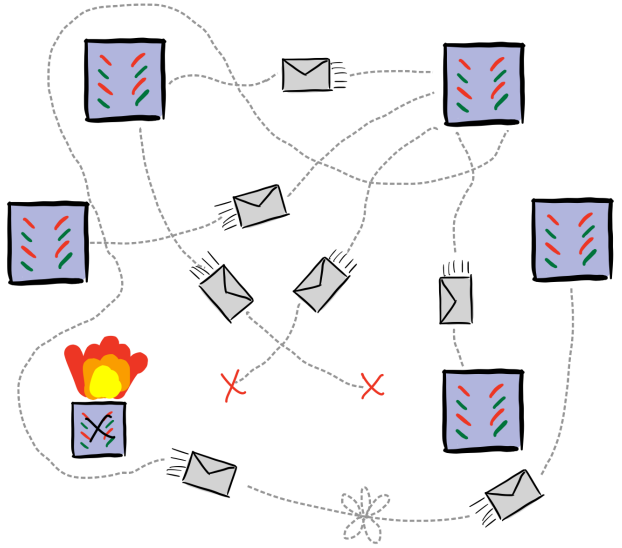
План

- Возможные варианты и схемы взаимодействия
- Передача сообщений между парой процессов
- Схема "запрос-ответ" и удаленные вызовы процедур (RPC)

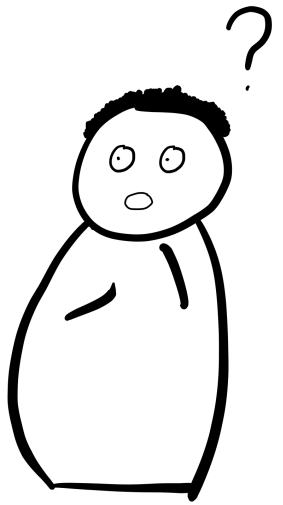
Обмен сообщениями (Message Passing)

- Наиболее общая и универсальная модель взаимодействия процессов в РС
- Сообщения передаются по ненадежным каналам, могут теряться и доставляться не в том порядке
- Возможны различные варианты взаимодействий и их реализации
- Реализация может предоставлять некоторые гарантии программисту



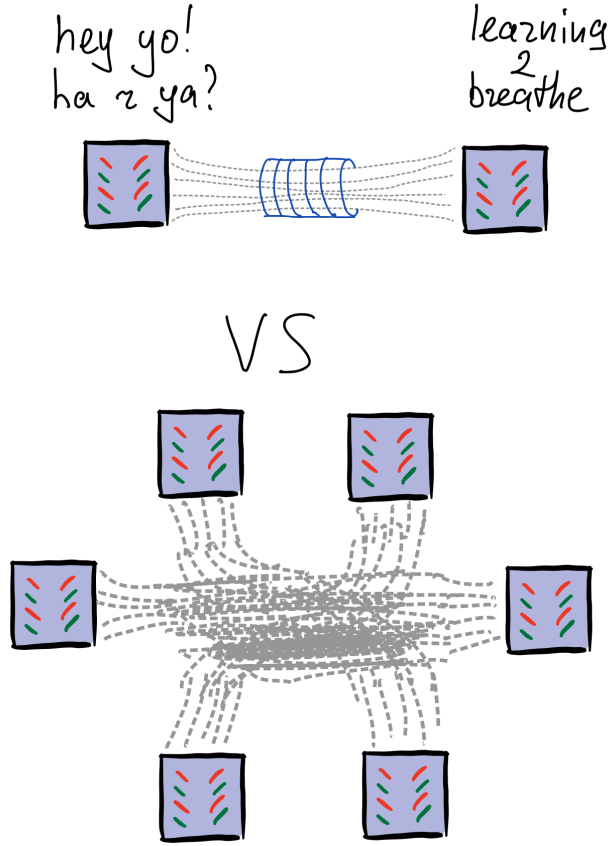
Варианты взаимодействий

- 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?
- Блокируются ли процессы во время отправки/приема сообщений?



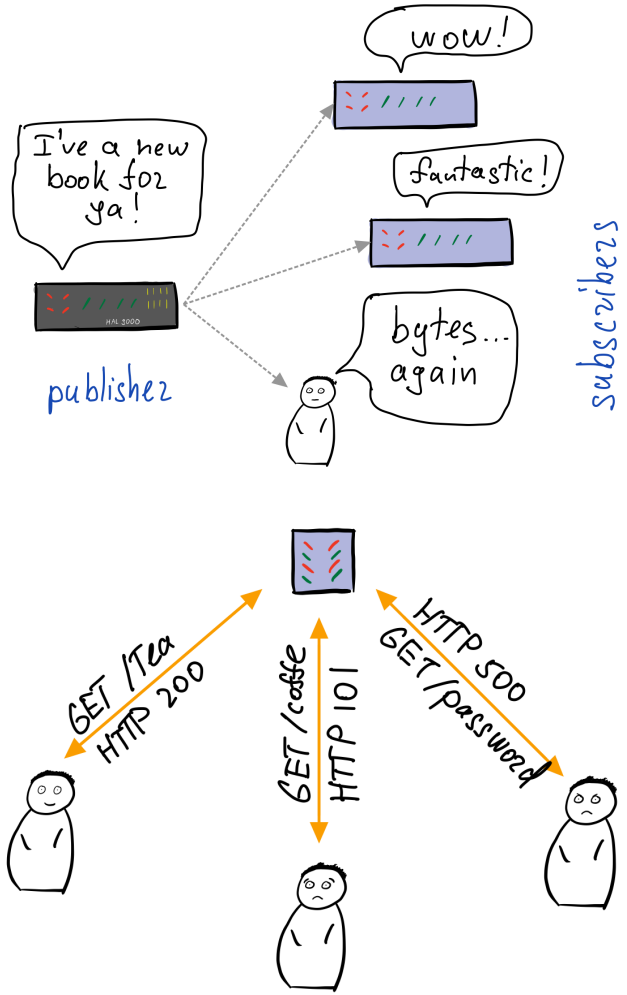
Число процессов

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



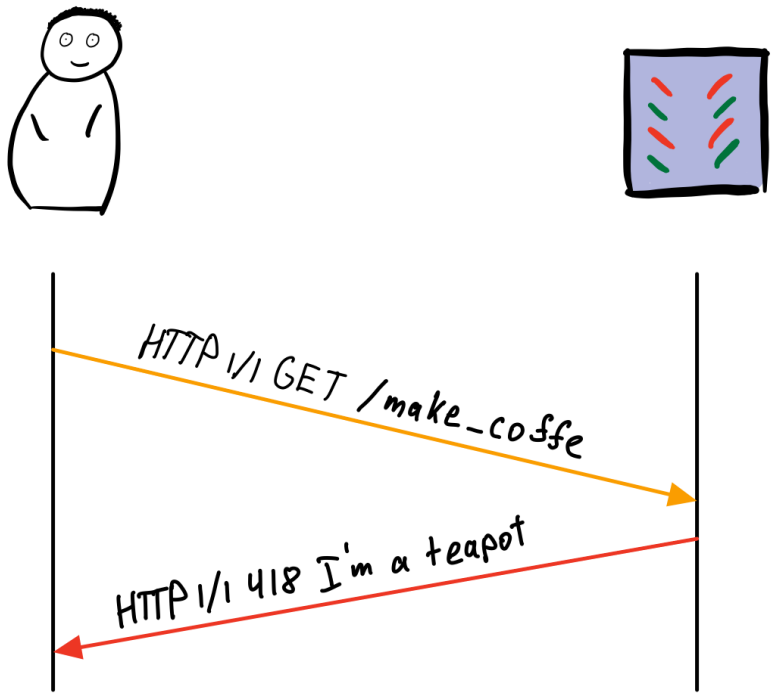
Направления передачи сообщений

- В одну сторону (unidirectional)
  - роли отправителей и получателей зафиксированы
  - пример: producer-consumer
- В обе стороны (bidirectional)
  - любой процесс может отправить сообщение другому
  - пример: клиент-сервер



Требуется ли ответ от получателя?

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



Кто инициирует передачу сообщения?

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

Связывание процессов

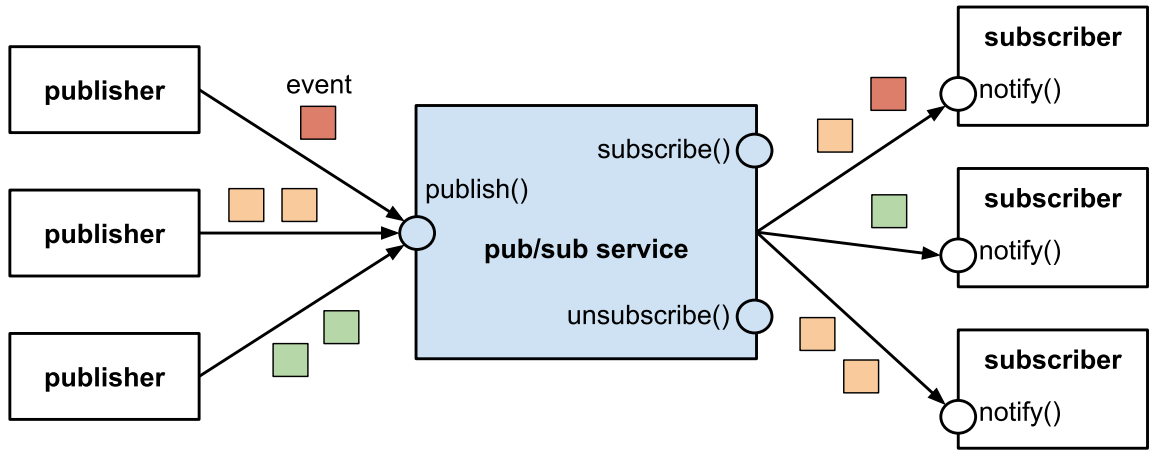
- Связывание по пространству (space coupling)
  - Процессы должны обладать информацией друг о друге
  - Например, отправитель должен знать адрес получателя
- Связывание по времени (time coupling)
  - Процессы должны выполняться в одно время
  - Transient vs persisent communication

Непрямое взаимодействие (indirect)

Происходит через некоторого посредника или абстракцию, без прямого связывания между отправителями и получателями

Примеры

- Очередь сообщений
- Издатель-подписчик
- Общая память

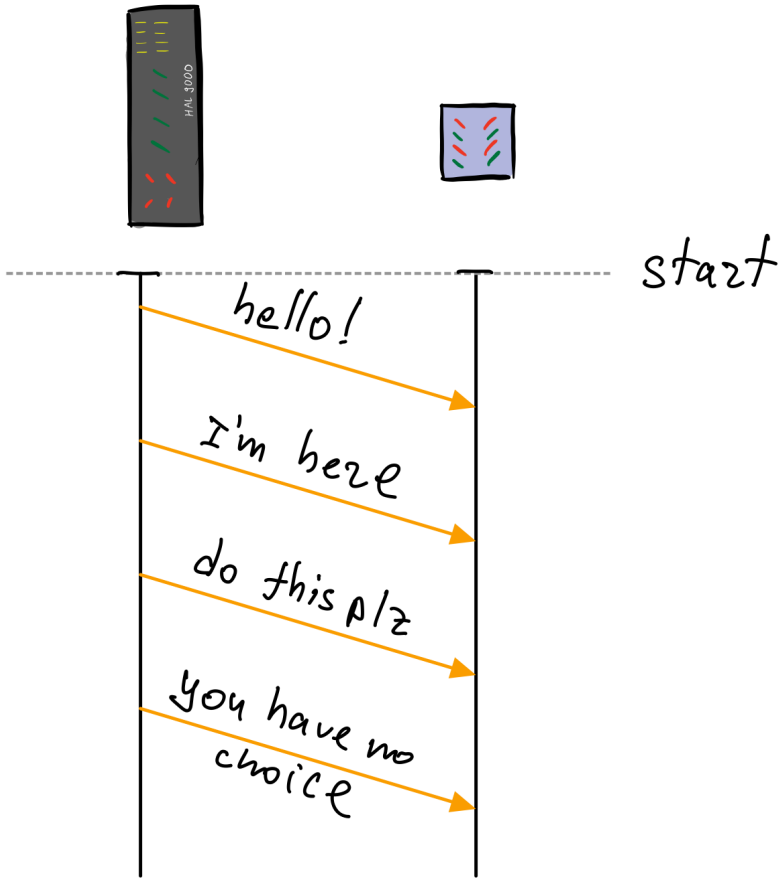


Блокируются ли процессы?

- Синхронное взаимодействие
  - отправитель блокируется (до приема сообщения к доставке, доставки сообщения получателю, окончания обработки сообщения...)
  - получатель блокируется до приема сообщения
- Асинхронное взаимодействие
  - отправитель продолжает выполнение сразу после отправки сообщения
  - получатель может неблокирующим образом проверить наличие сообщений

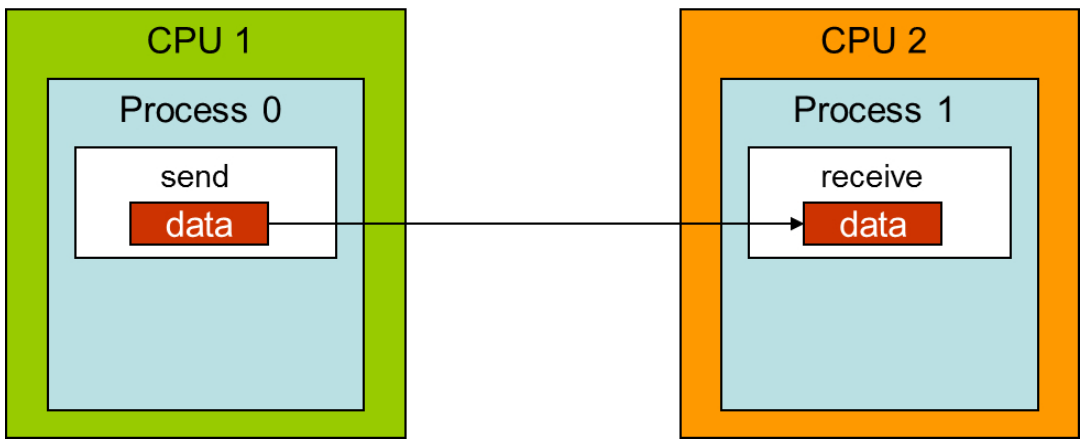
Простейший вариант

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



Реализация

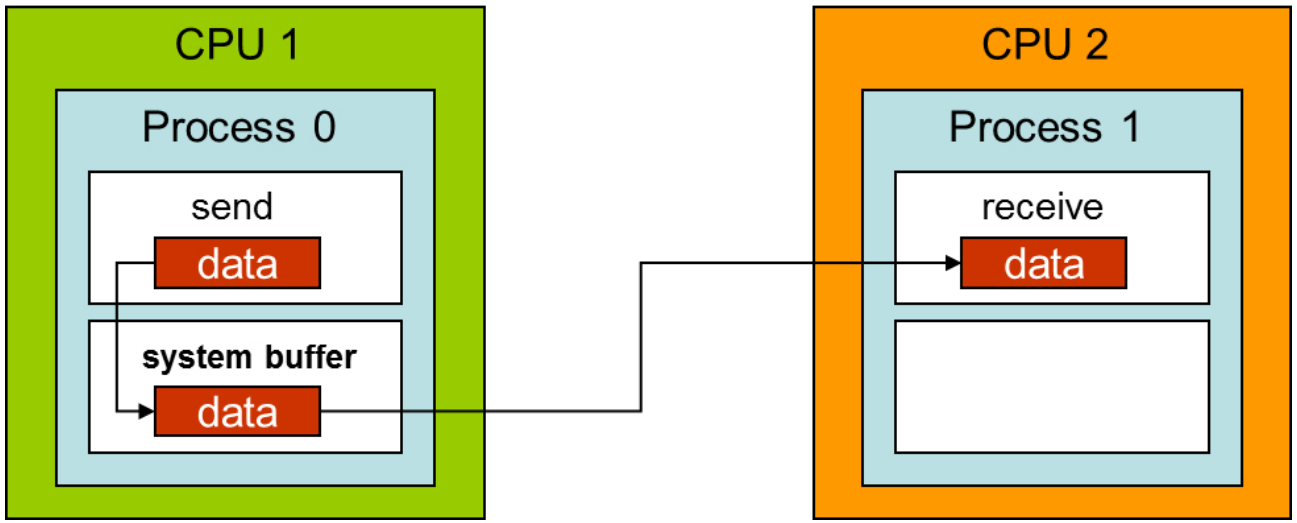
- send(buffer, dest)
  - блокируется до ...
- recv(buffer)
  - блокируется до получения сообщения и размещения его в буфере



Когда завершается send()?

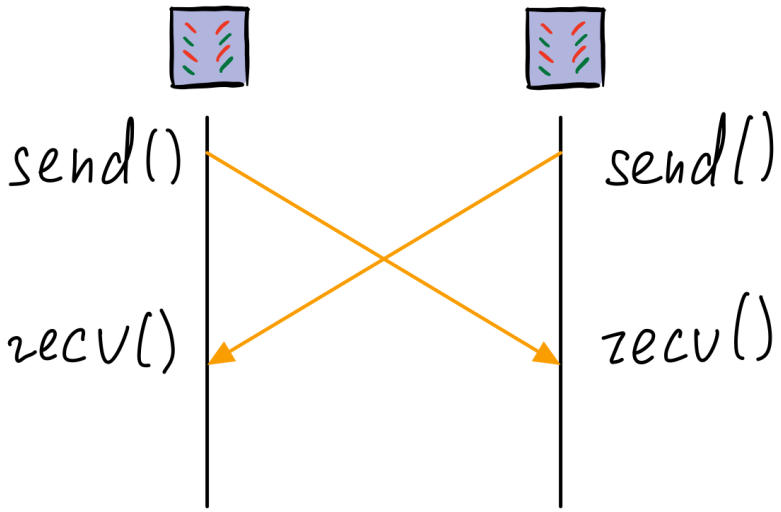
- Буфер можно повторно использовать, не опасаясь испортить передаваемое сообщение?
- Сообщение покинуло узел процесса-отправителя?
- Сообщение принято процессом-получателем?

Возможная реализация



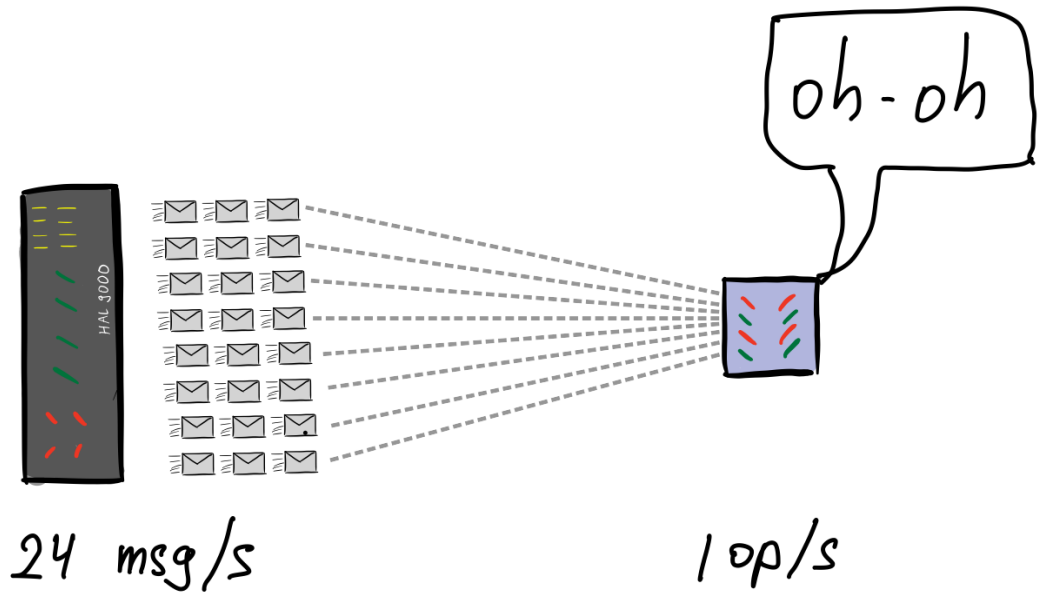
Проблема 1

Два процесса отправляют друг другу сообщения и хотят получить чужое сообщение

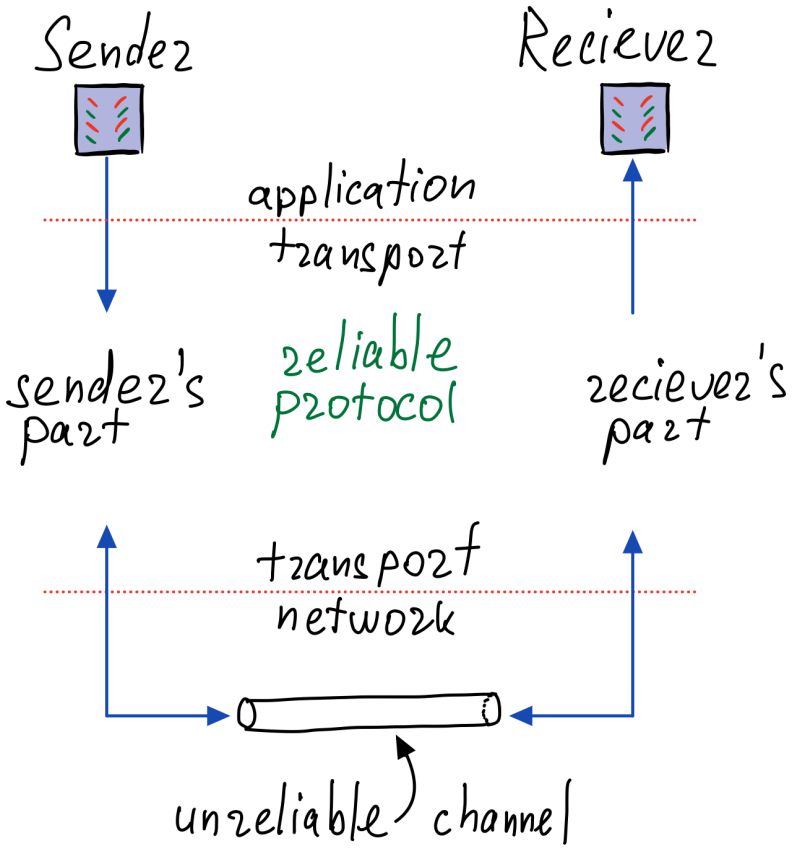


Проблема 2

Отправитель передает сообщения быстрее, чем получатель может их обрабатывать



Надежная передача сообщений



На основе материалов Jim Curose. Computer Networks: Principles of Reliable Data Transfer

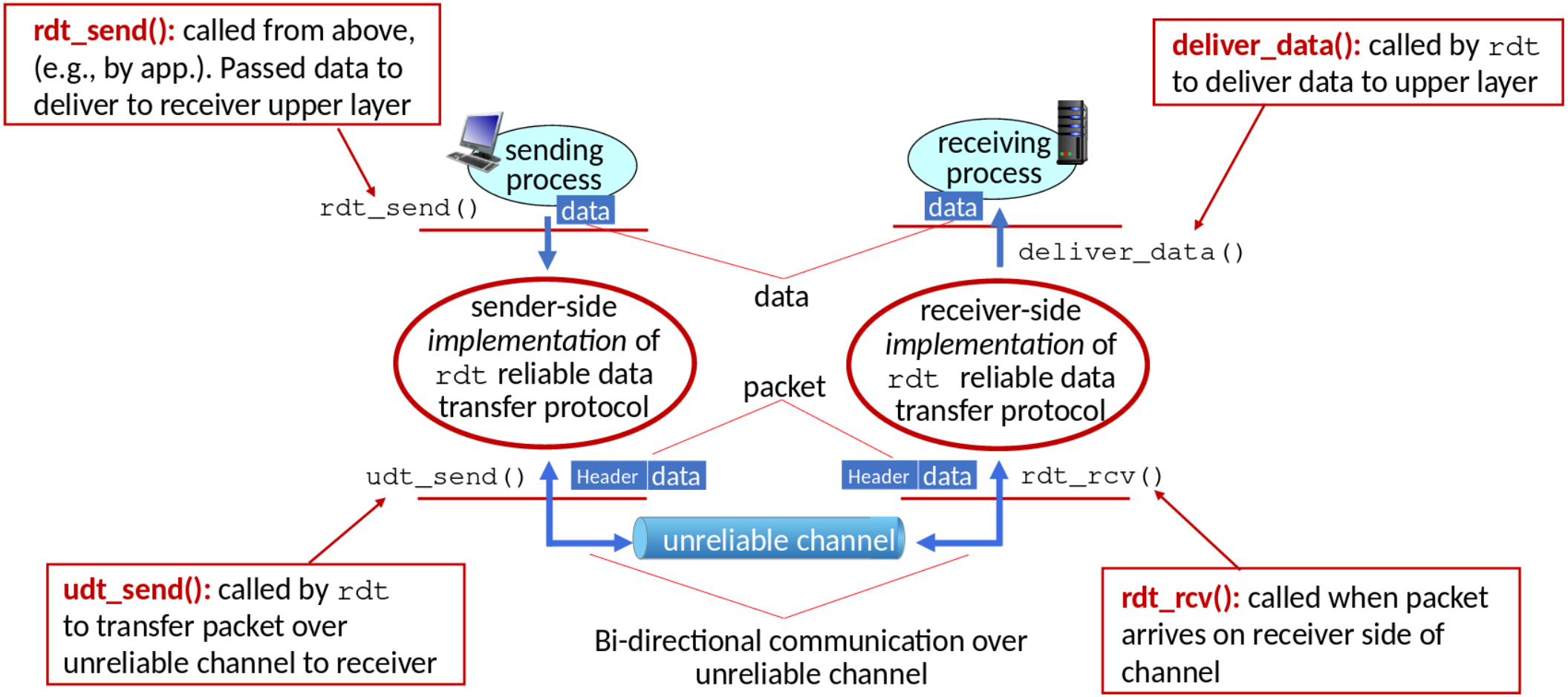
Надежная передача сообщений

- Сеть представляет собой ненадежный канал передачи данных
  - Сообщения могут теряться, искажаться, дублироваться, приходить в другом порядке
- 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

Возможные гарантии реализации

- 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

Интерфейсы

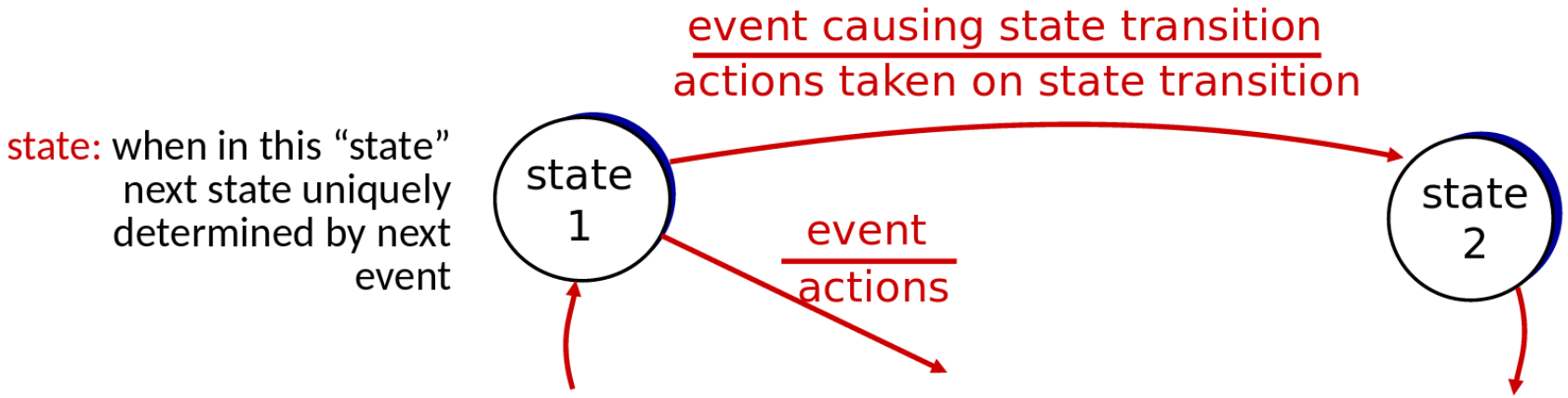

rdt_send(): called from above, (e.g., by app.). Passed data to deliver to receiver upper layer
deliver_data(): called by rdt to deliver data to upper layer
sending process
receiving process
data
rdt_send()
deliver_data()
data
sender-side implementation of rdt reliable data transfer protocol
receiver-side implementation of rdt reliable data transfer protocol
packet
udt_send()
Header
data
rdt_rcv()
unreliable channel
udt_send(): called by rdt to transfer packet over unreliable channel to receiver
Bi-directional communication over unreliable channel
rdt_rcv(): called when packet arrives on receiver side of channel

Протокол

- Описание правил взаимодействия компонентов системы
- Типы, семантика и структура сообщений
- Форматы передачи данных
- Правила обработки сообщений
- Адресация компонентов
- Управление соединением
- Обнаружение и обработка ошибок
- ...

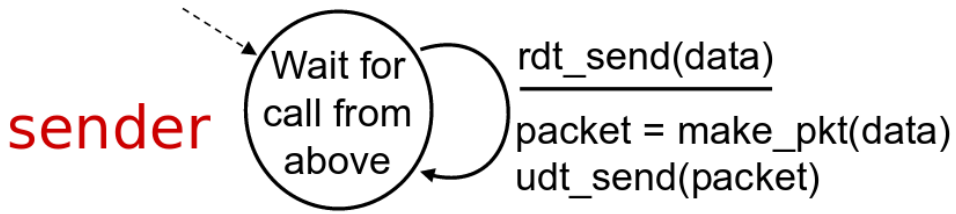
Описание протокола

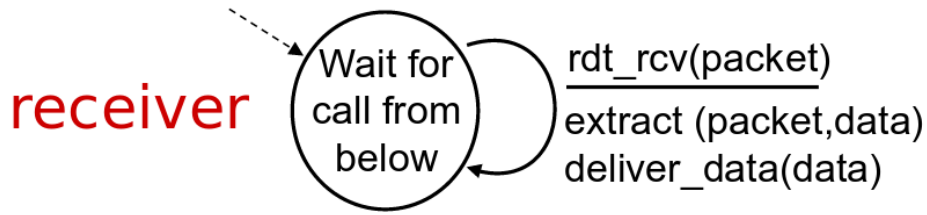
- Будем использовать конечные автоматы для описания поведения отправителя и получателя в нашем протоколе
  - Состояния, в которых может находиться процесс
  - События, приводящие к переходу между состояниями
  - Действия, выполняемые во время переходов между состояниями



Протокол 1.0

- Низлежащий канал - надежный
  - сообщения передаются без ошибок (bit errors)
  - сообщения не теряются (packet loss)
- Конечные автоматы для отправителя и получателя





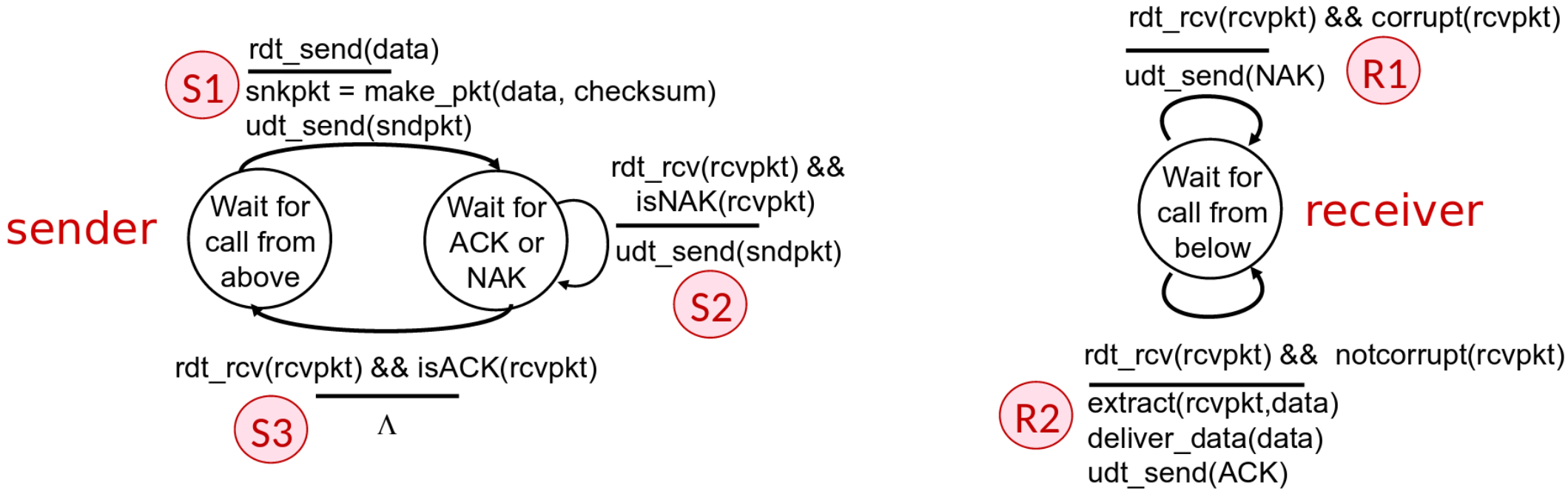
Канал с ошибками

- 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
- NACK (negative acknowledgement)
  - сообщение получено с ошибками

Протокол 2.0


sender
S1
rdt_send(data)
snkpkt = make_pkt(data, checksum)
udt_send(sndpkt)
Wait for call from above
Wait for ACK or NAK
rdt_rcv(rcvpkt) && isNAK(rcvpkt)
udt_send(sndpkt)
S2
rdt_rcv(rcvpkt) && isACK(rcvpkt)
S3
Λ
rdt_rcv(rcvpkt) && corrupt(rcvpkt)
udt_send(NAK)
R1
Wait for call from below
receiver
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
R2
extract(rcvpkt,data)
deliver_data(data)
udt_send(ACK)

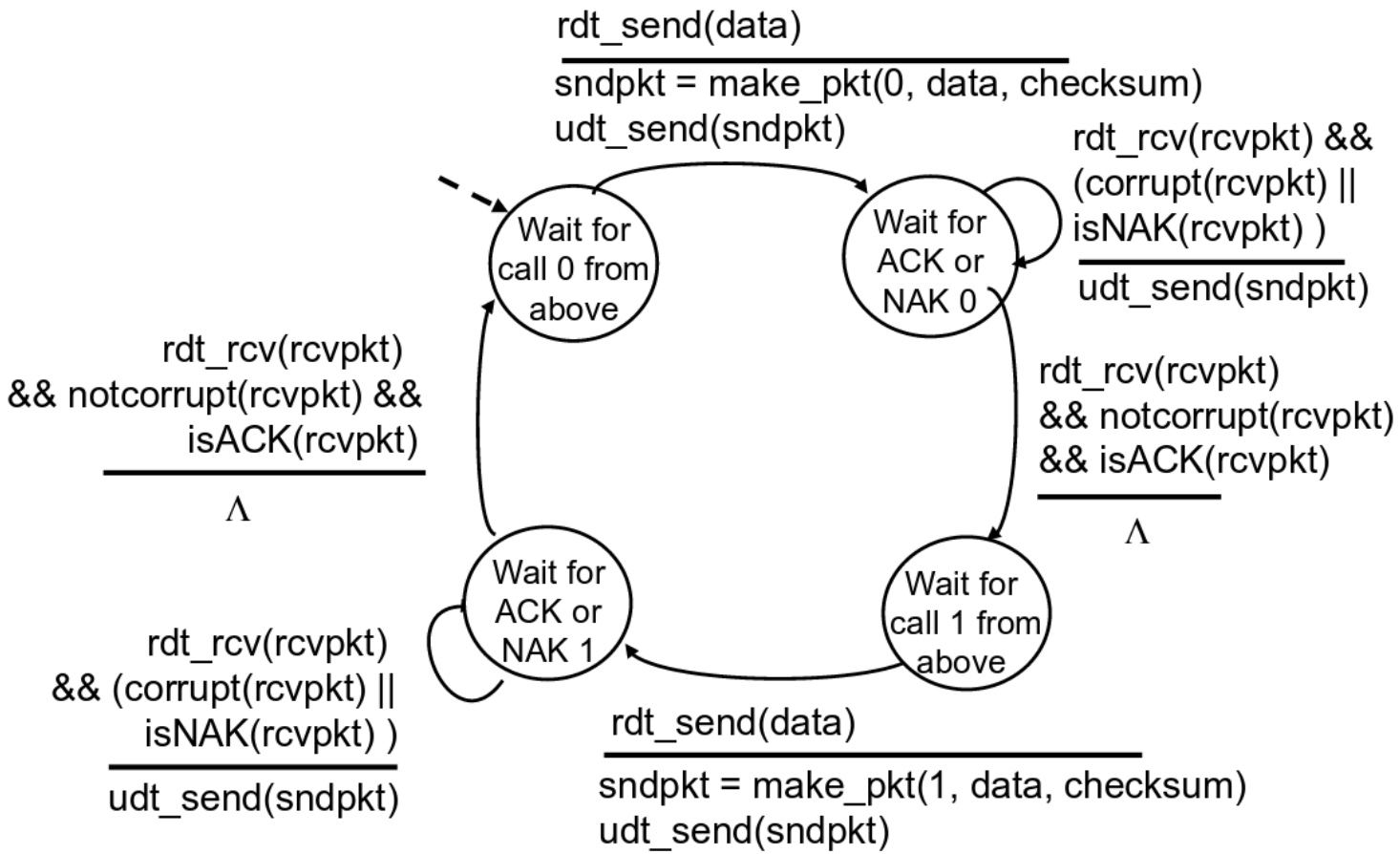
Проблема

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

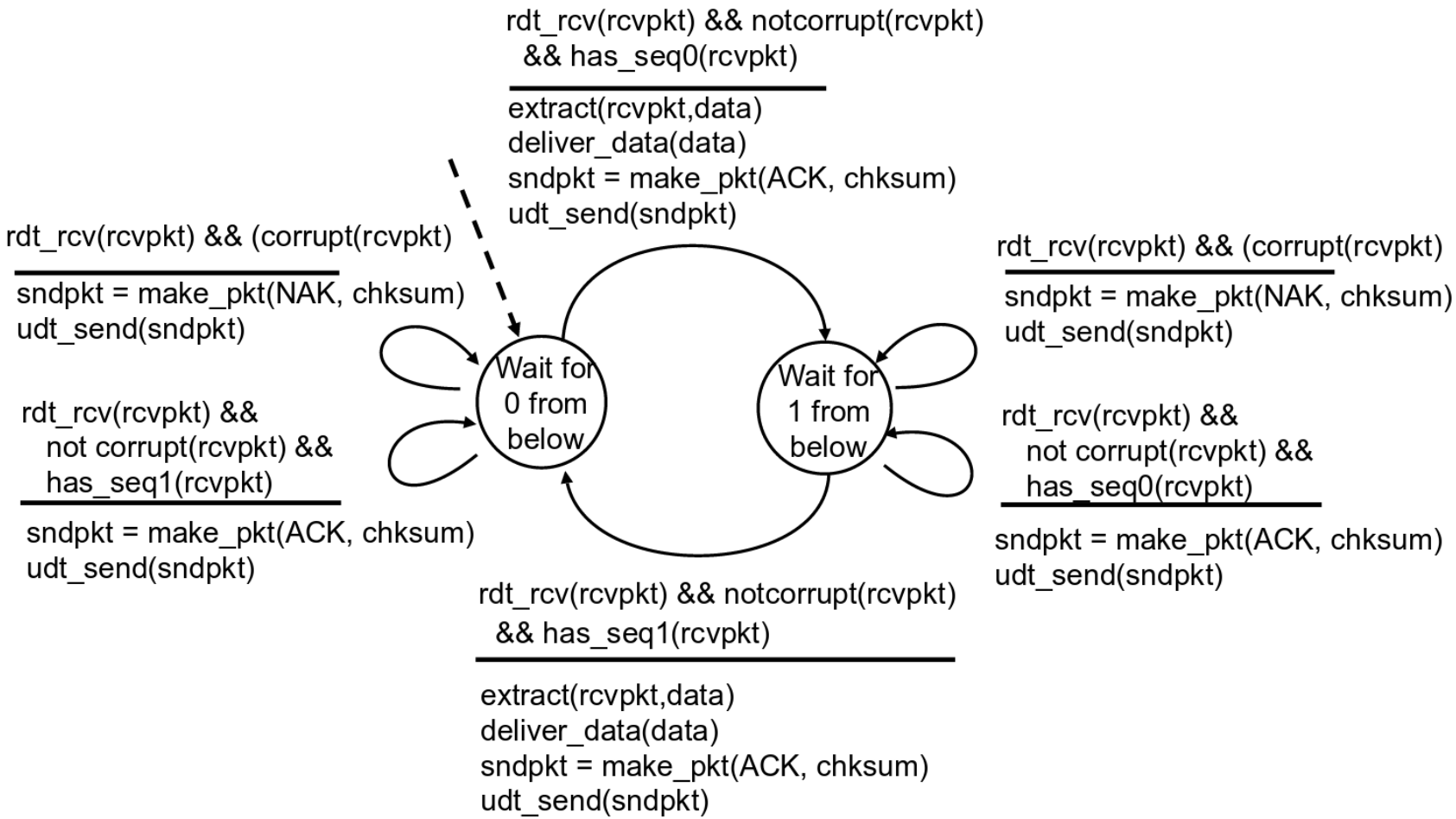
Дедупликация сообщений

- 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 (NACK-free)

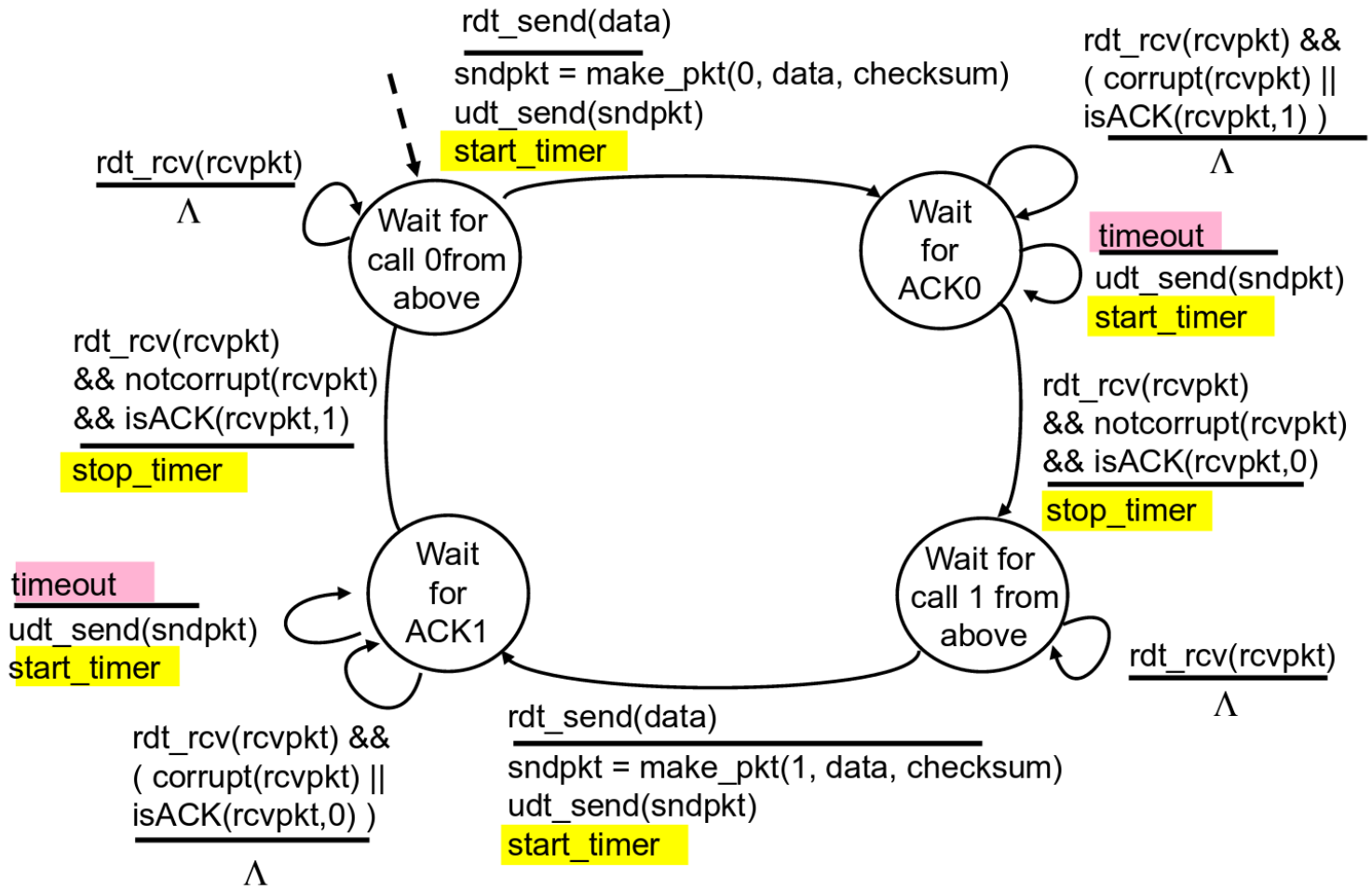
Канал с потерей сообщений

- Сообщения могут теряться/отбрасываться в процессе передачи
- Как обнаруживать и обрабатывать такие ошибки?

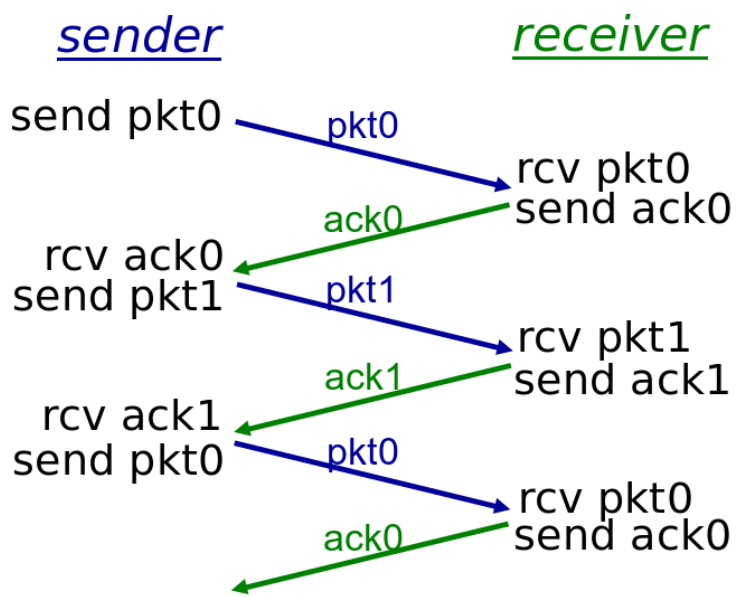
Базовые принципы

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

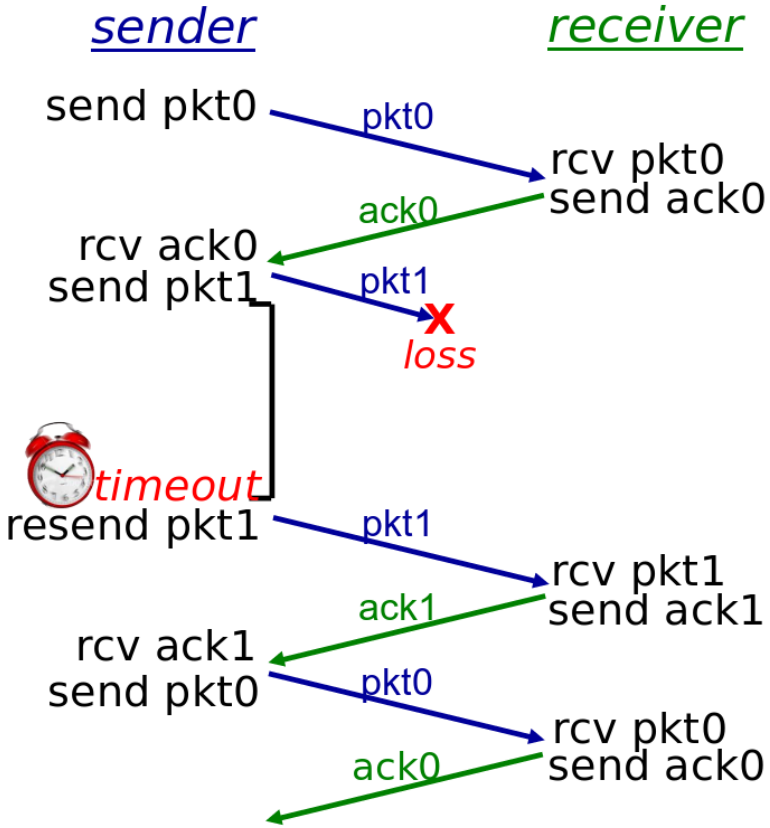
Протокол 3.0


rdt_send(data)
sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)
start_timer
rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
isACK(rcvpkt,1))
Λ
rdt_rcv(rcvpkt)
Λ
Wait for
call 0from
above
Wait
for
ACK0
timeout
udt_send(sndpkt)
start_timer
rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt,1)
stop_timer
rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt,0)
stop_timer
Wait
for
ACK1
Wait for
call 1 from
above
timeout
udt_send(sndpkt)
start_timer
rdt_rcv(rcvpkt)
Λ
rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
isACK(rcvpkt,0))
Λ
rdt_send(data)
sndpkt = make_pkt(1, data, checksum)
udt_send(sndpkt)
start_timer

Возможные сценарии

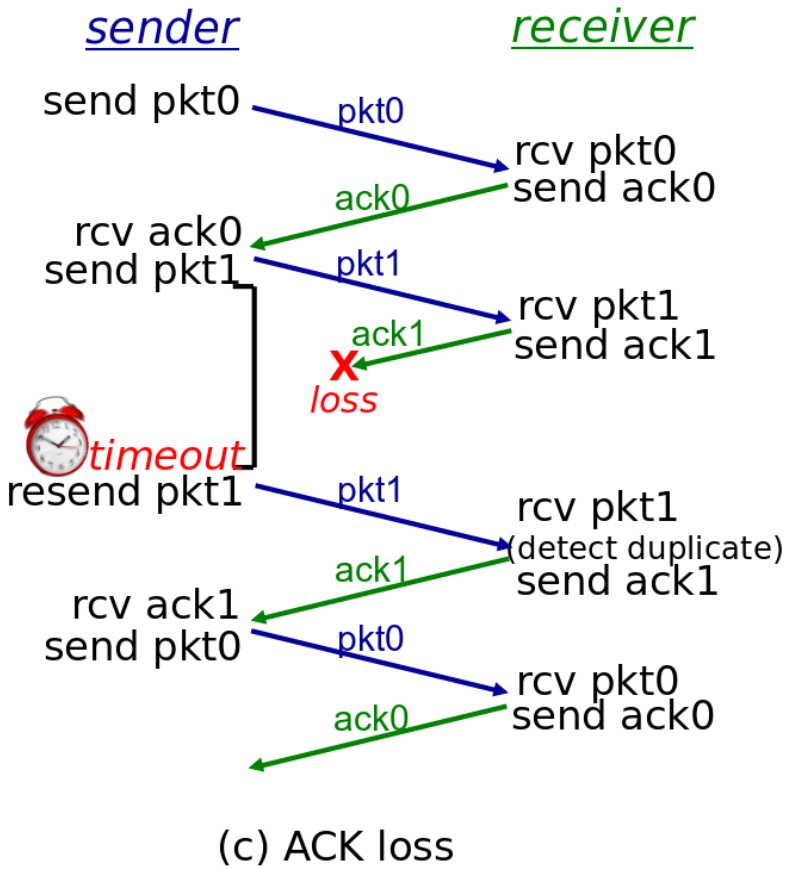


(a) no loss

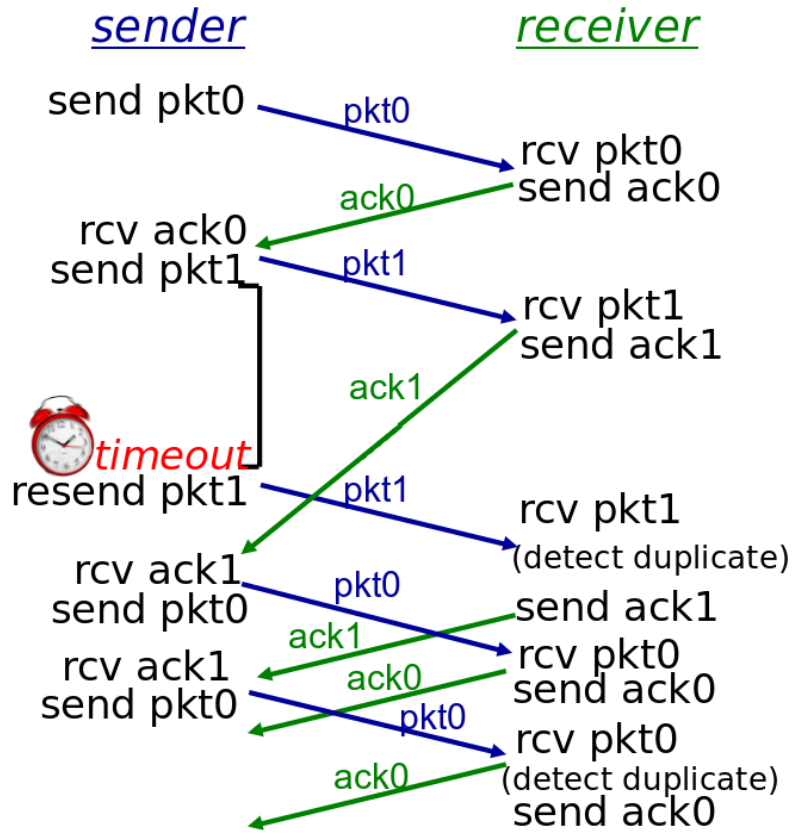


(b) packet loss

Возможные сценарии (2)

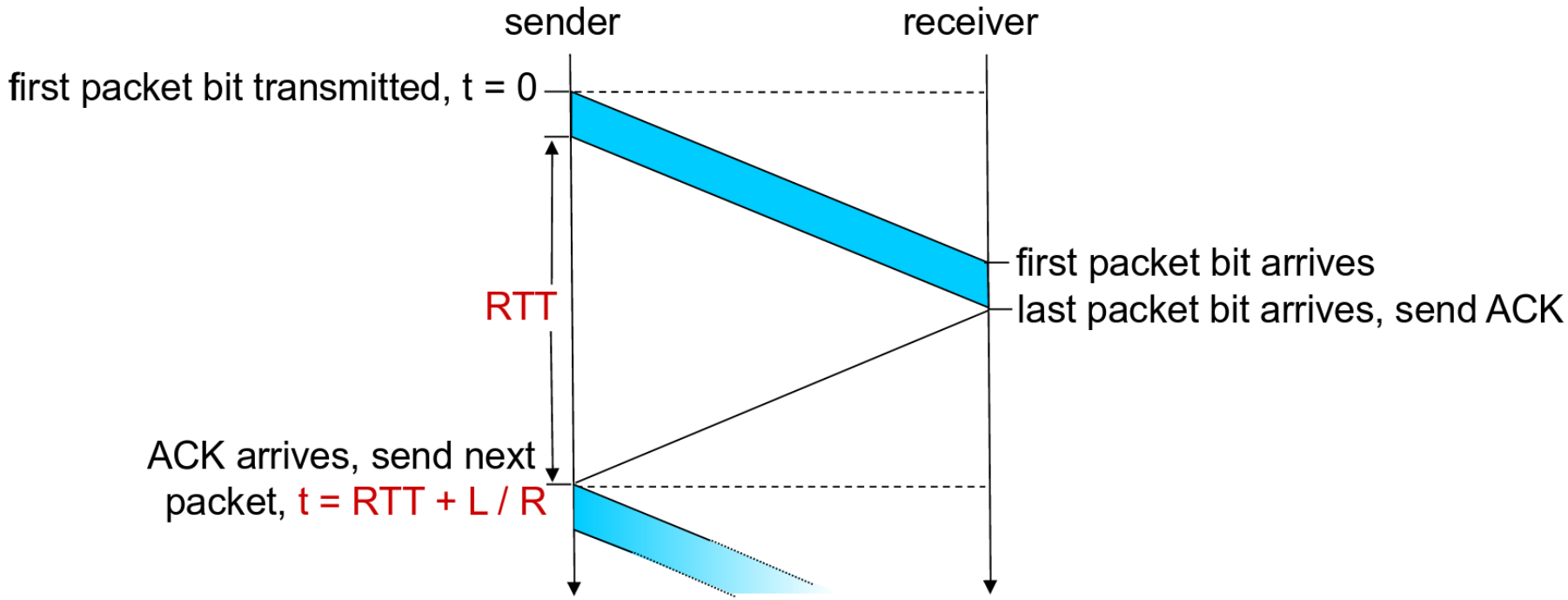


(c) ACK loss



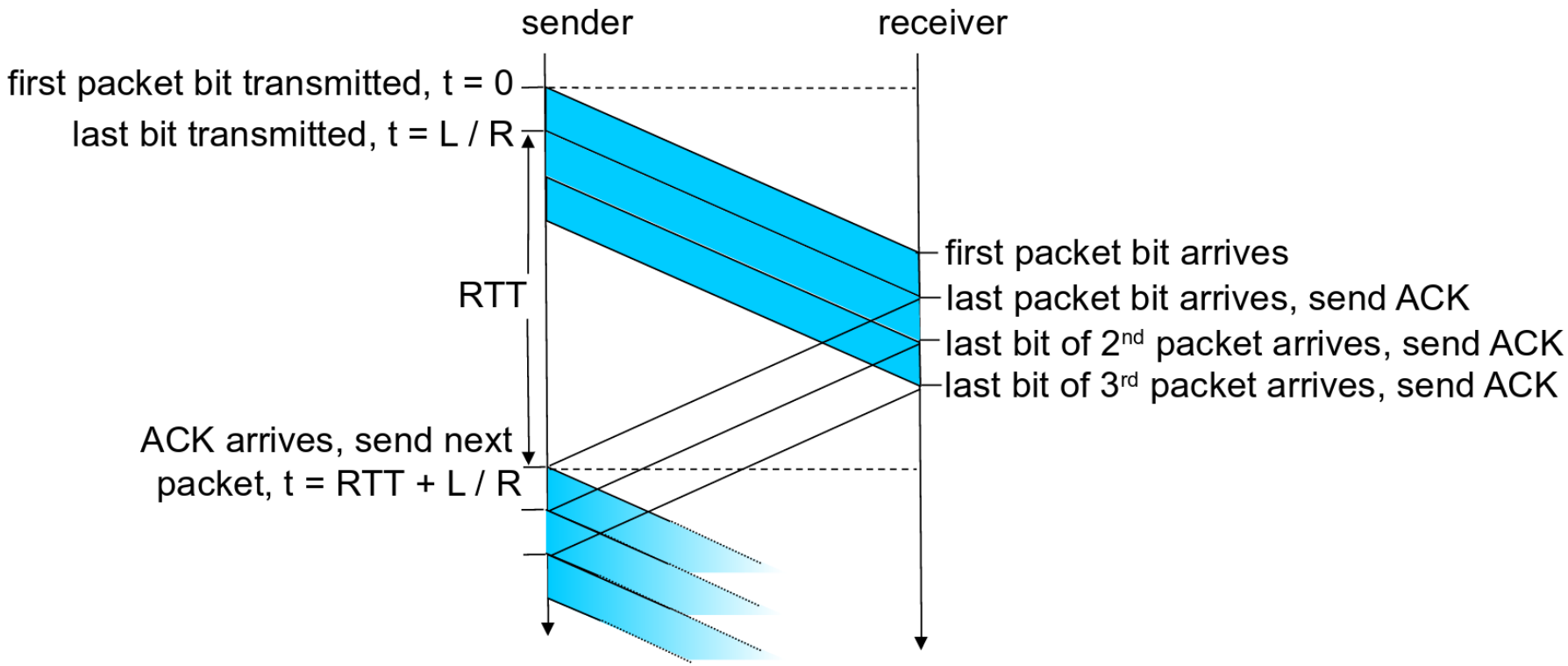
(d) premature timeout/ delayed ACK

Производительность



Принцип stop-and-wait приводит к низкой утилизации сети

Pipelining

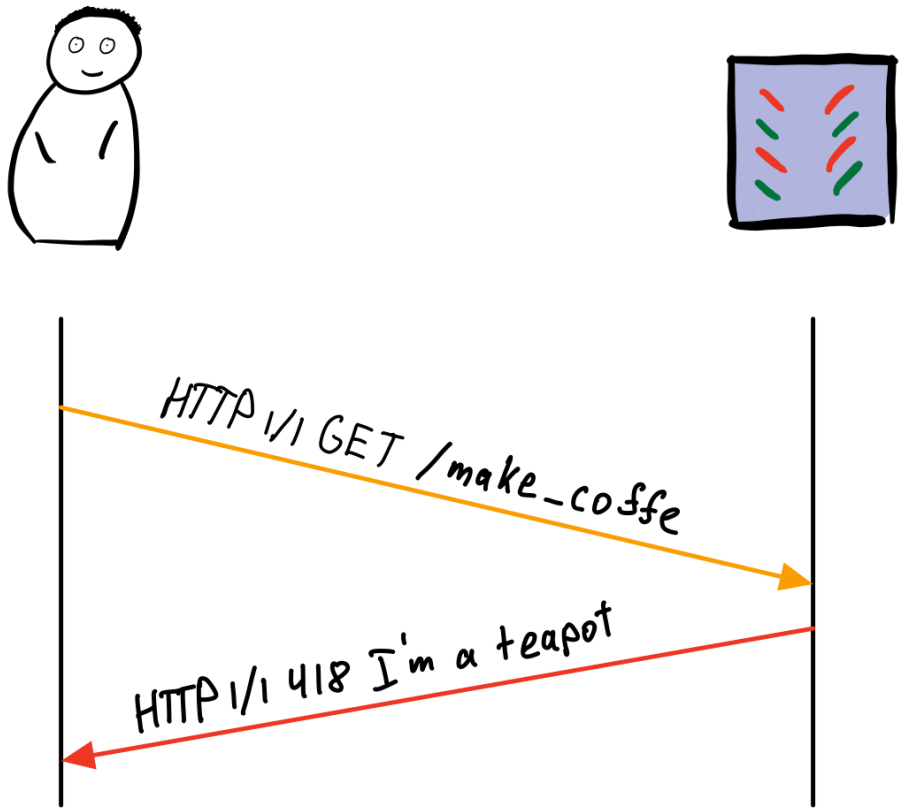


Отправка N пакетов за раз увеличивает утилизацию в N раз

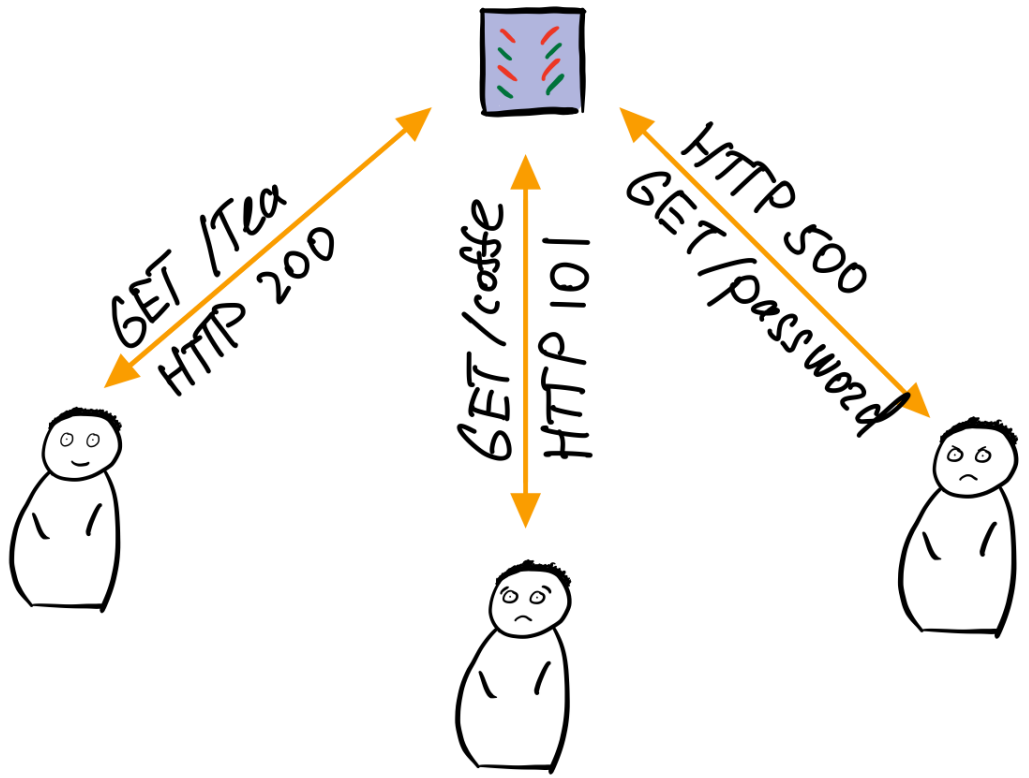
Сохранение порядка сообщений

- Как обеспечить получение сообщений в порядке их отправки?

Схема взаимодействия "запрос-ответ"



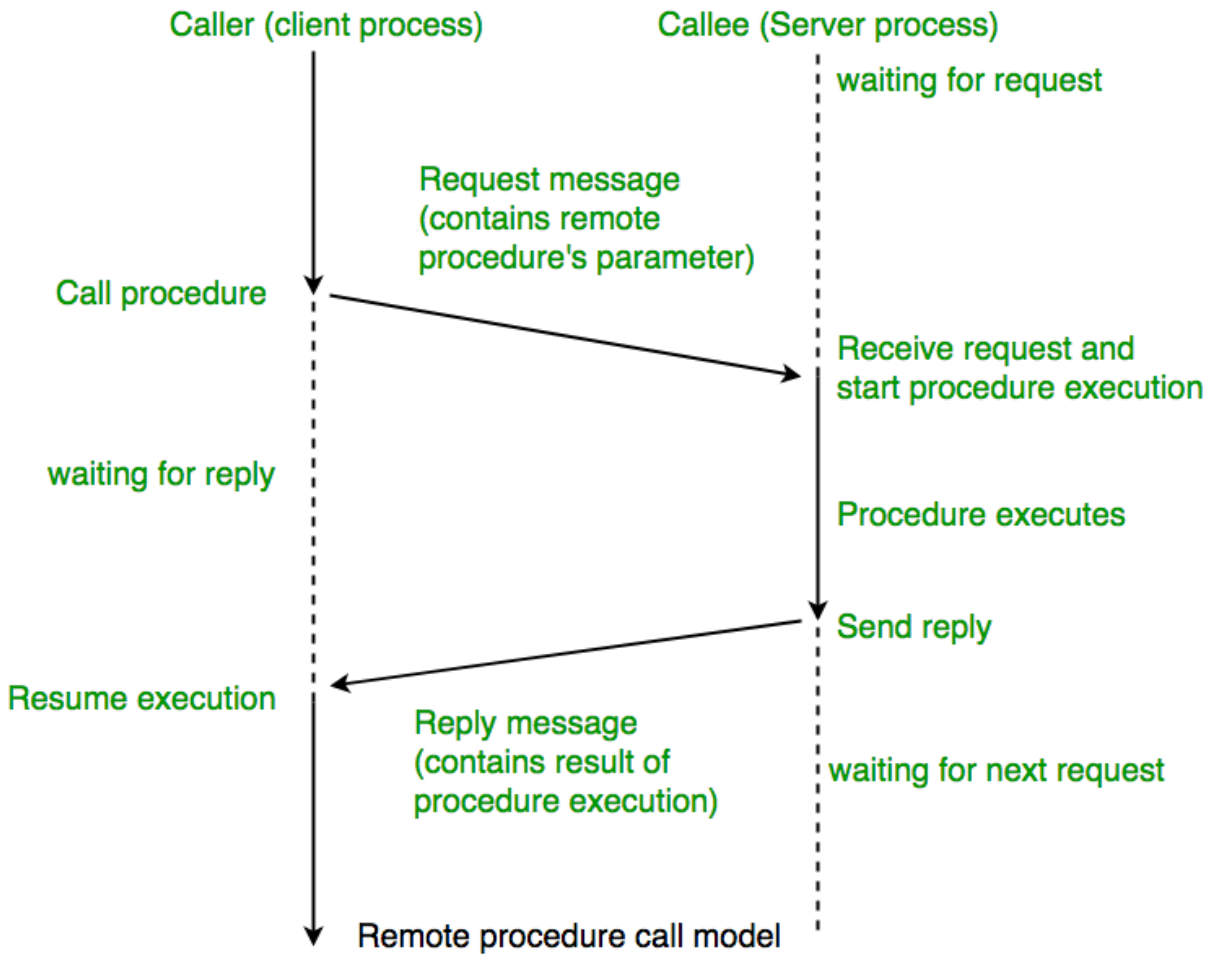
Клиент-сервер



Интерфейс

- Описание функциональности компонента и способов взаимодействия с ней
- Использование привычных абстракций программирования
  - Функции, процедуры, методы...
  - Параметры, типы данных, структуры...
  - Модули, пакеты, объекты...
- Подразумевает использование некоторого протокола для реализации взаимодействия с компонентом

Remote Procedure Call (RPC)



История RPC

- 1969: ARPANET
- 1970-е:
  - Ранние протоколы, ориентированные взаимодействие пользователя с сервером (Email, FTP, Telnet)
  - Интерес к протоколам взаимодействия между приложениями
  - 1974: RCF 674 "Procedure Call Protocol"
  - 1975: RFC 684 "A Commentary on Procedure Calling as a Network Protocol"
  - 1976: RFC 707 "A High-Level Framework for Network-Based Resource Sharing"

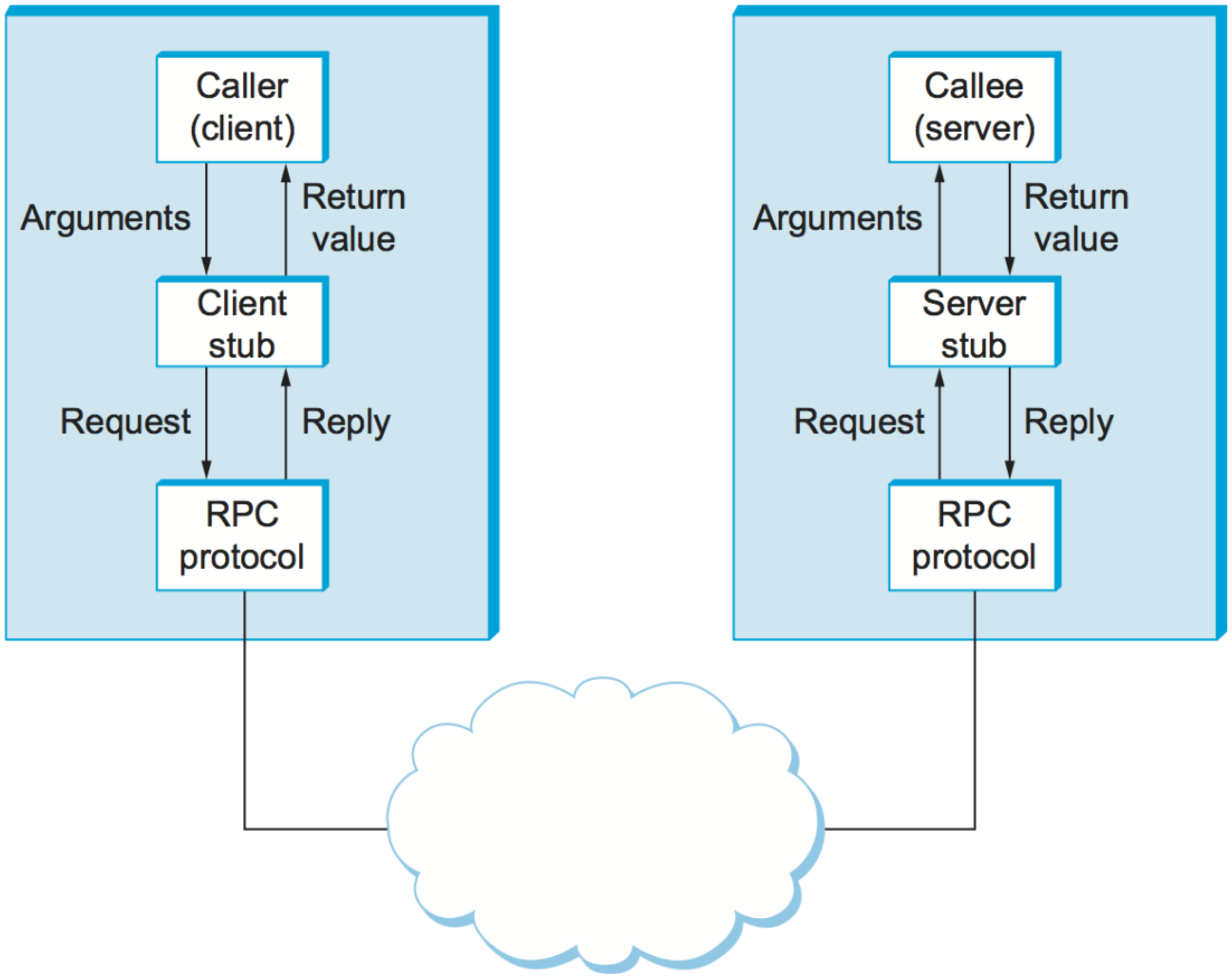
История RPC (2)

- 1980-е:
  - 1984: Birrell A. D., Nelson B. J. Implementing Remote Procedure Calls.
  - 1988: Tanenbaum A. S., Van Renesse R. A Critique of the Remote Procedure Call Paradigm.
  - Закрытые проприетарные реализации RPC (Apollo, Sun, IBM, HP...)
- 1990-е:
  - Распределенные объекты (CORBA, Java RMI, DCOM)

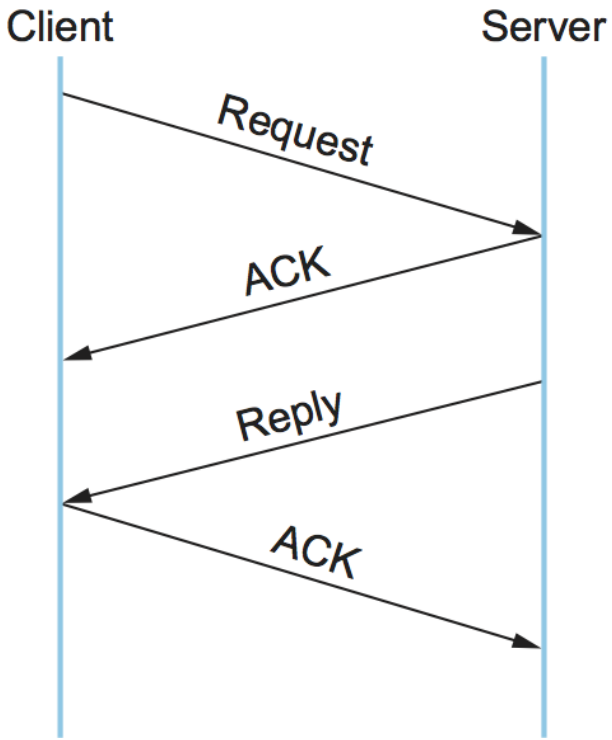
История RPC (3)

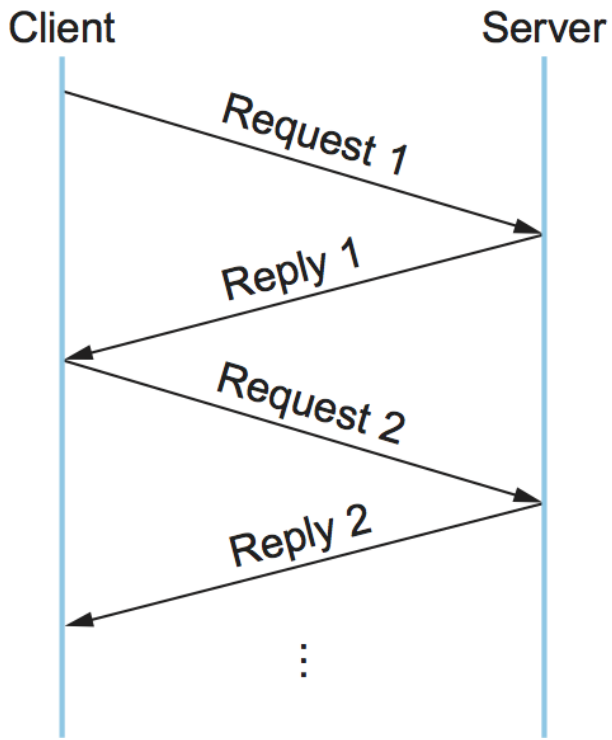
- 2000-е:
  - RPC поверх HTTP
  - Веб-сервисы (SOAP, REST)
- 2010-е
  - Современные реализации RPC (Thrift, Avro, Finagle, gRPC...)

Реализация RPC



Обеспечение надежности





Обеспечение надежности

- Подтверждения
- Идентификаторы запросов
- Хранение ответов на стороне сервера

Семантика вызова

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

Идемпотентные операции

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

RPC: детали реализации

- Описание интерфейса (IDL), генерация стабов
- Передача параметров вызова и результатов по сети (маршалинг)
- Выбор транспортного протокола
- One-way RPC
- Асинхронные (неблокирующие) вызовы
- Обработка ошибок, таймауты
- Регистрация сервера и связывание клиента с сервером
- Приоритезация запросов и очередь на стороне сервера

Sun RPC (aka ONC RPC)

- Одна из первых массовых реализаций RPC
- RFC: 1057 (1988), 1831 (1995), 5531 (2009)
- Доступна на многих ОС
- Поддерживает UDP и TCP
- IDL: нумерация программ/версий/процедур, один аргумент и возвращаемое значение
- Маршалинг данных с помощью External Data Representation (XDR)
- Связывание с помощью port mapper (порт 111)
- Клиент не отправляет серверу подтверждение о получении ответа
- Семантика at-least-once

Nework File System

- Один из первых массовых примеров использования RPC
- RFC 1094 (1989)
- Протокол не хранит состояние на стороне сервера (stateless)
- Почти все операции сделаны идемпотентными
  - как можно сделать идемпотентной операцию удаления файла?
  - что если сервер перезагрузится?
- read/write() оперируют блоками размера 8 КБ
- Soft vs hard mounting

Remote Method Invocation (RMI)

- Развитие идей RPC для поддержки разработки распределенных систем на основе объектно-ориентированного программирования
- Вызов метода удаленного объекта (remote object)
- Понятие ссылки на удаленный объект (remote object reference)
- Передача ссылок в аргументах и результате вызова
- Поддержка принципов ООП (наследование, полиморфизм...)
- Поддержка обратных вызовов (callbacks)
- Примеры: CORBA, Java RMI, ZeroC Ice

Современные реализации RPC

- Thrift
- Avro
- Finagle
- gRPC
  - масштабируемые сервисы
  - использование TCP, TLS, HTTP/2
  - поддержка streaming

Промежуточное ПО (middleware)

- Слой ПО между программными компонентами и ОС узлов распределенной системы, упрощающий реализацию приложений
  - Поддержка взаимодействия между компонентами по сети
  - Скрытие гетерогенности (архитектур, ОС, языков программирования)
  - Реализация типовой функциональности (например, именование и поиск)

Отличия локальных вызовов от удаленных

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

Недостатки удаленных вызовов?

- Может вводить в заблуждение программиста
- Ограниченная схема взаимодействия (point-to-point request-reply)
- Специфичные для каждого приложения методы и типы данных
- Скрывает workflow и переходы между состояниями
- Требует использования одинакового/тяжелого стека ПО на обеих сторонах
- Проблемы масштабирования

Литература

- van Steen M., Tanenbaum A.S. Distributed Systems: Principles and Paradigms (разделы 4.1-4.3)
- Kurose J., Ross K. Computer Networking: A Top-Down Approach (раздел 3.4)
- Peterson L., Davie B. Computer Networks: A Systems Approach (раздел 5.3)

Дополнительно

- Birrell A.D., Nelson B.J. Implementing Remote Procedure Calls (1984)
- Tanenbaum A.S., Van Renesse R. A Critique of the Remote Procedure Call Paradigm (1988)
- Jim Waldo et al. A Note on Distributed Computing (1994)
- Henning M. Another Note on Distributed Computing (2008)
- Vinoski S. Mythbusting Remote Procedure Calls (2012)
- Meiklejohn C., McCaffrey C. A Brief History of Distributed Programming: RPC (2016)
- Kalia A. Datacenter RPCs can be General and Fast (2019)