

2. Взаимодействие между процессами в РС

Сухорослов Олег Викторович

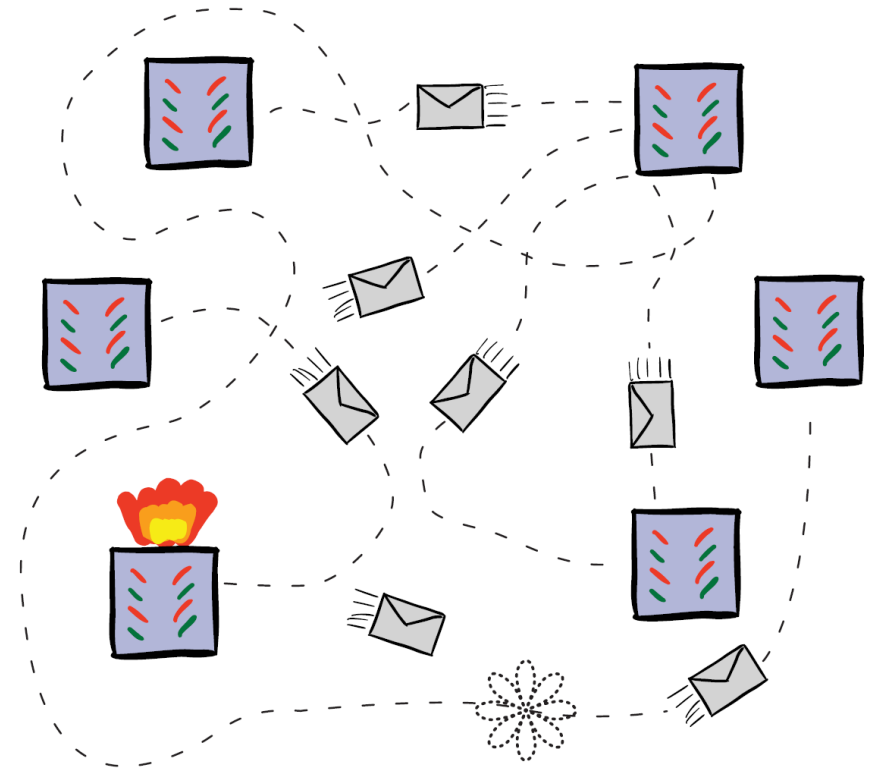
16.09.2024

План лекции

- Возможные разновидности взаимодействий
- Передача сообщений между парой процессов
- Схема "запрос-ответ" и удаленные вызовы процедур (RPC)

Обмен сообщениями (Message Passing)

- Наиболее общая и универсальная **модель взаимодействия** процессов в РС
- Сообщения передаются по **ненадежным каналам**
- Реализация может предоставлять некоторые **гарантии** по доставке сообщений
- Возможны различные **варианты и схемы** взаимодействий

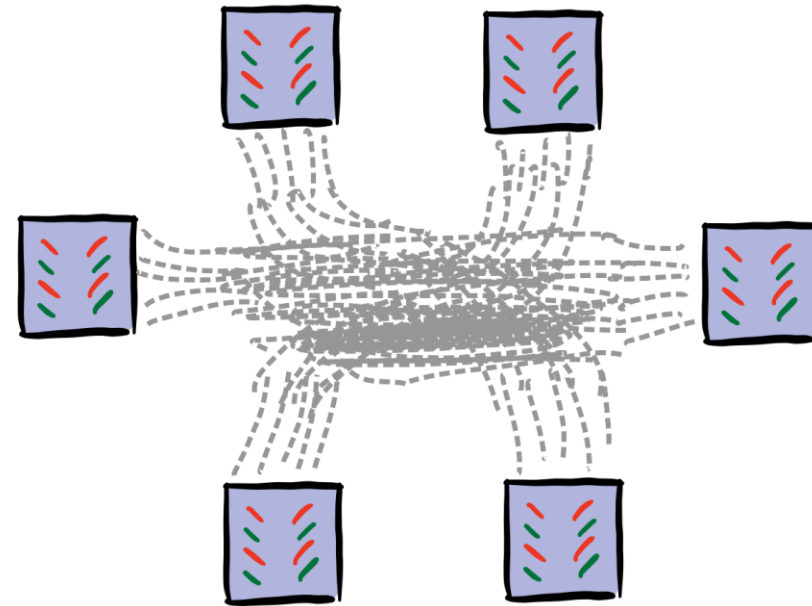
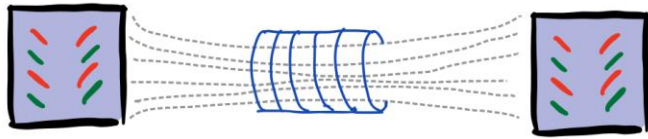


Варианты взаимодействий

- Сколько процессов взаимодействует?
- В каких направлениях передаются сообщения?
- Требуется ли ответ от получателя?
- Кто инициирует передачу сообщения?
- Должны ли отправитель и получатель "знать" друг друга?
- Должны ли процессы работать одновременно?

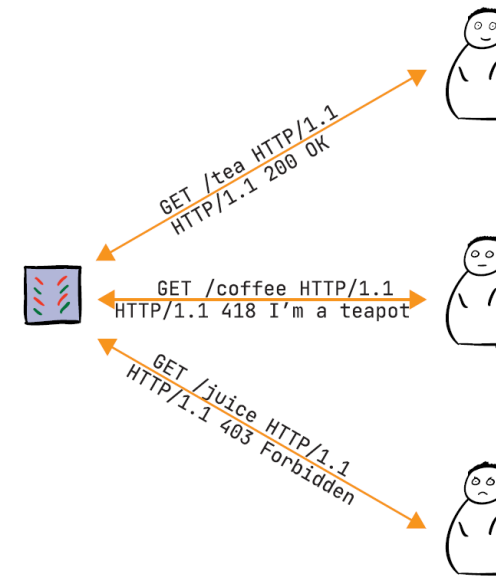
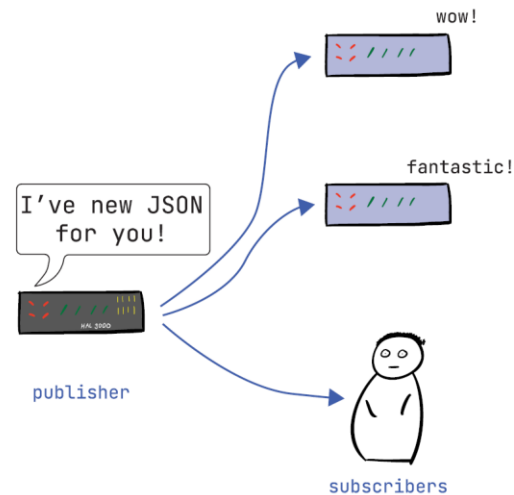
Число взаимодействующих процессов

- Парные взаимодействия (point-to-point communication)
- Групповые взаимодействия (group communication)



Направления передачи сообщений

- В одну сторону (unidirectional)
 - роли отправителей и получателей зафиксированы
 - пример: издатель-подписчик
- В обе стороны (bidirectional)
 - процесс может быть как отправителем, так и получателем
 - пример: клиент-сервер

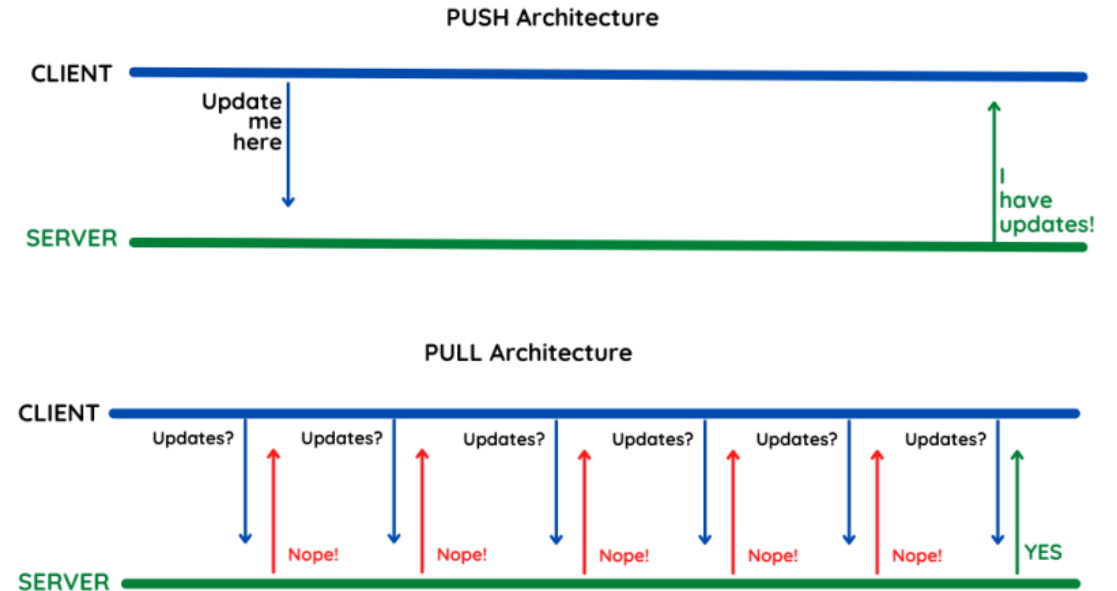


Требуется ли ответ от получателя?

- Отправка в одну сторону (one-way)
- Схема "запрос-ответ" (request-response)

Кто инициирует передачу сообщения?

- Push
 - отправитель инициирует доставку
 - получатель пассивен
- Pull
 - получатель инициирует доставку
 - отправитель пассивен

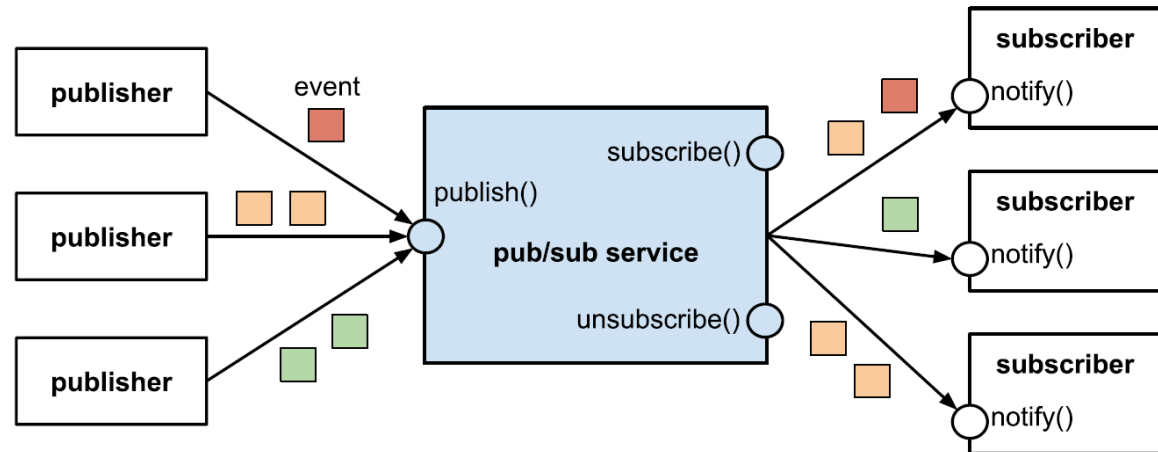


Связывание процессов

- Связывание по пространству (space coupling)
 - Процессы должны обладать информацией друг о друге
 - Например, отправитель должен знать адрес получателя
- Связывание по времени (time coupling)
 - Процессы должны выполняться в одно время
 - Transient vs persistent communication

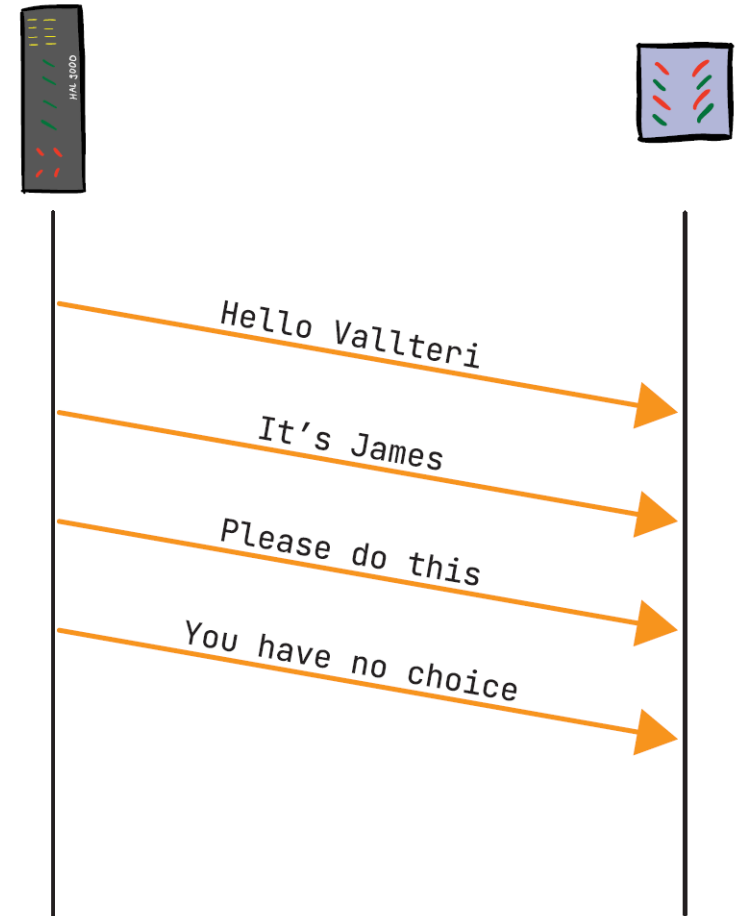
Непрямое взаимодействие (indirect)

- Происходит через некоторого посредника или абстракцию, без прямого связывания между отправителями и получателями
- Примеры
 - Очередь сообщений
 - Издатель-подписчик
 - Общая память



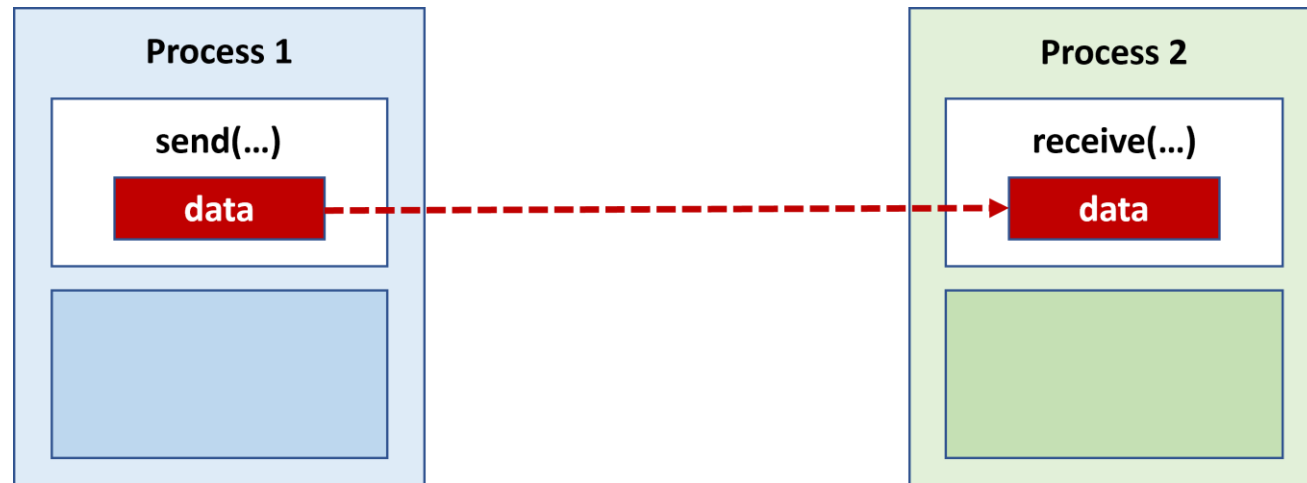
Простейший вариант

- Два процесса
- Один отправитель, другой получатель
- Передачу инициирует отправитель
- Отправитель знает адрес получателя
- Процессы работают одновременно



Примитивы взаимодействия

- `send(data, dest)`
 - `data`: указатель на буфер с отправляемыми данными, их размер
 - `dest`: адрес получателя
- `receive(data, source)`
 - `data`: указатель на буфер с принимаемыми данными, их размер
 - `source`: адрес отправителя

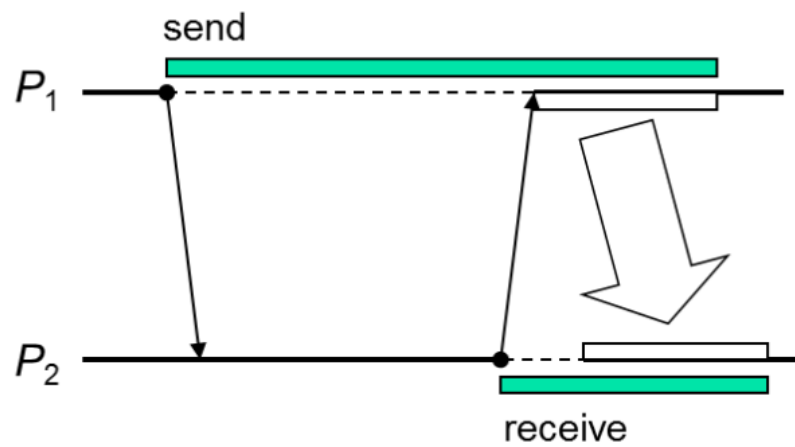


Когда завершается send()?

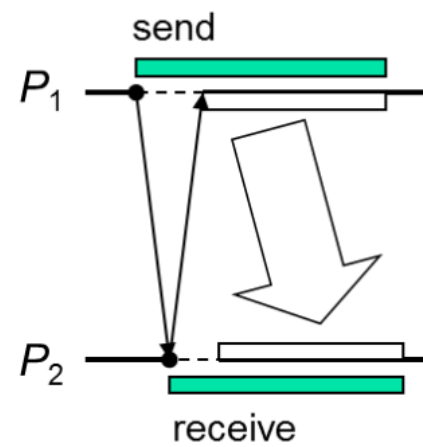
- Сообщение принято процессом-получателем?
- Сообщение покинуло узел процесса-отправителя?
- Буфер можно повторно использовать, не опасаясь испортить сообщение?

Process 1	Process 2
<pre>data = 100 send(&data, process2) data = 0</pre>	<pre>receive(&data, process1) print(data)</pre>

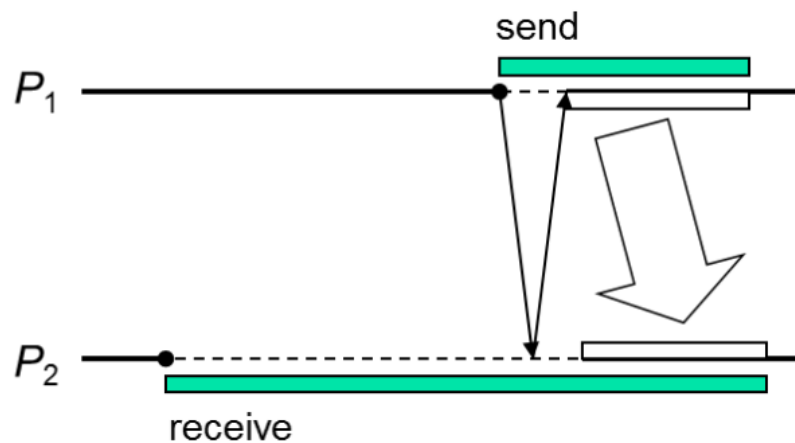
Блокирующая передача без буферизации



(a)



(б)

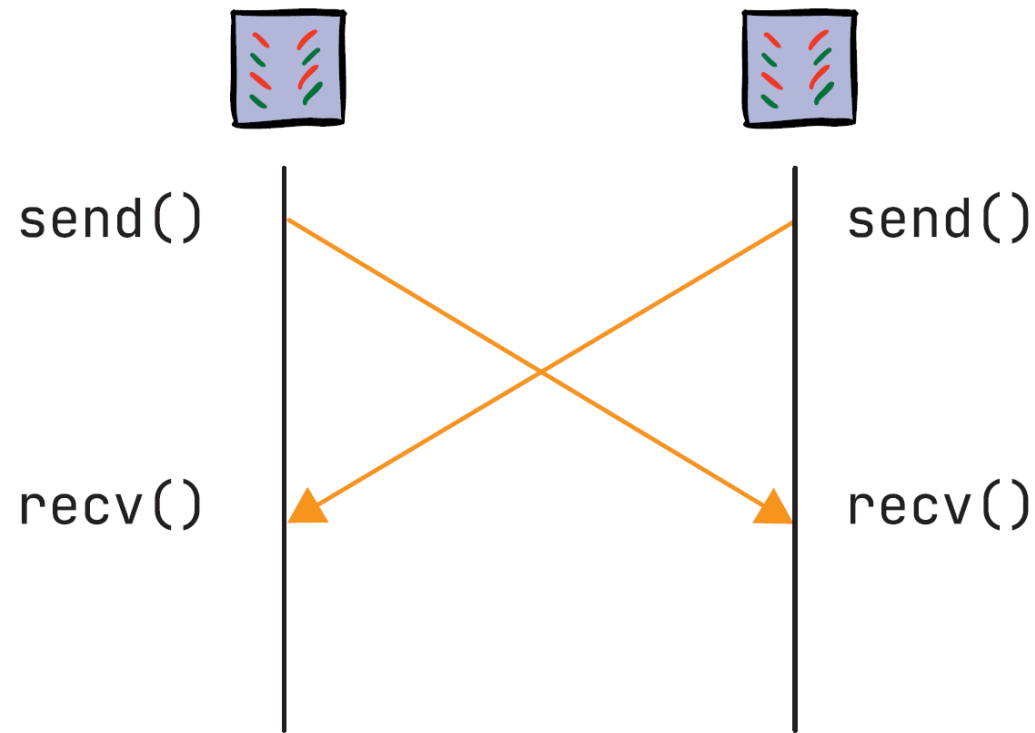


(B)

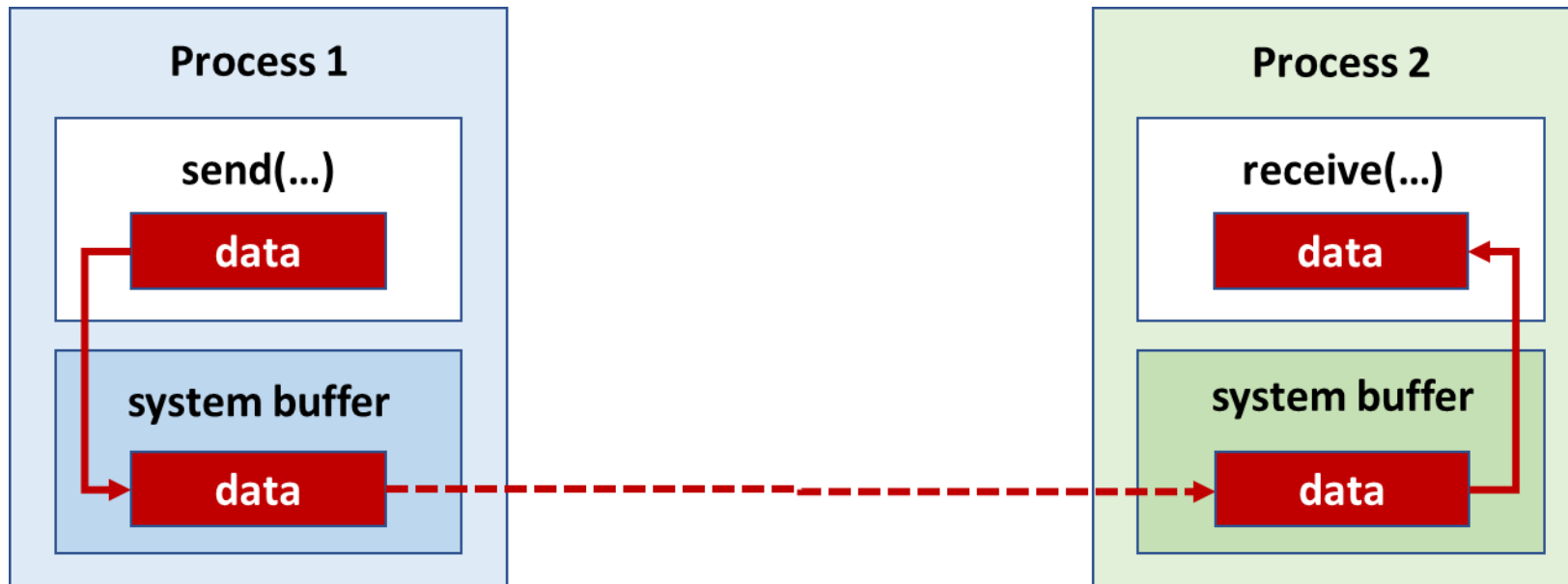
Время →

- вызов команды `send()` или `receive()`
- длительность блокировки процесса
- длительность отправки / приема данных

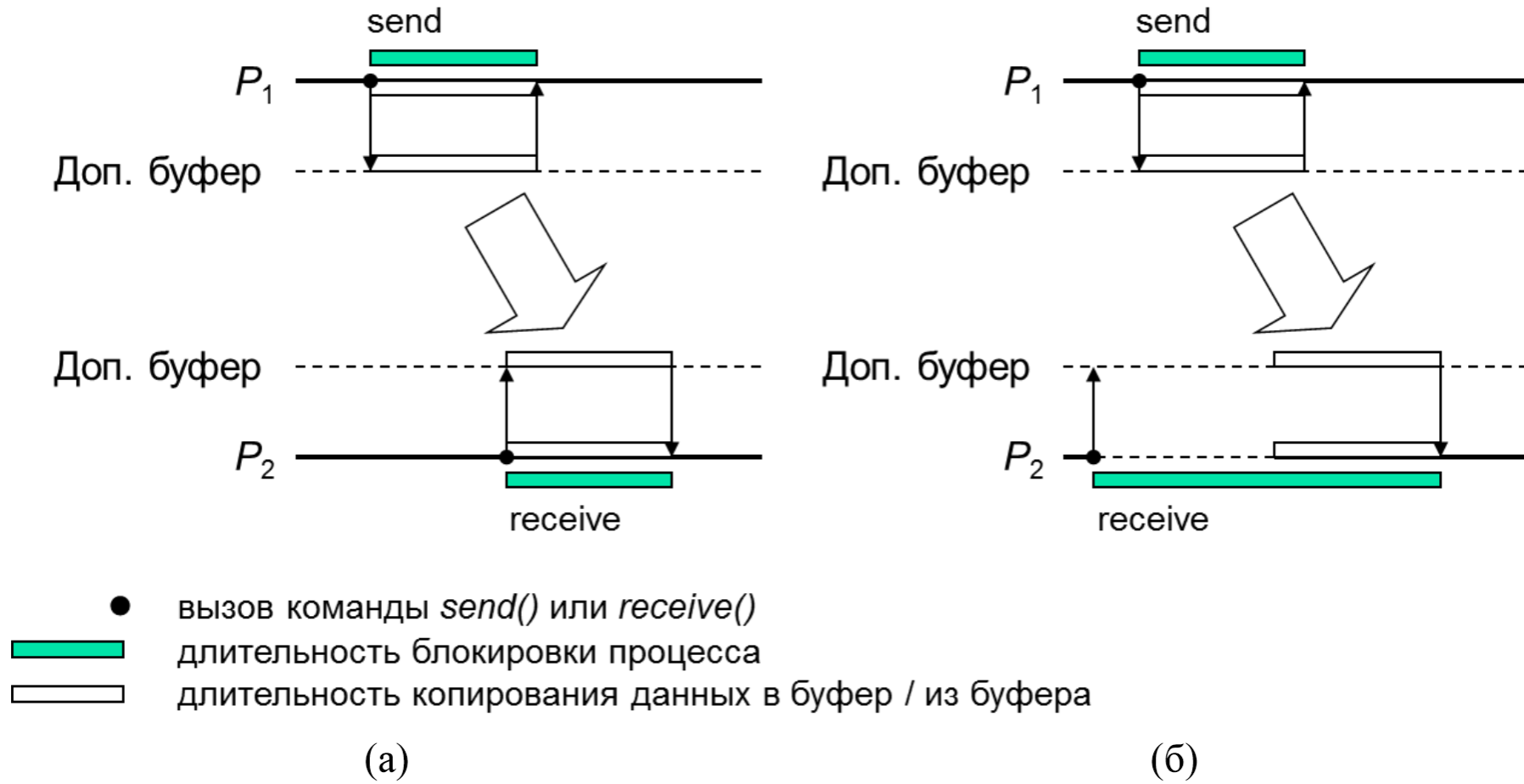
Какая здесь возможна проблема?



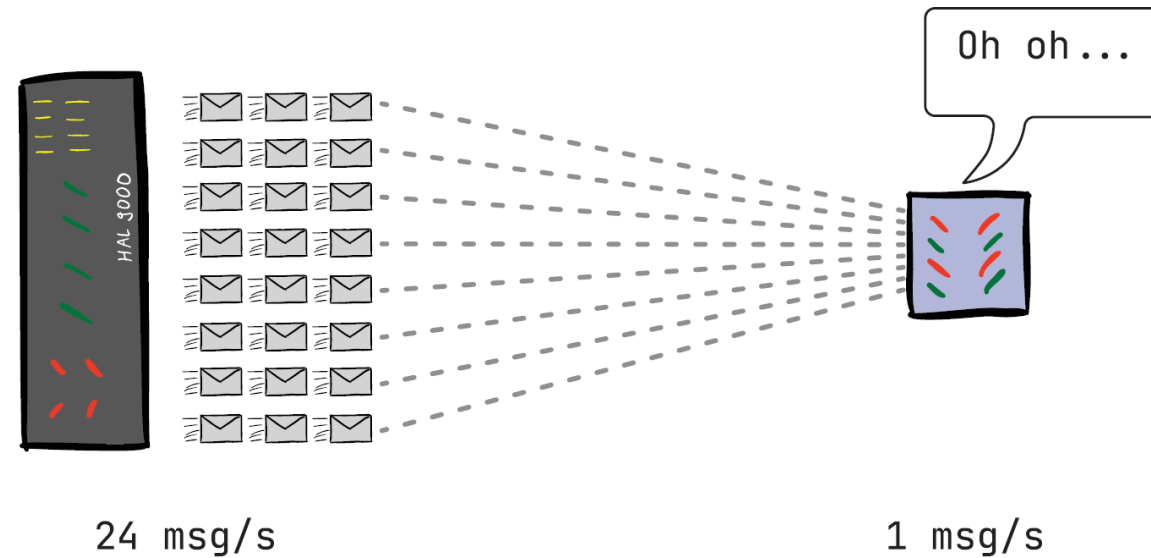
Использование системных буферов



Блокирующая передача с буферизацией



Слишком быстрый отправитель



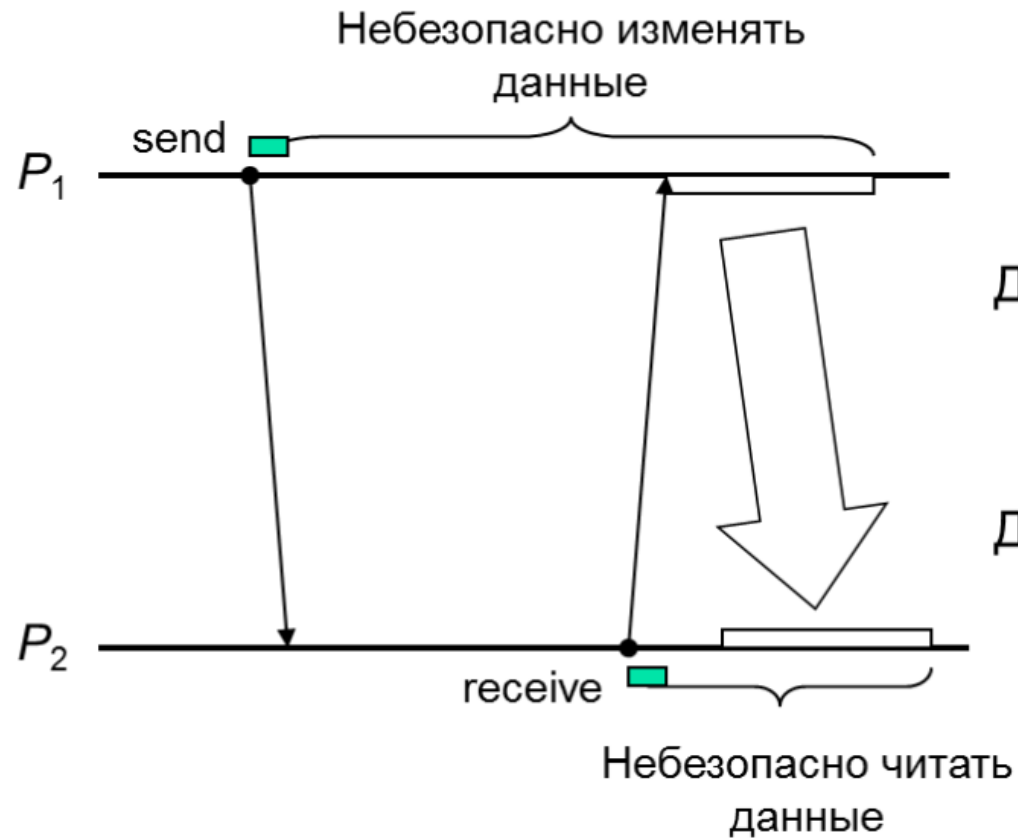
Process 1	Process 2
<pre>loop { produce_data(&data) send(&data, process2) }</pre>	<pre>loop { receive(&data, process1) consume_data(&data) }</pre>

Взаимные блокировки при буферизации

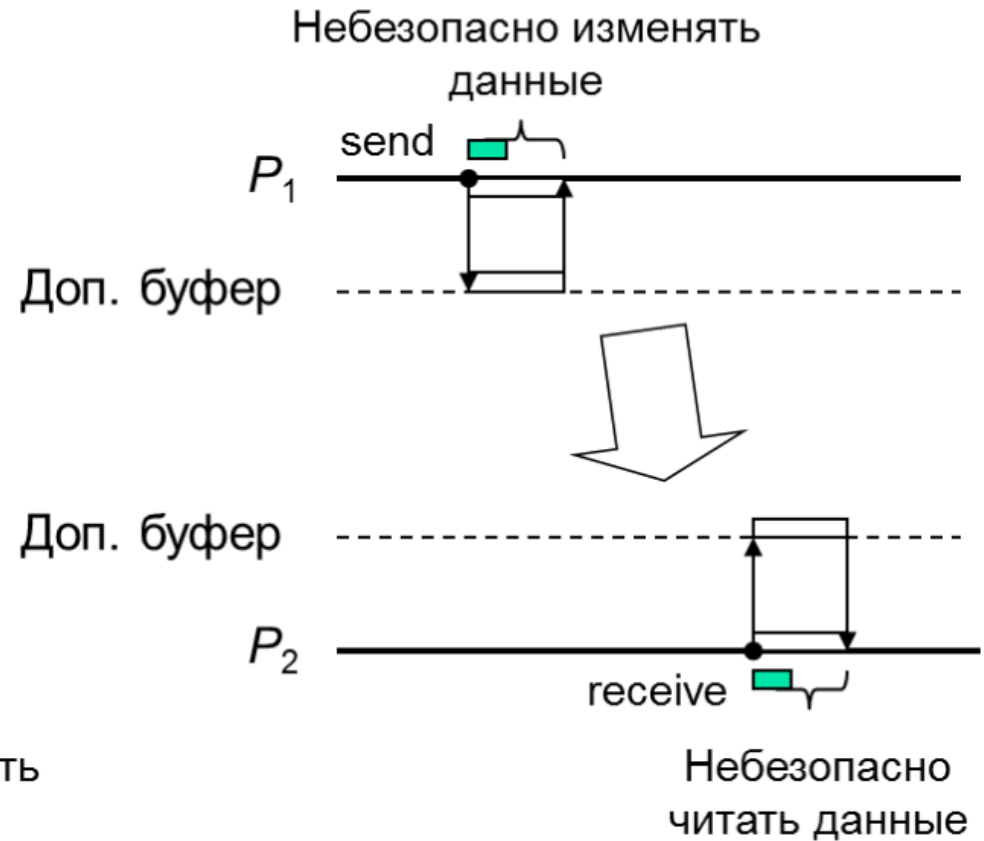
Process 1	Process 2
<code>receive(&a, process2)</code> <code>send(&b, process2)</code>	<code>receive(&a, process1)</code> <code>send(&b, process1)</code>

Process 1	Process 2
<code>send(&a, process2)</code> <code>receive(&b, process2)</code>	<code>send(&a, process1)</code> <code>receive(&b, process1)</code>

Неблокирующие отправка и получение



(a)

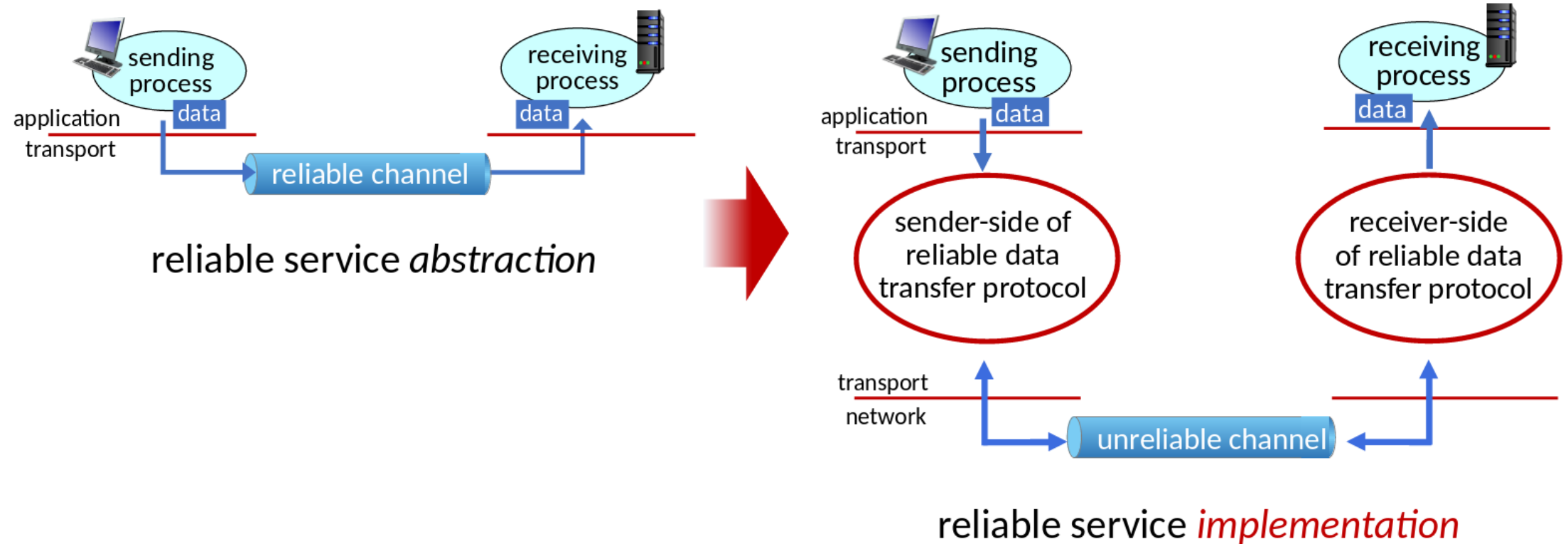


(б)

Синхронные и асинхронные взаимодействия

- Блокирующие и неблокирующие операции описывают *локальное* поведение вызывающего процесса (без учета других процессов)
- Можно рассматривать взаимодействие также с *глобальной* точки зрения
- Синхронный обмен сообщениями
 - отправитель и получатель ждут друг друга для передачи каждого сообщения
 - операция отправки считается завершенной только после того, как получатель закончит прием сообщения
- Асинхронный обмен сообщениями
 - не происходит никакой координации между отправителем и получателем сообщения

Надежная передача сообщений

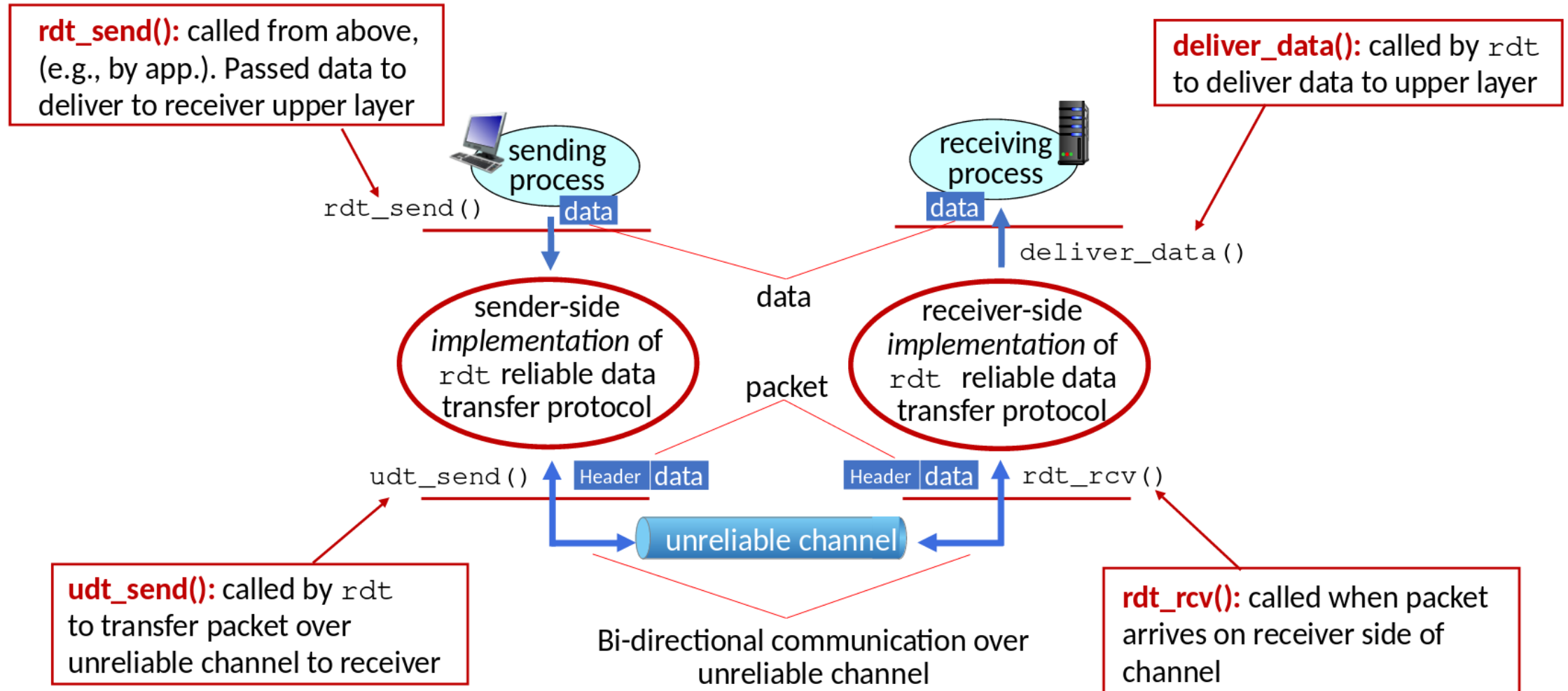


На основе материалов [Jim Curose. Computer Networks: Principles of Reliable Data Transfer](#)

Надежная передача сообщений

- Сеть представляет собой **ненадежный канал** передачи данных
 - Сообщения могут искажаться и теряться в процессе передачи
 - Будем считать, что канал не переупорядочивает сообщения
- Как поверх ненадежного канала реализовать абстракцию **надежного канала**?
 - Односторонний канал (отправитель, получатель)
 - Все отправленные сообщения доставляются получателю
- Процессы не "видят" состояний друг друга (получено ли сообщение)
 - Единственный способ узнать - передать эту информацию в сообщении
 - Для этого нам понадобится описать **протокол**

Интерфейсы



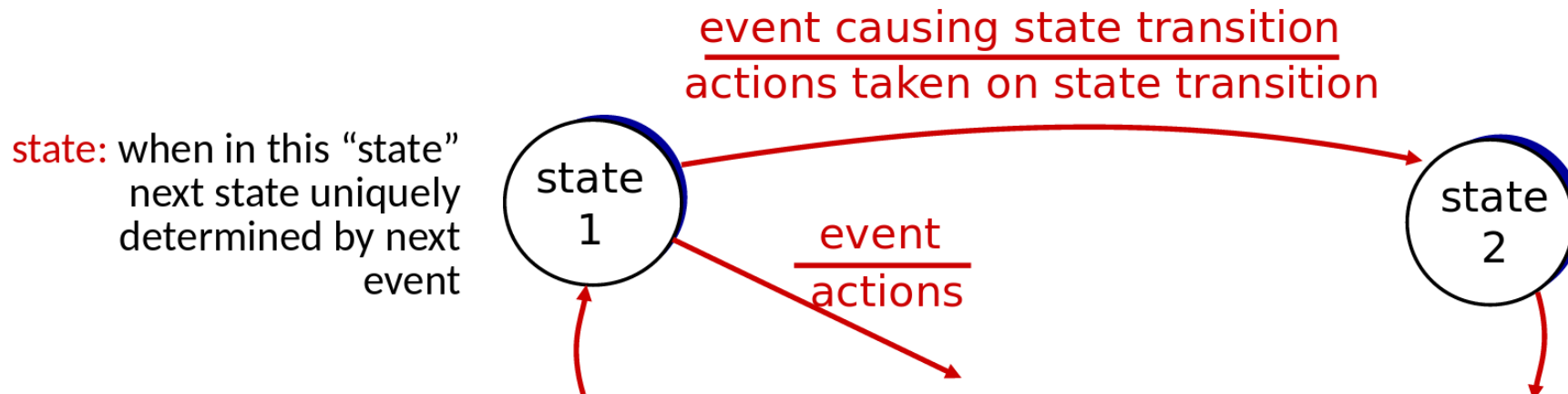
Протокол (передачи данных)

- Набор правил взаимодействия между компонентами системы (процессами, устройствами...)
- Типы, семантика и структура сообщений
- Форматы передачи данных
- Правила обработки сообщений
- Адресация компонентов
- Управление соединением
- Обнаружение и обработка ошибок
- ...

Описание протокола

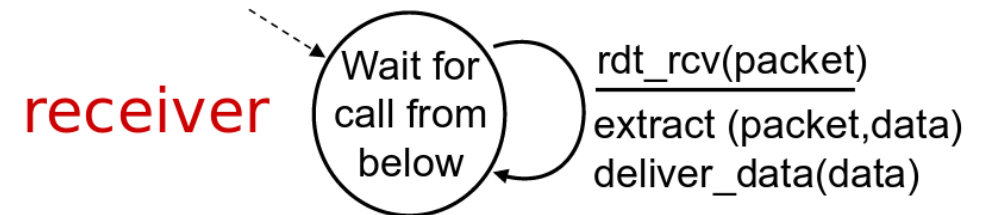
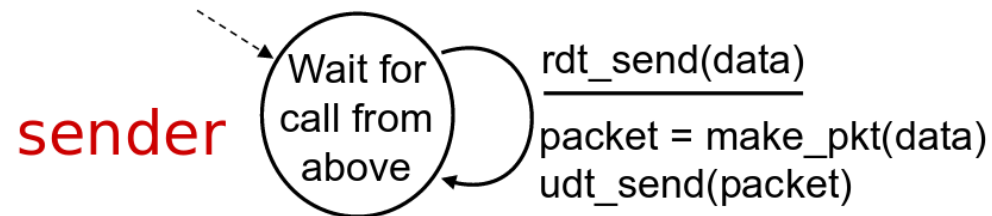
Будем использовать конечные автоматы (finite-state machine) для описания поведения отправителя и получателя в нашем протоколе

- Состояния, в которых может находиться процесс, в т.ч. начальное
- События, приводящие к переходу между состояниями
- Действия, выполняемые во время переходов между состояниями



Протокол 1.0

- Рассчитан на передачу сообщений по надежному каналу
 - сообщения передаются без ошибок и не теряются
- Конечные автоматы для отправителя и получателя:



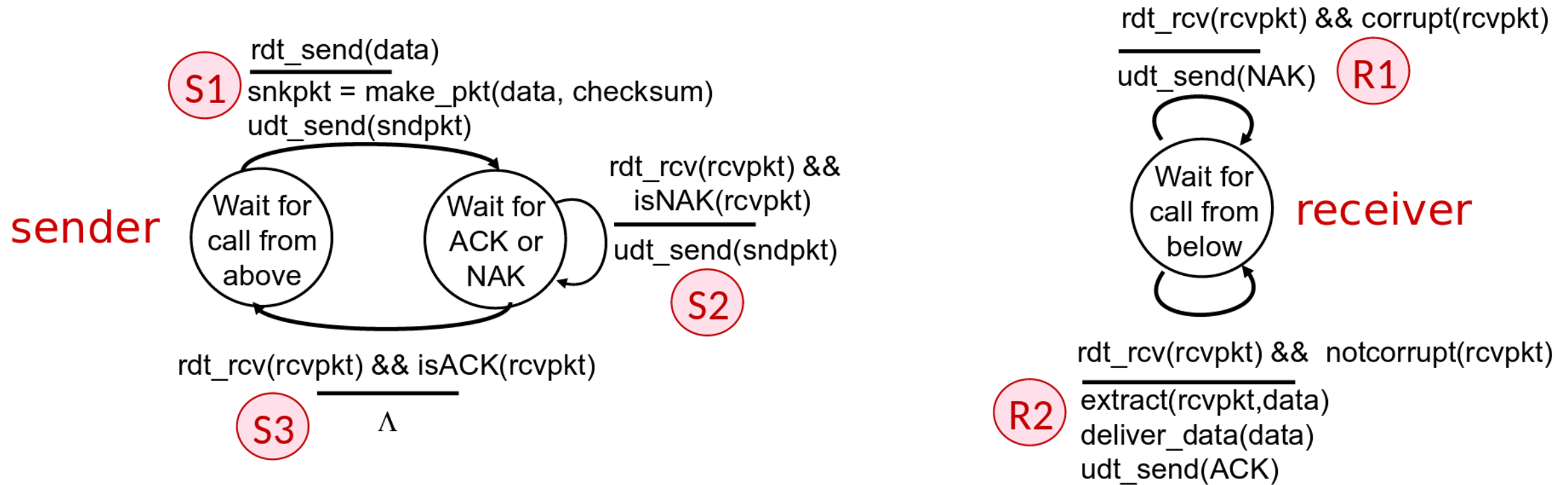
Канал с ошибками (bit errors)

- Теперь содержимое сообщения при передаче может повредиться
 - например, бит 1 вместо 0
- Как обнаруживать и обрабатывать такие ошибки?

Служебные сообщения

- ACK (acknowledgement)
 - сообщение получено в целостности
- NAK (negative acknowledgement)
 - сообщение получено с ошибками

Протокол 2.0



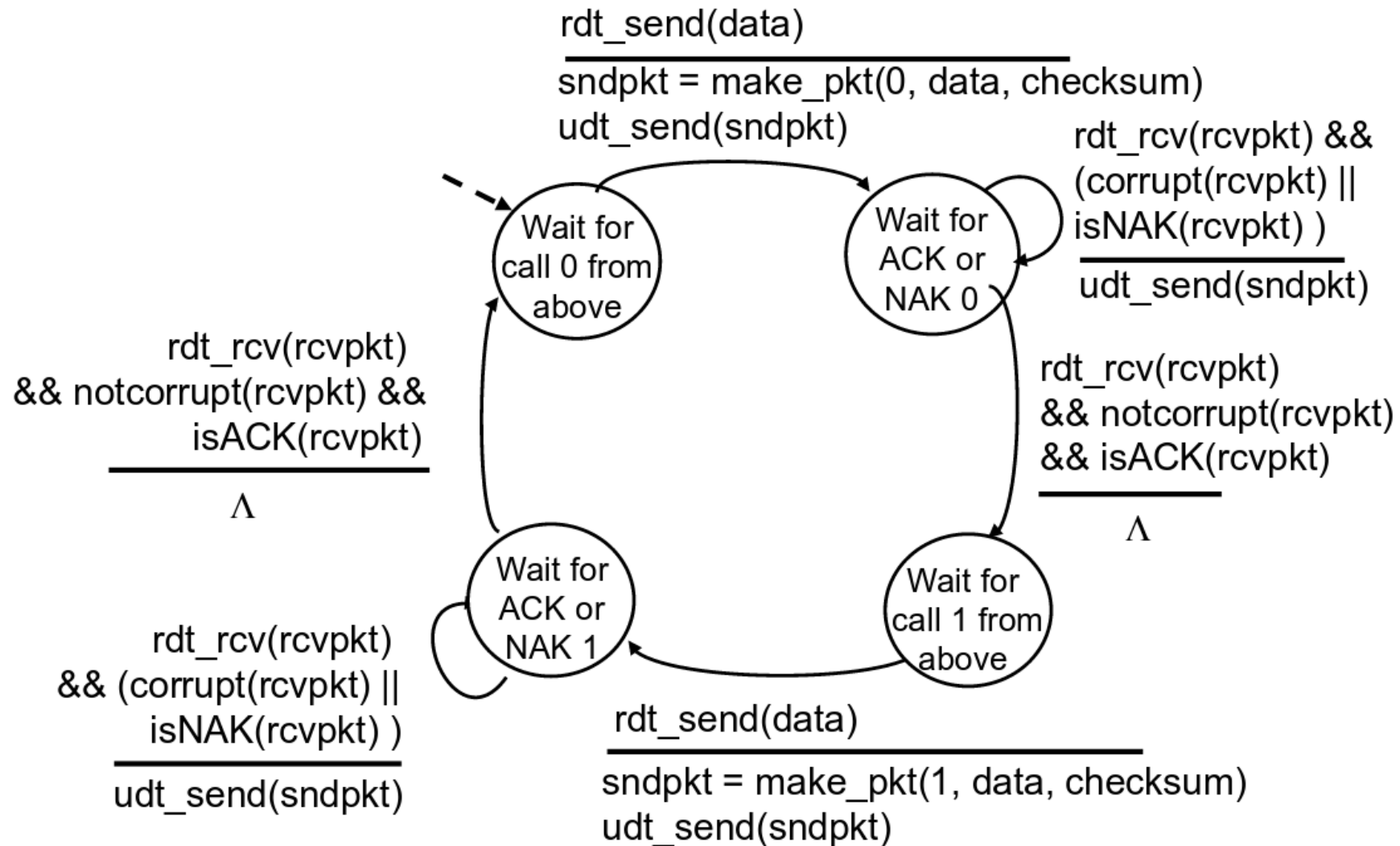
Проблема

- Что если окажется повреждено сообщение с ACK или NAK?
- Можно ли безопасно отправить исходное сообщение еще раз?

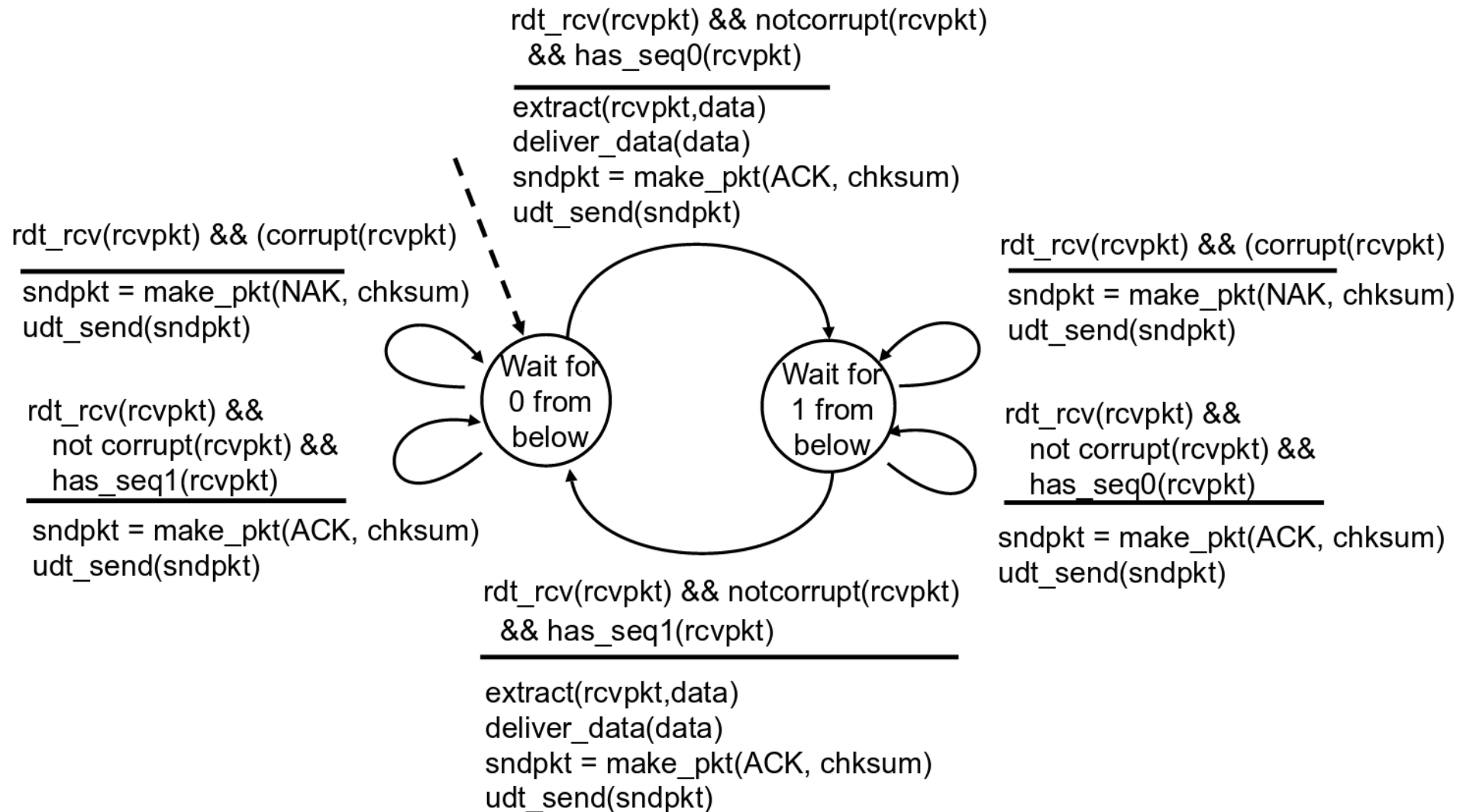
Дедупликация сообщений

- Отправитель добавляет в сообщение sequence number (SN)
 - В нашем случае достаточно чередовать 0 и 1
- Получатель ожидает сообщения с определенным SN
 - Сообщения с другим SN игнорируются (не доставляются выше)

Протокол 2.1: отправитель



Протокол 2.1: получатель



Упражнение

Протокол, который использует только ACKs (NAK-free)

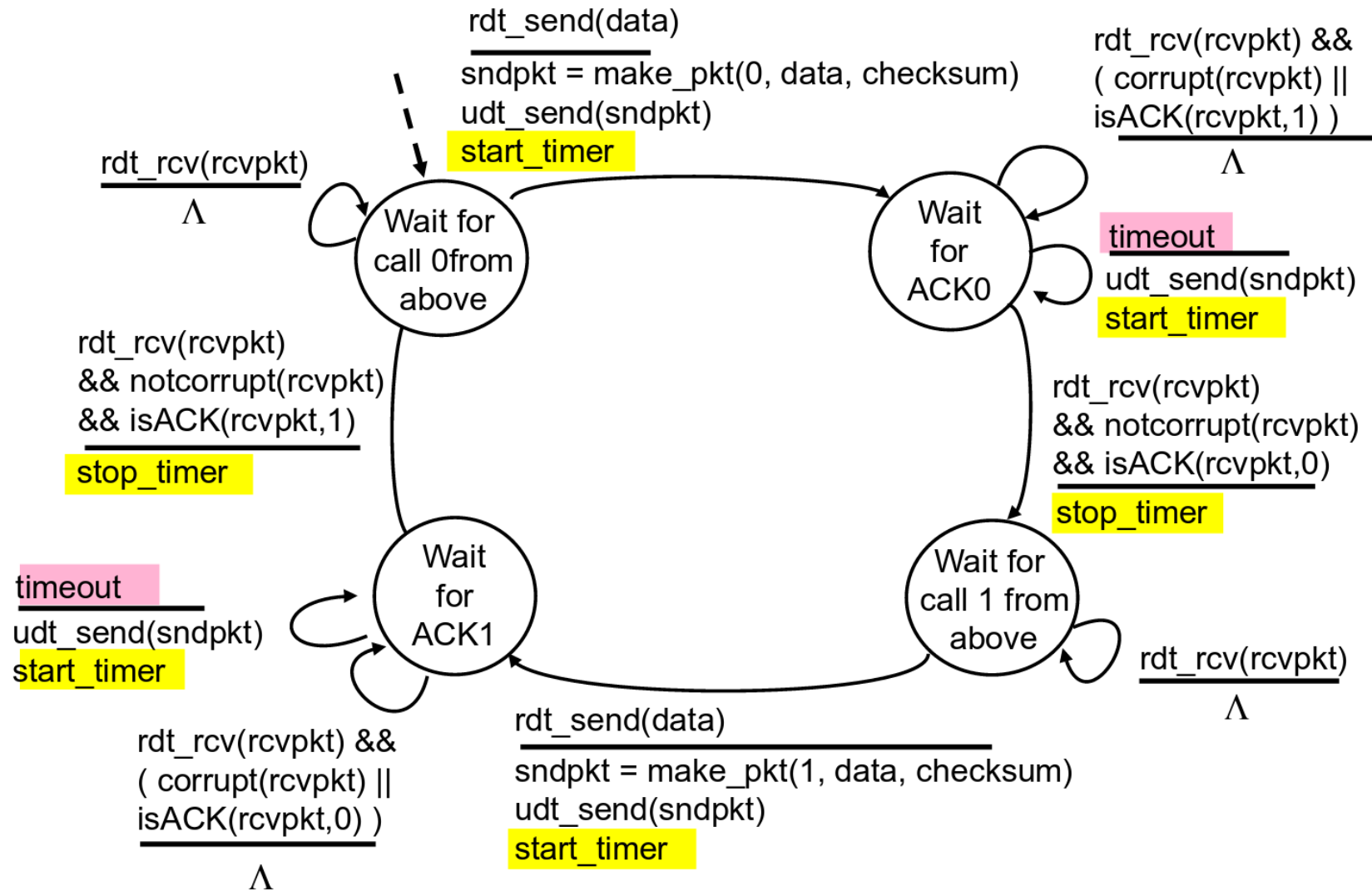
Канал с ошибками и потерей сообщений

- Теперь сообщения могут не только повреждаться, но и теряться в процессе передачи (packet loss)
- Как обнаруживать и обрабатывать потерю сообщения?

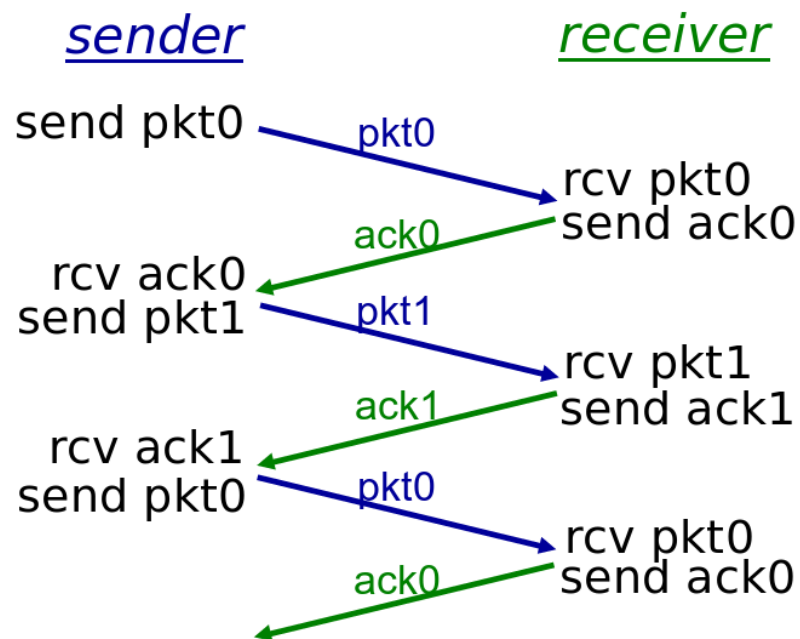
Базовые принципы

- Отправитель ожидает АСК в течение некоторого времени
 - требуется таймер на стороне отправителя
- Если АСК не получен вовремя, сообщение отправляется повторно
- Если сообщение (или АСК) не было потеряно, а задержалось
 - сообщение будет продублировано, но это решается с помощью SN
 - получатель должен указывать SN внутри АСК

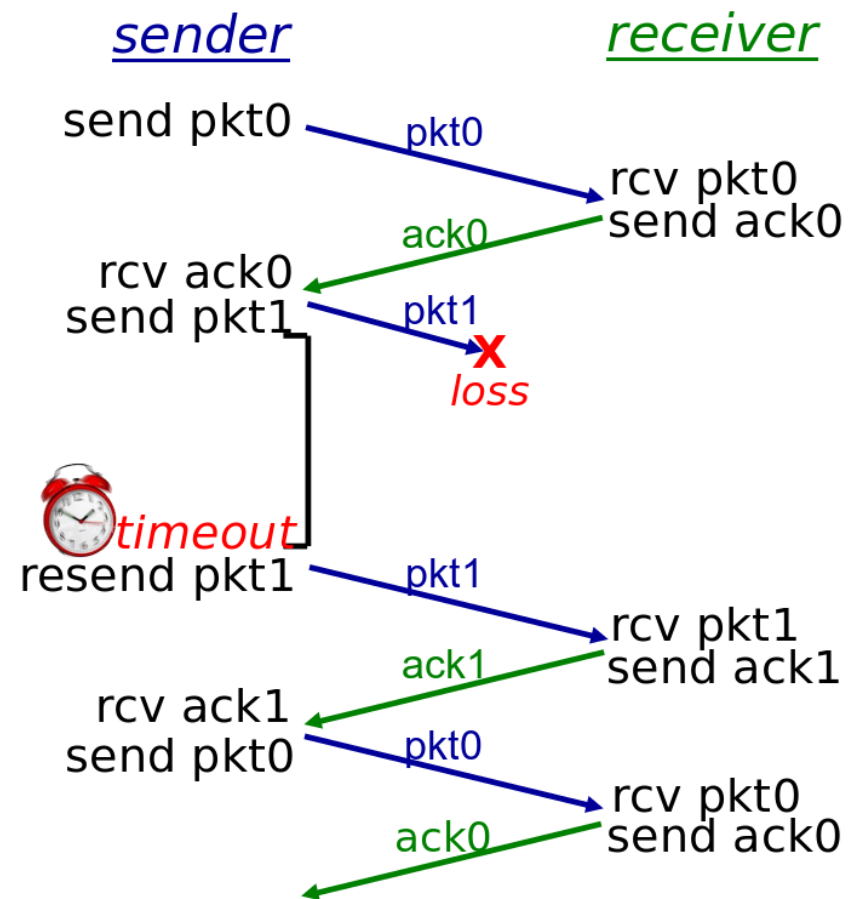
Протокол 3.0: отправитель



Возможные сценарии

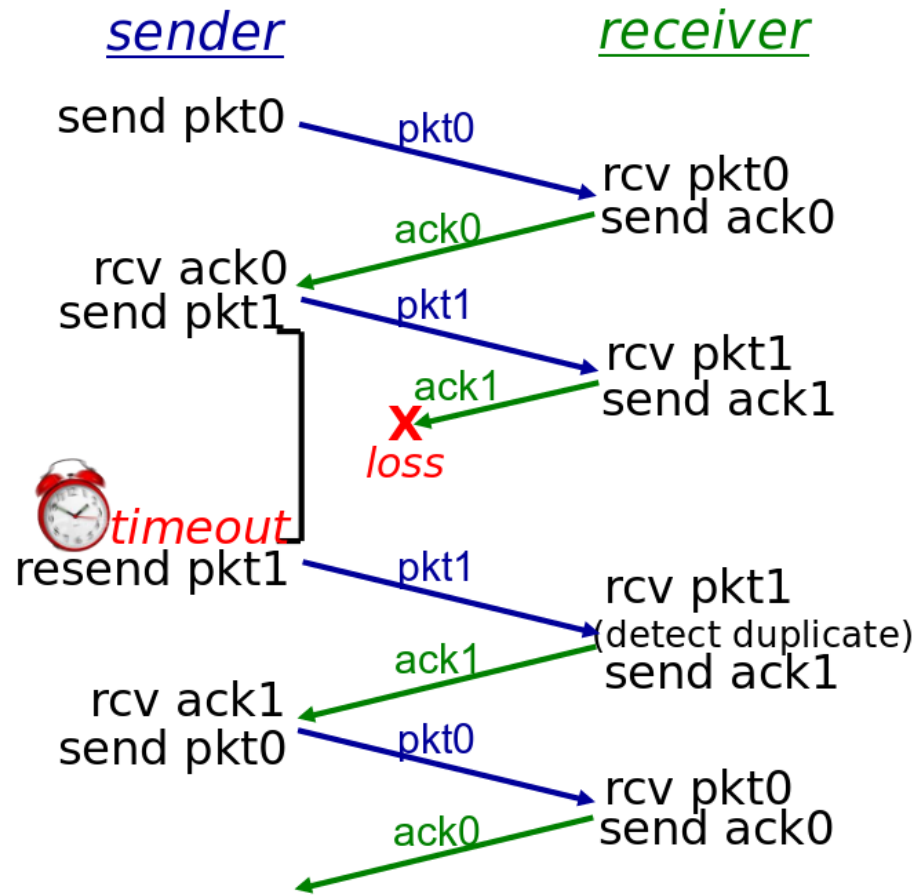


(a) no loss

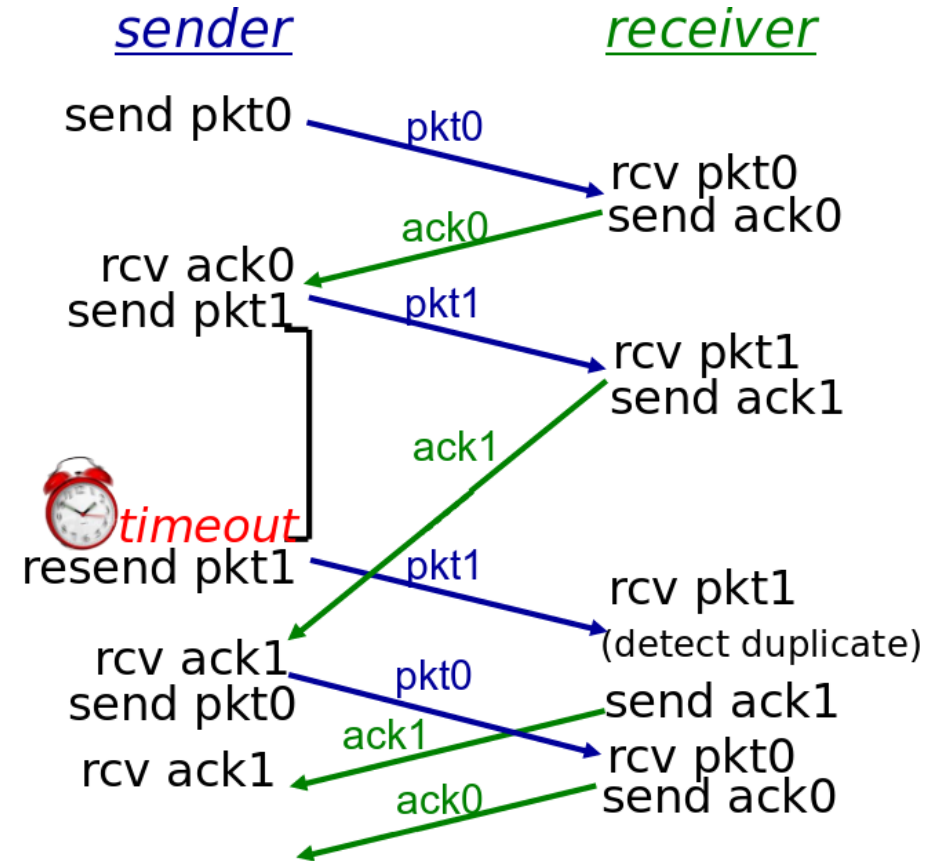


(b) packet loss

Возможные сценарии (2)



(c) ACK loss



(d) premature timeout/ delayed ACK

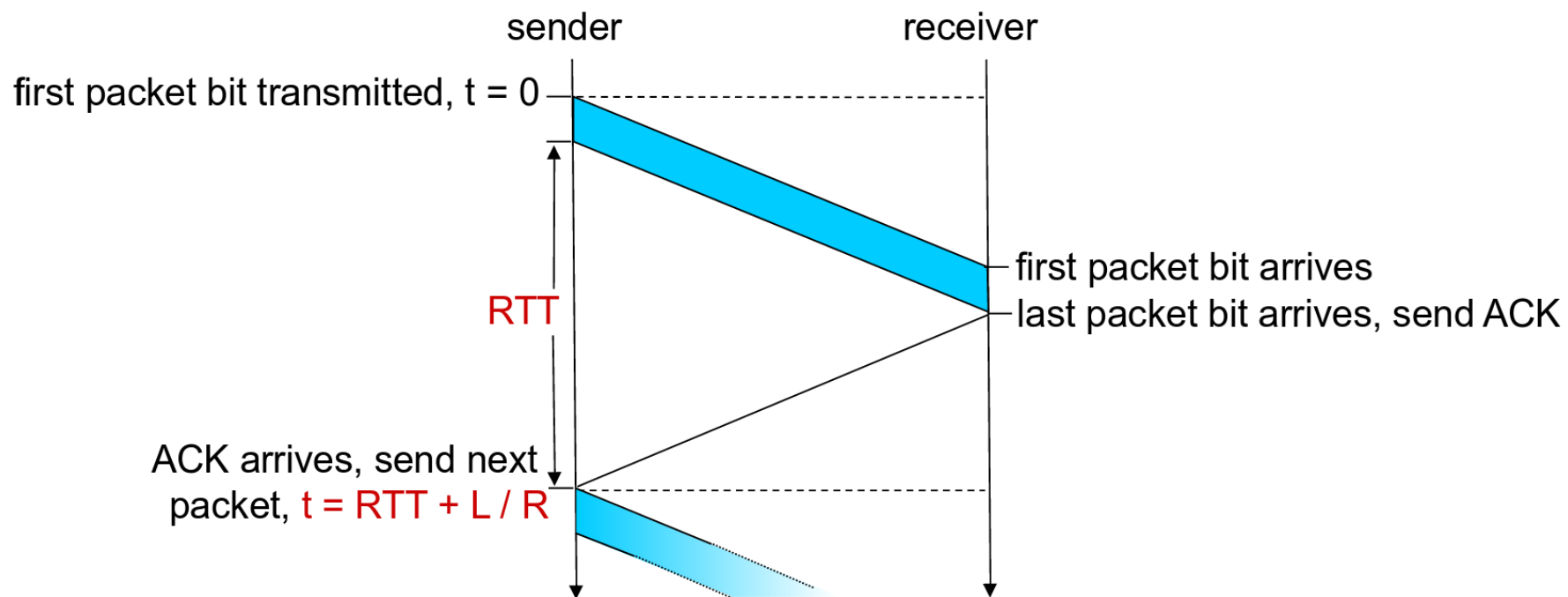
Переупорядочивание сообщений

- Что если канал (помимо повреждения и потерь) может также переупорядочивать сообщения в процессе передачи?
 - Сломает ли это наш протокол?
 - Приведите пример сценария

Возможные гарантии транспорта

- Контроль целостности сообщений
- Доставка сообщения
 - возможно будет доставлено, возможно несколько раз (zero or more, best effort)
 - будет доставлено не более 1 раза (at most once)
 - будет доставлено как минимум 1 раз (at least once)
 - будет доставлено ровно 1 раз (exactly once)
- Порядок доставки сообщений
 - в произвольном порядке
 - в порядке их отправки

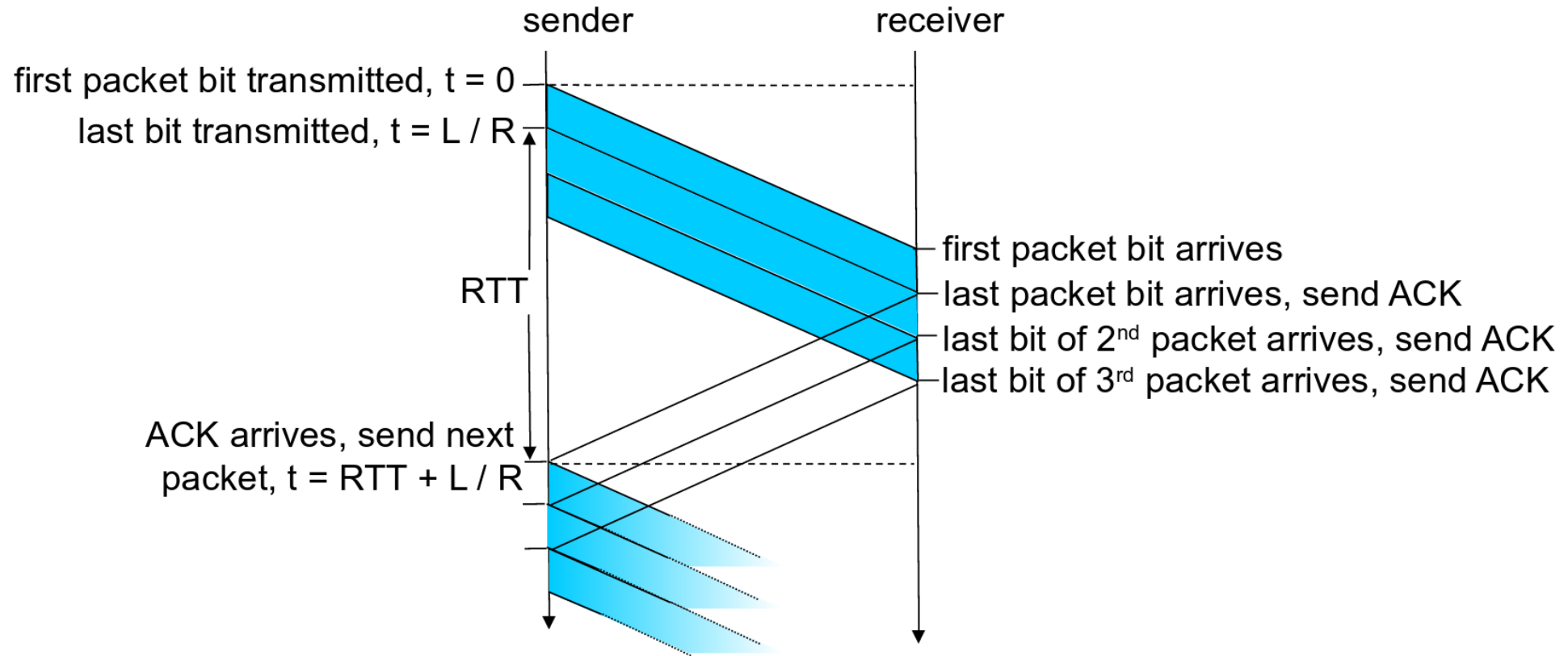
Производительность



RTT - время передачи сигнала туда и обратно,
 L - размер пакета, R - пропускная способность сети

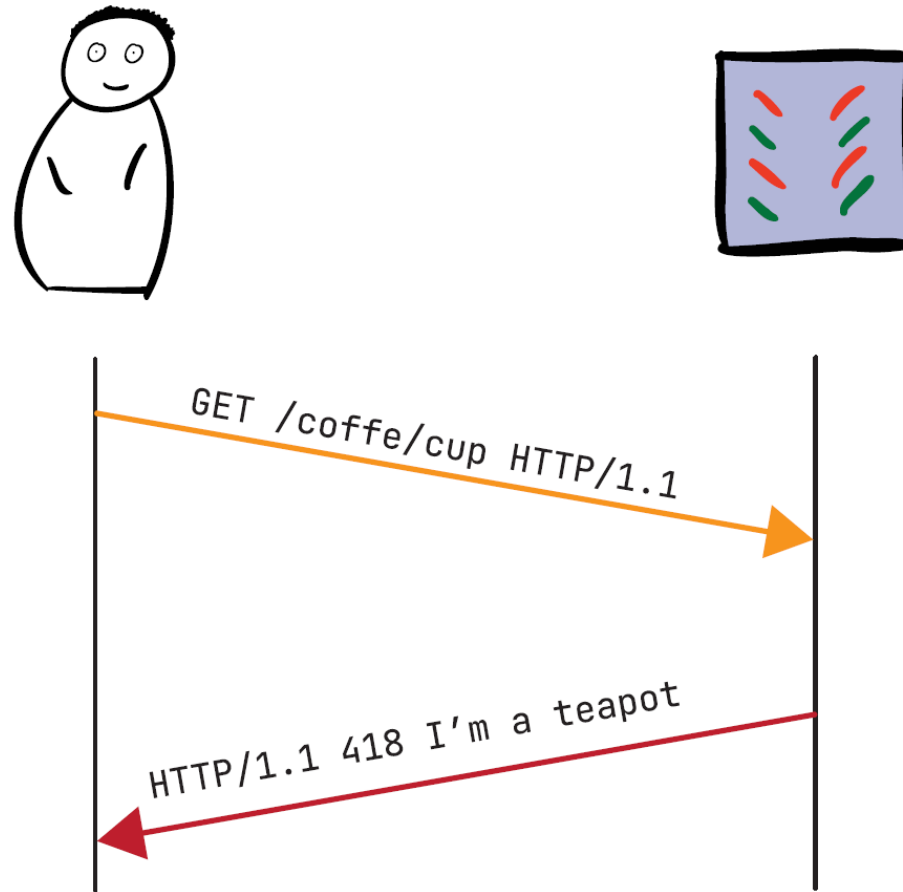
Передача по принципу stop-and-wait приводит к низкой утилизации сети

Pipelining

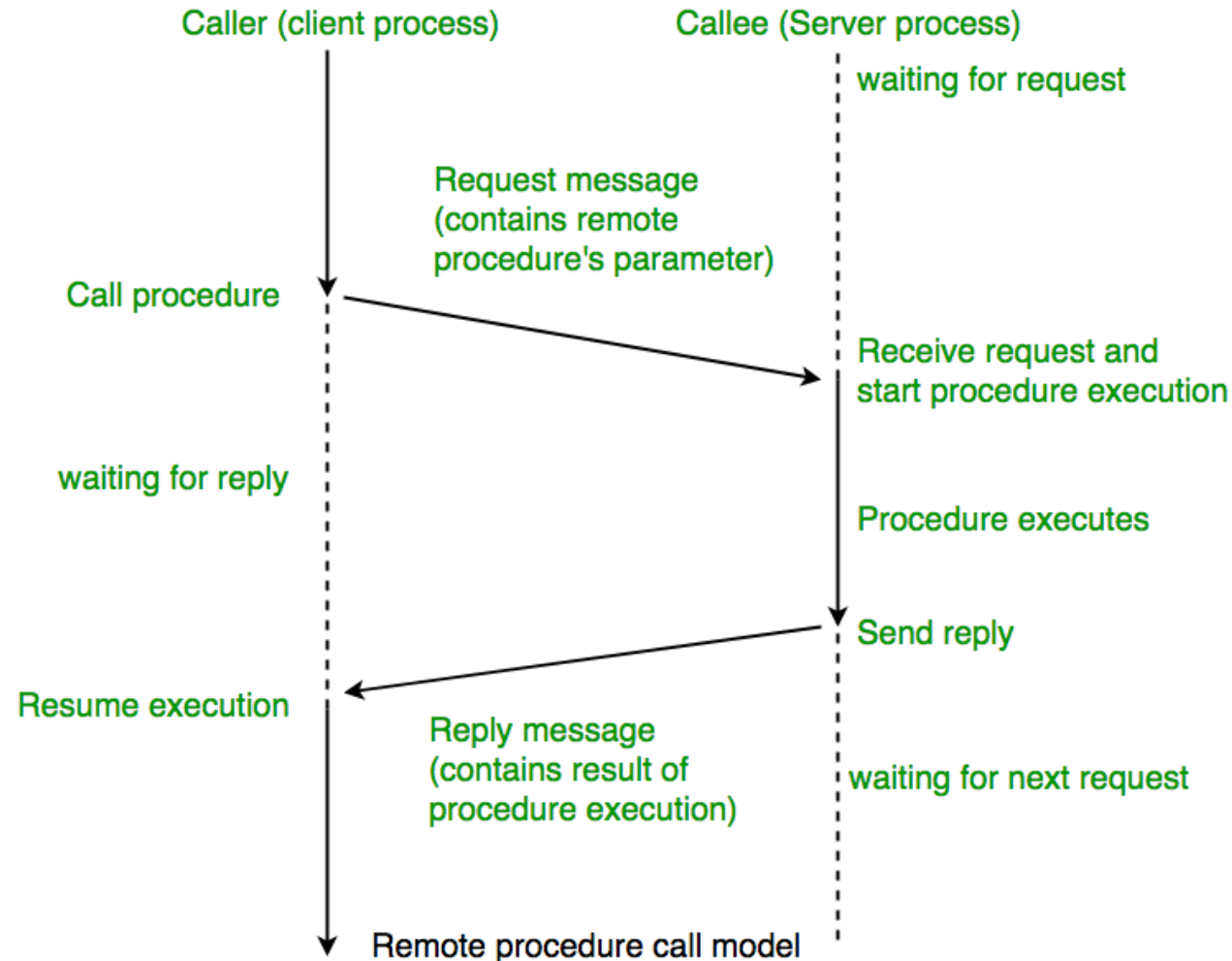


Отправка N пакетов за раз увеличивает утилизацию в N раз

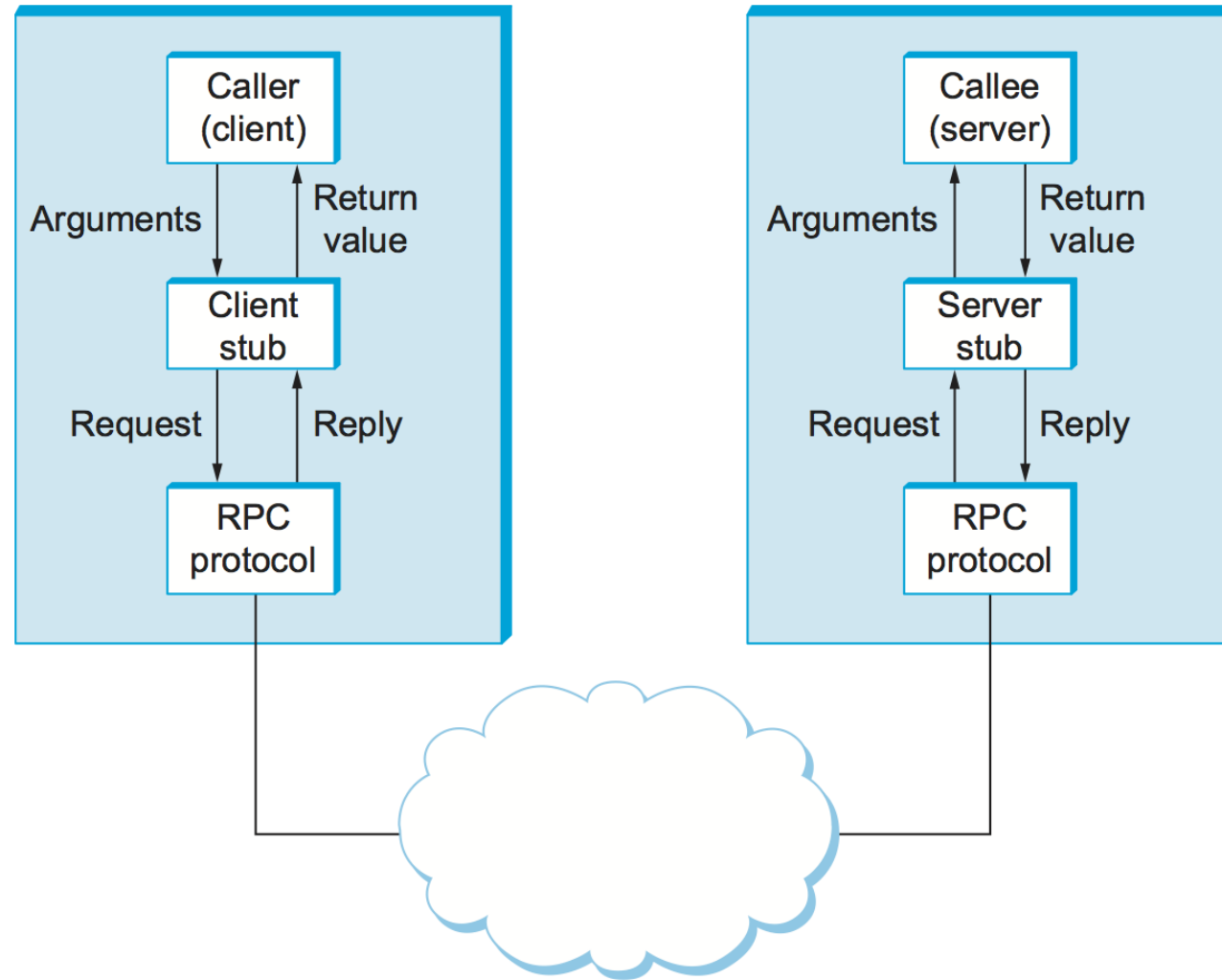
Схема взаимодействия "запрос-ответ"



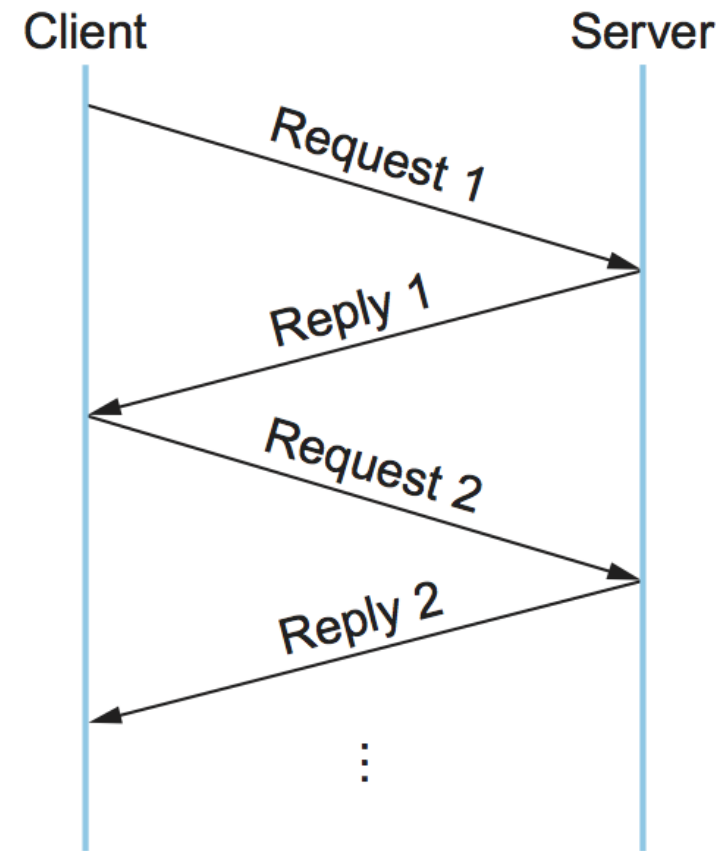
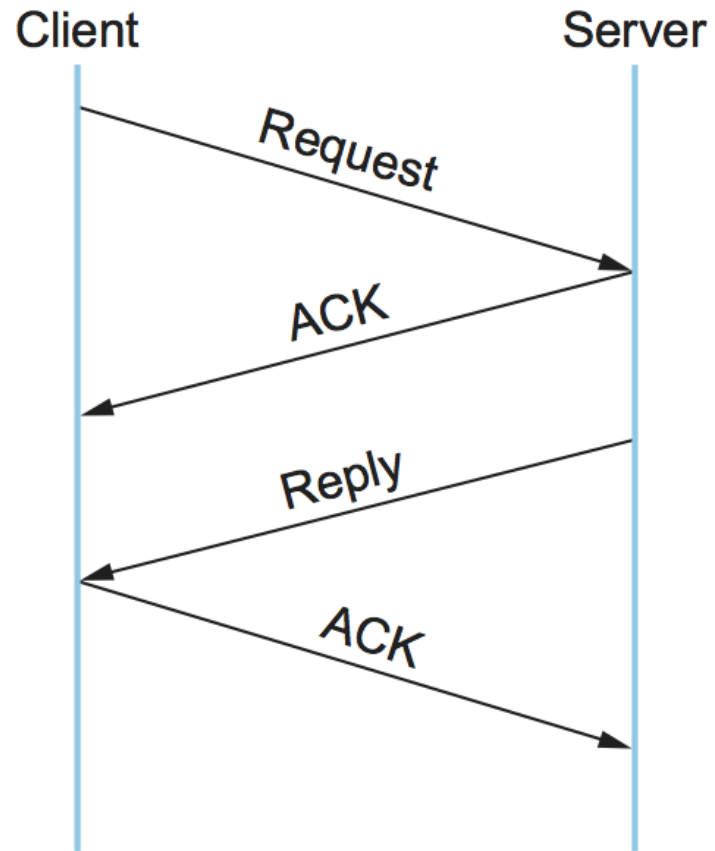
Remote Procedure Call (RPC)



Реализация RPC



Обеспечение надежности



Семантика (гарантии) RPC-вызова

- zero or more (best effort, maybe)
- at most once
- at least once
- exactly once

- ср. с семантикой **локального** вызова

Идемпотентные операции

- Результат операции не изменяется при многократном вызове
 - состояние сервера не изменяется или изменяется одинаковым образом
 - многократные вызовы эквивалентны однократному
- Примеры?
- В чем преимущество использования таких операций?

Отличия удаленных вызовов от локальных

- Производительность
- Выполняются удаленно, нет нагрузки на клиента
- Новые типы ошибок (исключений)
- Семантика вызова
- Передача параметров, адресное пространство
- Блокирующие вызовы и таймауты
- Что если сервер перезагрузился/обновил версию?
- Новые виды атак и угроз безопасности

Материалы к лекции

- [Distributed Systems: Principles and Paradigms](#) (разделы 4.1-4.3)
- [Computer Networking: A Top-Down Approach](#) (раздел 3.4)
- [Computer Networks: A Systems Approach](#) (раздел 5.3)

Дополнительно

- [Implementing Remote Procedure Calls](#) (1984)
- [A Critique of the Remote Procedure Call Paradigm](#) (1988)
- [A Note on Distributed Computing](#) (1994)
- [Another Note on Distributed Computing](#) (2008)
- [Mythbusting Remote Procedure Calls](#) (2012)
- [A Brief History of Distributed Programming: RPC](#) (2016)
- [Datacenter RPCs can be General and Fast](#) (2019)