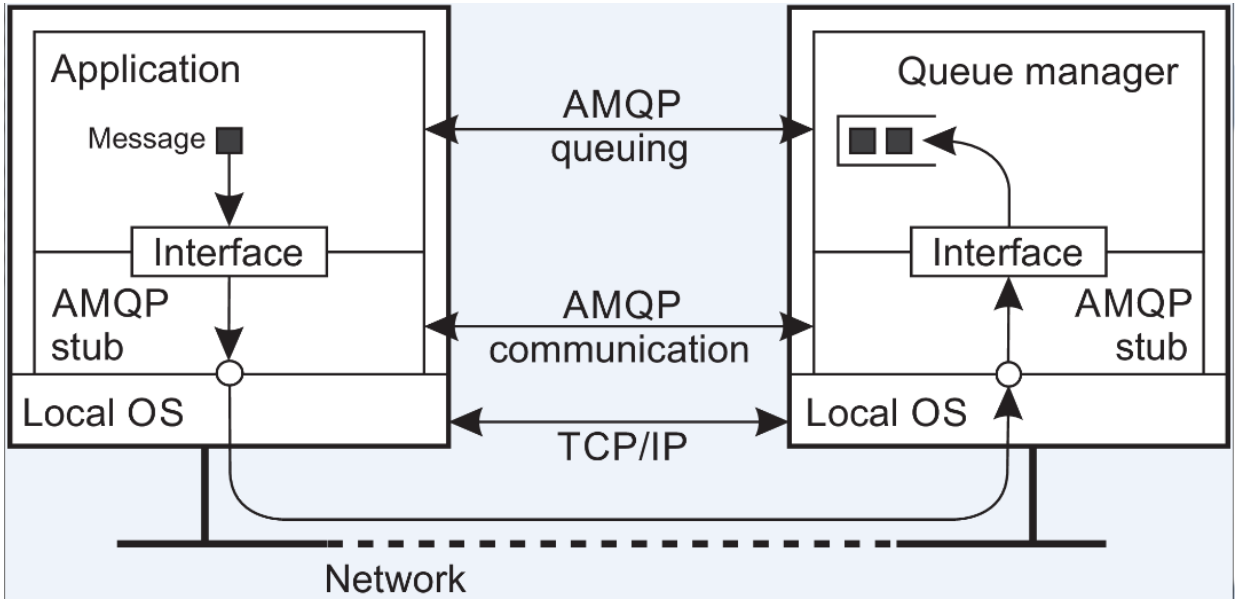
- Как только организация решает использовать **систему очередей сообщений** от производителя **X**, ей, возможно, придется довольствоваться решениями, которые предоставляет **только X**. Таким образом, решения для организации очередей сообщений в значительной степени являются правильными. электронные решения.
- В 2006 году была сформирована рабочая группа для изменения этой ситуации, результатом которой стала спецификация **Advanced Message-Queuing Protocol**, или просто **AMQP**.
- Существуют разные версии AMQP, последняя из которых - 1.0.
- Существуют также различные реализации AMQP, в частности версии до 1.0, которые к моменту создания версии 1.0 приобрели значительную популярность.
- Поскольку версия до 1.0 сильно отличается (насовместимы) от версии 1.0, но при этом имеет постоянную базу пользователей, то мы можем увидеть в реальных РИС различные версии до 1.0.
- AMQP или Advanced Message Queuing Protocol —технология, изначально разработанная **John O'Hara** для нужд банка «JP Morgan Chase & Co».
- Впоследствии информация о структуре сети и исходные тексты библиотек ПО были предоставлены организации **OASIS**. С ее помощью, с **2014 года**, этот протокол получил призвание в виде международного официального стандарта **ISO/IEC 19464**.

AMQP ADVANCED MESSAGE-QUEUING PROTOCOL

- **AMQP** (Advanced Message Queuing Protocol) обеспечивает **взаимодействие между клиентами и брокерами** (промежуточным ПО для обмена сообщениями). Он создан для того, чтобы путем стандартизации сообщений предоставить возможность широкому кругу различных приложений и систем работать вместе независимо от их внутренней структуры.
  - **Брокер** – это приложение, реализующее модель AMQP, которое принимает соединения клиентов для маршрутизации сообщений и т.п.
  - **Сообщение** – это единица передаваемых данных (включая полезные данные и атрибуты сообщения).
  - **Потребитель** – приложение, которое получает сообщения из очередей.
  - **Производитель** – приложение, которое отправляет сообщения в очередь через обмен.
- **Примечание**: AMQP не определяет полезную нагрузку сообщений, потому протокол может передавать **разные типы данных**.

ОСНОВНЫЕ КОМПОНЕНТЫ AMQP

- Модель AMQP, которая предоставляет способы приема, маршрутизации, хранения сообщений, добавления сообщений в очередь и их обработки, основана на четких определениях следующих компонентов:
- **Точка обмена**: часть брокера (то есть сервер), которая получает сообщения и направляет их в очереди.
- **Очередь сообщений**: именованный объект, с которым связаны сообщения и откуда их получают потребители.
- **Привязки:** правила распространения сообщений из точек обмена в очереди.



КОММУНИКАЦИИ AMQP

- AMQP позволяет приложению устанавливать **соединение** (**connection**) с администратором очередей; соединение - это контейнер для ряда односторонних **каналов (channels)**.
- Двунаправленная связь устанавливается через **сеансы (sessions)**: логическое группирование двух каналов. Соединение может иметь несколько сеансов, но обратите внимание, что канал не обязательно должен быть частью сеанса.
- Наконец, для фактической передачи сообщений нужна **ссылка (link)**. По сути, ссылка или, скорее, ее конечные точки отслеживают статус передаваемых сообщений.
- Таким образом, он обеспечивает детализированное управление потоком между приложением и администратором очередей, и, действительно**, разные политики управления могут применяться одновременно для разных сообщений**, которые передаются **через один и тот же сеанс** соединения.
- **Управление потоком** осуществляется **за счет кредитов**: получатель может сообщить отправителю, **сколько сообщений** ему **разрешено** отправить **по** определенной **ссылке**.

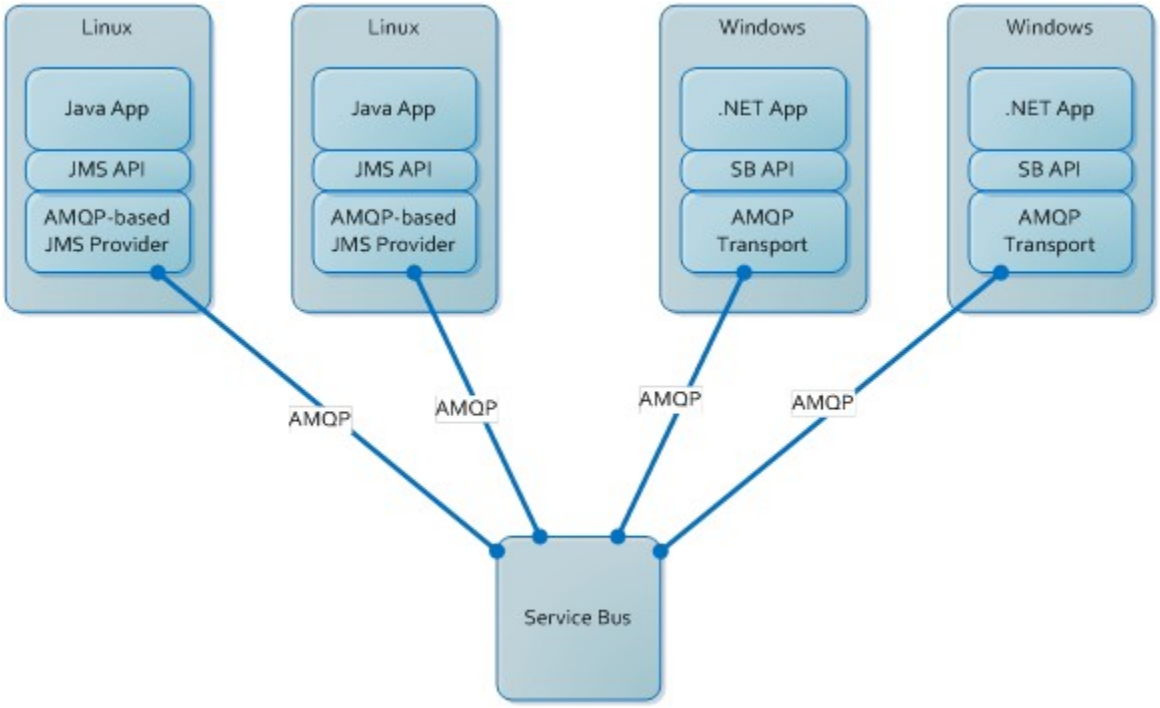
ПЕРЕДАЧА СООБЩЕНИЙ AMQP

- Передача сообщения обычно происходит **в три этапа.**

1. На стороне **отправителя** сообщению **присваивается** уникальный **идентификатор**, и оно записывается локально, чтобы оно находилось **в неустановленном состоянии**. Заглушка впоследствии передает сообщение на сервер, где заглушка AMQP также записывает его как находящееся в неустановленном состоянии. В этот момент заглушка на стороне сервера передает его администратору очередей.
2. Предполагается, что **принимающее приложение** (в данном случае диспетчер очередей) **обрабатывает сообщение** и обычно **сообщает своей заглушке**, что оно **завершено**. Заглушка передает эту информацию исходному отправителю, после чего сообщение в заглушке AMQP исходного отправителя переходит в установленное состояние.
3. **Заглушка AMQP** исходного **отправителя** теперь **сообщает заглушке** исходного **получателя,** что **передача** сообщения **завершена** (это означает, что с этого момента исходный отправитель забудет о сообщении).
4. Заглушка **получателя** теперь также может отбросить все, что связано с сообщением, формально записав это также как выполненное.

ИСПОЛЬЗОВАНИЕ СИСТЕМНОЙ ШИНЫ ДЛЯ ИНТЕГРАЦИИ ПРИЛОЖЕНИЙ СОЗДАННЫХ ДЛЯ РАЗНЫХ ПРОГРАММНО-АППАРАТНЫХ ПЛАТФОРМ

Поддержка протокола AMQP 1.0 в Azure Service Bus означает, что предоставляемые сервисной шиной возможности очередей и обмена сообщениями с использованием брокера публикации/подписки можно использовать на различных платформах.



- **Пример сценария развертывания, показывающий межплатформенный обмен сообщениями с использованием служебной шины и протокола AMQP 1.0**

КОММУНИКАЦИИ ОРИЕНТИРОВАННЫЕ НА ПЕРЕДАЧУ ПОТОКОВ (STREAM-ORIENTED COMMUNICATION)

КОММУНИКАЦИИ ОРИЕНТИРОВАННЫЕ НА ПЕРЕДАЧУ ПОТОКОВ

- Виды связи RPC, RMI, MOM – все они основываются на обмене дискретными данными.
  - Для данных этого типа временные характеристики их доставки не влияют на корректность, а только на производительность обработки.
- Коммуникации на основе потоков предполагают, что содержимое сообщений должно быть доставлено с определенной скоростью, так как от этого зависит корректность его представление.
  - Например, аудио- и видео данные.
- Для воспроизведения звука необходимо, чтобы выборки аудиопотока проигрывались в том же порядке, в котором они представлены в потоке данных, и с интервалами ровно по 1/44100 с. Воспроизведение с другой скоростью создаст неверное представление об исходном звуке.

ПОНЯТИЕ СРЕДЫ ПЕРЕНОСА ИНФОРМАЦИИ

- Под средой понимается то, что несет информацию. Сюда могут входить среды передачи и хранения, среда представления, например монитор, и т. д. Важнейшая характеристика среды — способ *представле*
- Так, текст обычно кодируется символами ASCII или Unicode. Изображения могут быть представлены в различных форматах, например GIF или JPEG. Аудиопотоки в компьютерных системах могут кодироваться, например, с помощью 16-битных выборок, использующих импульсно-кодовую модуляцию.
- В *непрерывгюй среде представления {continuous representation media) временные* соотношения между различными элементами данных лежат в основе корректной интерпретации смысла данных.
- В отличие от непрерывной среды *дискрет7шя среда представления (discrete representation media), характеризуется тем, что временные соотношения между* элементами данных не играют фундаментальной роли в правильной интерпретации данных. Типичными примерами дискретной среды являются представления текста и статических изображений, а также объектный код и исполняемые файлы.

ПОТОК ДАННЫХ

- Для обмена критичной ко времени передачи информацией распределенные системы обычно предоставляют поддержку *потоков данных (data streams, или просто streams). Поток данных есть не что иное, как последовательность элементов* данных.
- Потоки данных применимы как для дискретной, так *и для непрерывной* среды представления.
- Так, каналы UNIX или соединения TCP/IP представляют собой типичные примеры дискретных потоков данных (байт-ориентированных).
- Воспроизведение звукового файла обычно требует непрерывного потока данных между файлом и устройством воспроизведения.
- Временные характеристики важны для непрерывных потоков данных. Для поддержания временных характеристик часто приходится выбирать между различными режимами передачи.

АСИНХРОННЫЙ И СИНХРОННЫЙ РЕЖИМЫ ПЕРЕДАЧИ

- В *асинхронном режиме передачи (asynchronous transmission mode) элементы данных передаются в поток один за другим, но на их* дальнейшую передачу никаких ограничений в части временных характеристик не вводится. Это традиционный вариант для дискретных потоков данных. Так, файл можно преобразовать в поток данных, но выяснять точный момент окончания передачи каждого элемента данных чаще всего бессмысленно.
- В *синхронном режиме передачи (synchronous transmission mode) для каждого* элемента в потоке данных определяется максимальная задержка сквозной передачи.
- Если элемент данных был передан значительно быстрее максимально допустимой задержки, это не важно. Так, например, датчик может с определенной частотой измерять температуру и пересылать эти измерения по сети оператору.

ИЗОХРОННЫЙ РЕЖИМ ПЕРЕДАЧИ

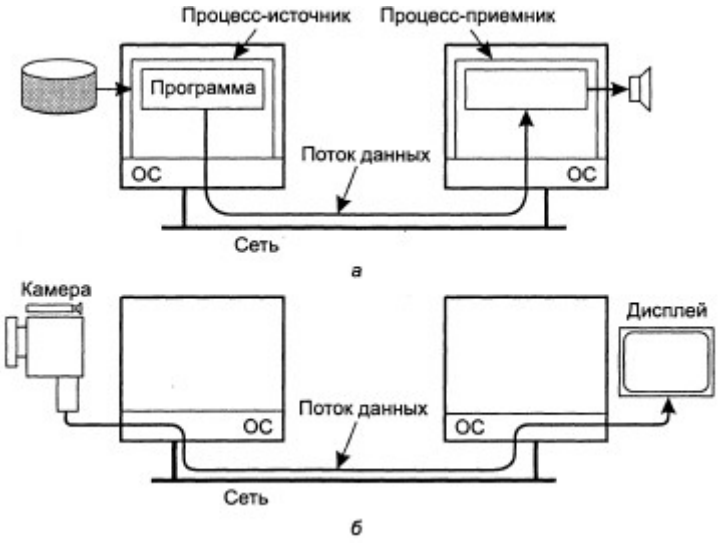
- При *изохронном режиме передачи (isochronous transmission mode)* необходимо, чтобы все элементы данных передавались вовремя. Это означает, что передача данных ограничена максимально, *а также минимально допустимыми* задержками, также известными под названием *предельного дрожания*.
- *Изохронный* режим передачи, в частности, представляет интерес для распределенных систем мультимедиа, поскольку играет значительную роль в воспроизведении аудио- и видеоинформации.

ПРОСТЫЕ И КОМПЛЕКСНЫЕ ПОТОКИ ДАННЫХ

- Простой поток данных (simple stream) содержит только одну последовательность данных.
- Комплексный поток данных (complex stream) состоит из нескольких связанных простых потоков, именуемых вложенными потоками данных (substreams).
- Взаимодействие между вложенными потоками в комплексном потоке часто также зависит от времени.
- Так, например, стереозвук может передаваться посредством комплексного потока, содержащего два вложенных потока, каждый из которых используется для одного аудиоканала.
- Важно отметить, что эти два вложенных потока постоянно синхронизированы. Другими словами, элементы данных каждого из потоков должны передаваться попарно для получения стереоэффекта.

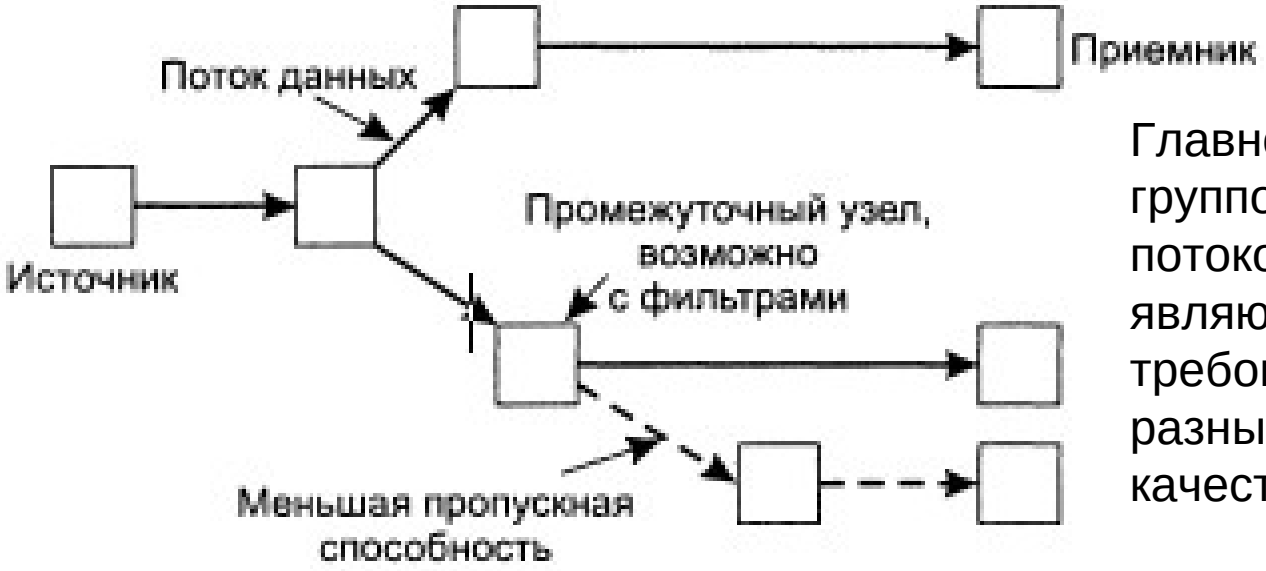
ПЕРЕДАЧА ПОТОКА ДАННЫХ ПО СЕТИ

- При передаче данных через сеть процесс-источник может читать данные из аудиофайла и пересылать их, байт за байтом, по сети. Приемник может быть процессом, по мере поступления выбирающим байты и передающим их на локальное устройство звуковоспроизведения.
- В распределенных мульти-медийных системах можно реализовать прямое соединение между источником и приемником (а).
- Также , видеопоток, создаваемый камерой, может напрямую передаваться на дисплей (б).



ГРУППОВАЯ ПЕРЕДАЧА ПОТОКОВ

- Другая ситуация имеет место в зависимости от того, имеется у нас всего один источник или приемник или мы можем организовать многостороннюю связь.
- Наиболее частая ситуация при многосторонней связи — присоединение к потоку данных нескольких приемников. Другими словами, осуществляется групповая рассылка потока данных нескольким получателям.



Главной проблемой групповой рассылки потоков данных являются разные требования разных приемников к качеству потока.

ПОТОКИ ДАННЫХ И КАЧЕСТВО ОБСЛУЖИВАНИЯ

- Временные зависимости и другие нефункциональные требования обычно выражаются
- в виде требований к *качеству обслуживания (Quality of Sewice, QpS).*
- *Эти* требования описывают, что должны сделать базовая распределенная система и сеть для того, чтобы гарантировать, например, сохранение в потоке данных временных соотношений.
- Требования QoS для непрерывных потоков данных в основном характеризуются временными диаграммами, объемом и надежностью.

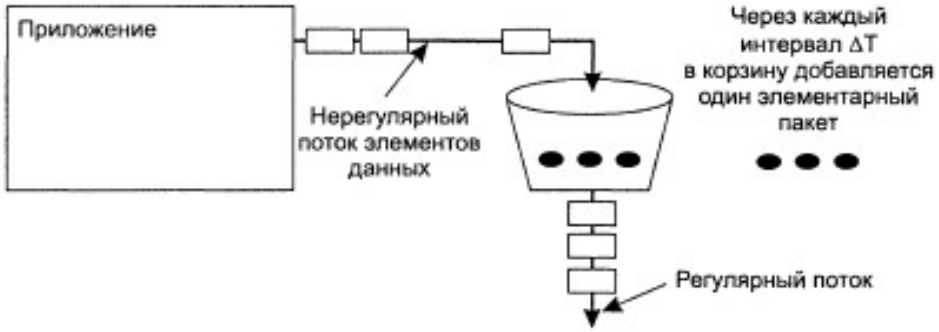
СПЕЦИФИКА QOS

- Требования QoS могут быть выражены по-разному. Один из подходов — предоставить точную *спецификацию передачи (flow specification), содержащую требования* по пропускной способности, скорости передачи, задержке и т. п.

Характеристики получаемых данных	Требуемое качество обслуживания
Максимальный размер элемента данных (байт)	Чувствительность к потерям (байт)
Скорость передачи корзины элементарных пакетов(байт/с)	Чувствительность к интервалам (мкс)
Размер корзины элементарных пакетов(байт)	Чувствительность к групповым потерям (элементов данных)
Максимальная скорость передачи (байт/с)	Минимальная фиксируемаязадержка (мкс) Максимальное отклонение задержки (мкс) Показатель соблюдения (единиц)

АЛГОРИТМ КОРЗИНЫ ЭЛЕМЕНТАРНЫХ ПАКЕТОВ

- Спецификация передачи приведенная выше сформулирована в понятиях алгоритма корзины элементарных пакетов, который описывает, каким образом поток формирует сетевой трафик.
- Основная идея состоит в том, что элементарные пакеты генерируются с постоянной скоростью. Элементарный пакет содержит фиксированное число байтов, которые приложение намерено передать по сети. Элементарные пакеты собираются в корзину, емкость которой ограничена.
- После переполнения корзины элементарные пакеты просто пропадают. Каждый раз, когда приложение хочет передать в сеть элемент данных размером N, оно должно извлечь из корзины столько элементарных пакетов, чтобы их суммарный размер был не менее N байт. То есть, например, если каждый элементарный пакет N имеет длину k байт, приложение должно удалить из корзины как минимум N/k элементарных пакетов.



СПАСИБО ЗА ВНИМАНИЕ !