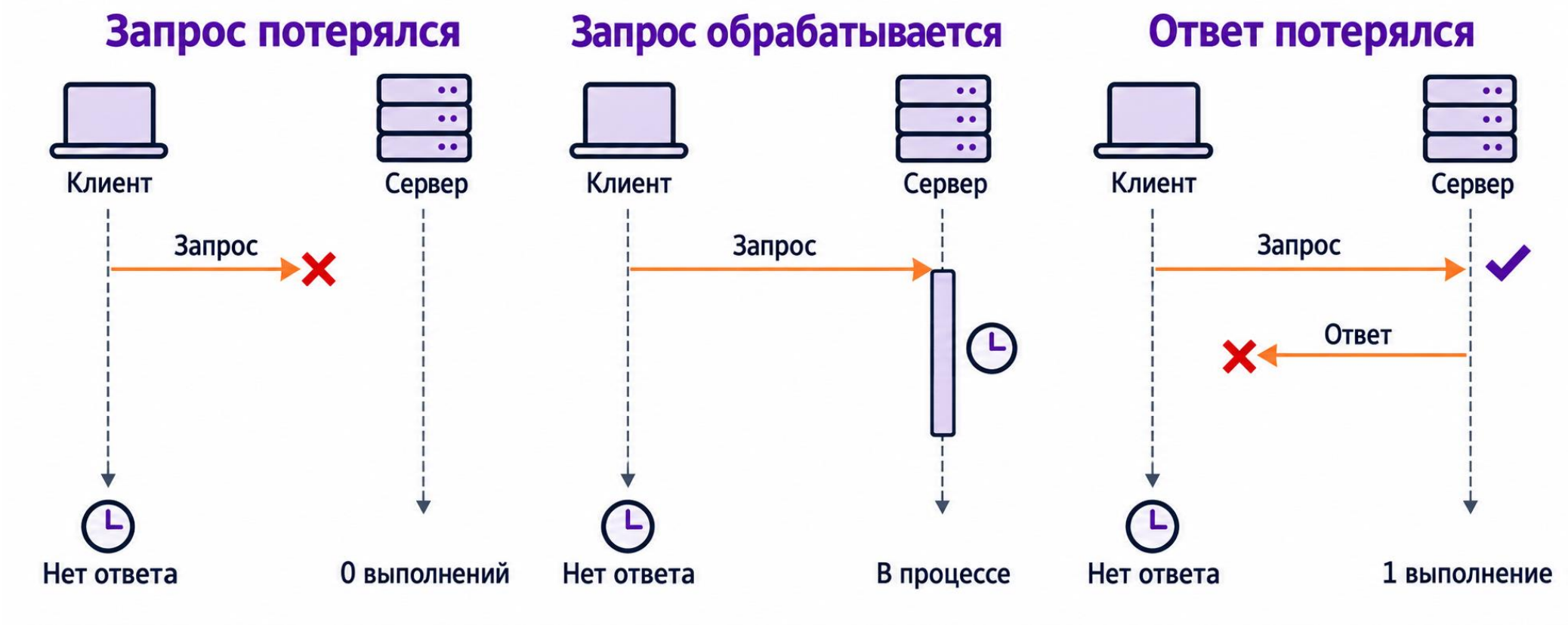
2. Взаимодействие между процессами в PC

Сухорослов Олег Викторович

14.09.2026

Клиент не получил ответ

Что произошло? Безопасно ли повторить запрос?

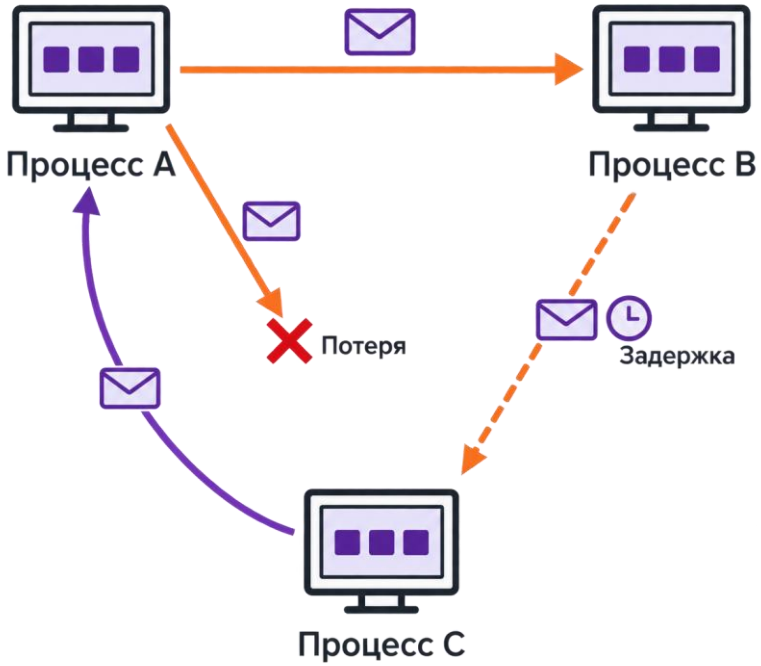


План

- Модели взаимодействия процессов
- Семантика `send` и `receive`: блокировки и буферизация
- Надёжная доставка: ошибки, потери, повторы и порядок
- RPC: гарантии и обработка отказов

Обмен сообщениями (Message Passing)

- Процессы взаимодействуют, отправляя и получая сообщения
- Каналы связи могут терять, задерживать, повреждать и переупорядочивать сообщения
- Подтверждения, повторная отправка и другие механизмы позволяют обеспечить гарантии доставки

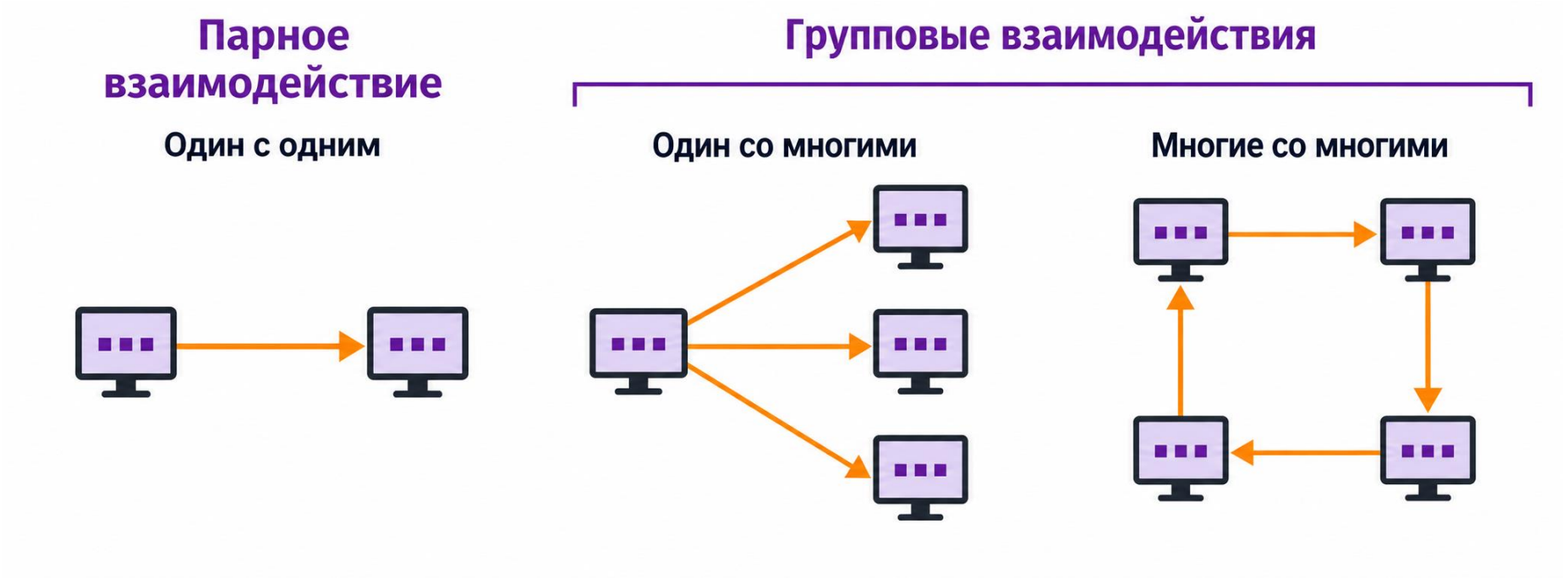


Варианты взаимодействий

- **Сколько процессов** участвует во взаимодействии?
- **В каких направлениях** передаются данные приложения?
- Предполагает ли сообщение **ответ приложения**?
- **Кто инициирует** передачу данных?
- Нужно ли знать **конкретного получателя**?
- Должны ли отправитель и получатель **быть активны одновременно**?

Число взаимодействующих процессов

- **Один с одним** – парное взаимодействие (*point-to-point communication*)
- **Один со многими** – например, рассылка обновления нескольким получателям
- **Многие со многими** – например, обмен сообщениями внутри группы реплик



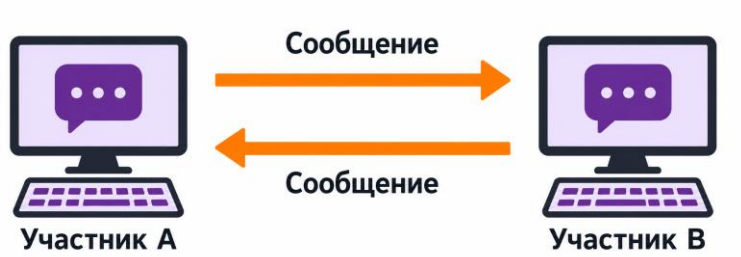
Направления передачи данных

В одну сторону (unidirectional)

- Один процесс отправляет данные, другой получает
- Пример: датчик передаёт измерения сборщику

В обе стороны (bidirectional)

- Каждый процесс может отправлять и получать данные
- Пример: участники чата обмениваются сообщениями



Требуется ли ответ от получателя?

Без ответа (one-way)

- Ответ приложения на сообщение не предусмотрен
- Пример: отправка события в систему сбора логов

Запрос–ответ (request–response)

- Получатель обрабатывает запрос и отправляет связанный с ним ответ
- Пример: запрос текущей температуры и ответ со значением

Кто инициирует передачу данных?

Push: по инициативе источника

- Источник отправляет данные **без отдельного запроса** на каждую передачу
- Пример: сервис присылает новые публикации подписчику

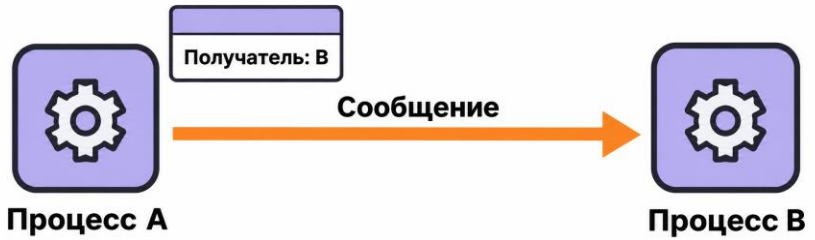
Pull: по запросу получателя

- Получатель **сам запрашивает данные** у источника
- Пример: приложение запрашивает новости при обновлении ленты

Связанность процессов

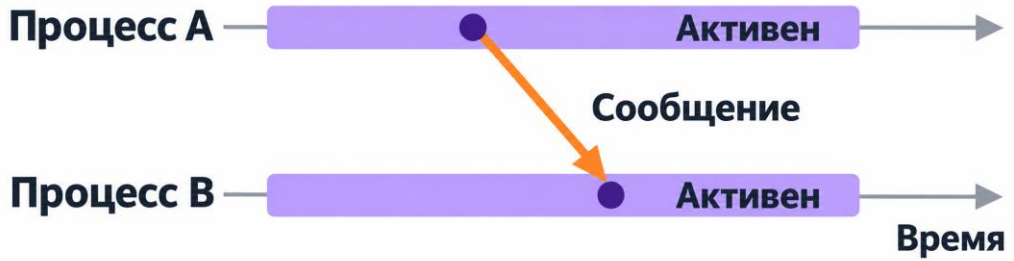
По пространству (space coupling)

- Отправитель должен **знать конкретного получателя**
- Например, его адрес или идентификатор



По времени (time coupling)

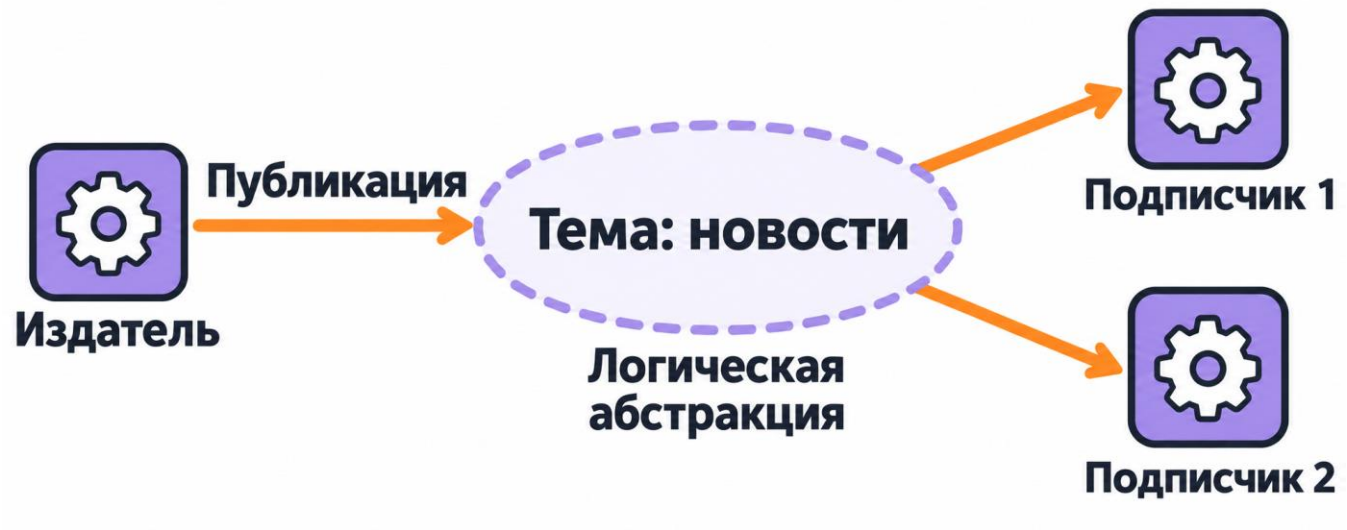
- Отправитель и получатель должны **участвовать в обмене одновременно**
- Если получатель недоступен, передать ему сообщение сейчас нельзя



Косвенное взаимодействие (indirect)

Обмен данными через **общую абстракцию**

- Примеры: очередь, тема (topic), общее хранилище
- Отправителю не обязательно **знать конкретного получателя**
- Если данные сохраняются, участники могут работать **в разное время**



Простая модель взаимодействия

- **Два процесса:** отправитель и получатель
- Передачу сообщения **инициирует отправитель**
- Отправитель **знает адрес получателя**
- Оба процесса **активны во время обмена**

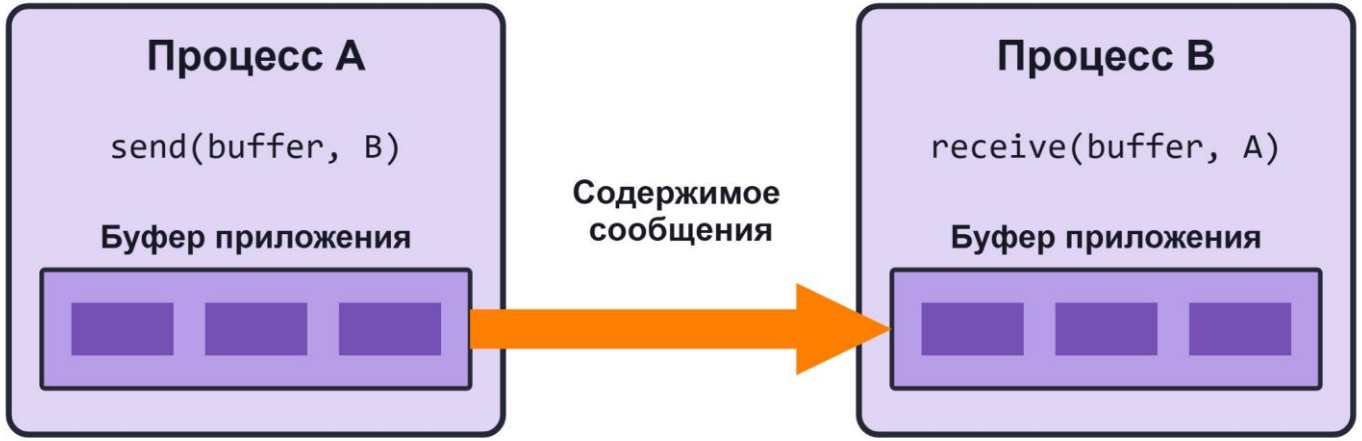
Отправка и получение сообщений

`send(buffer, dest)` – отправка сообщения

- `buffer` – буфер приложения с отправляемыми данными
- `dest` – адрес получателя

`receive(buffer, source)` – получение сообщения

- `buffer` – буфер приложения для принимаемых данных
- `source` – адрес отправителя



Что означает завершение send()?

Процесс A	Процесс B

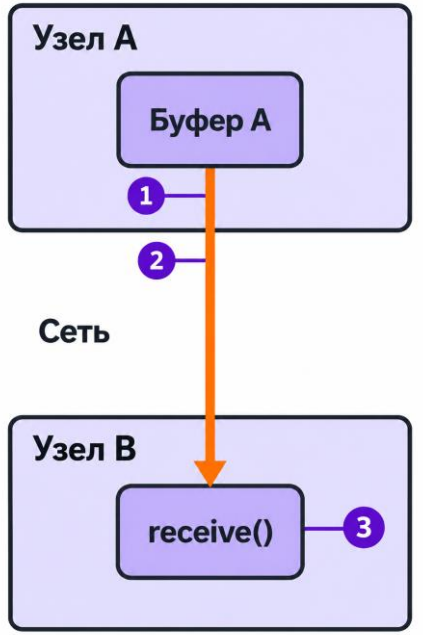
`data = 100`
`send(&data, B)`
`data = 0`
	`receive(&data, A)`
	`print(data) // ?`

Когда отправитель может безопасно изменить буфер?

Возможные моменты завершения:

- Операция больше не использует буфер приложения
- Сообщение покинуло узел отправителя
- `receive()` у получателя завершился

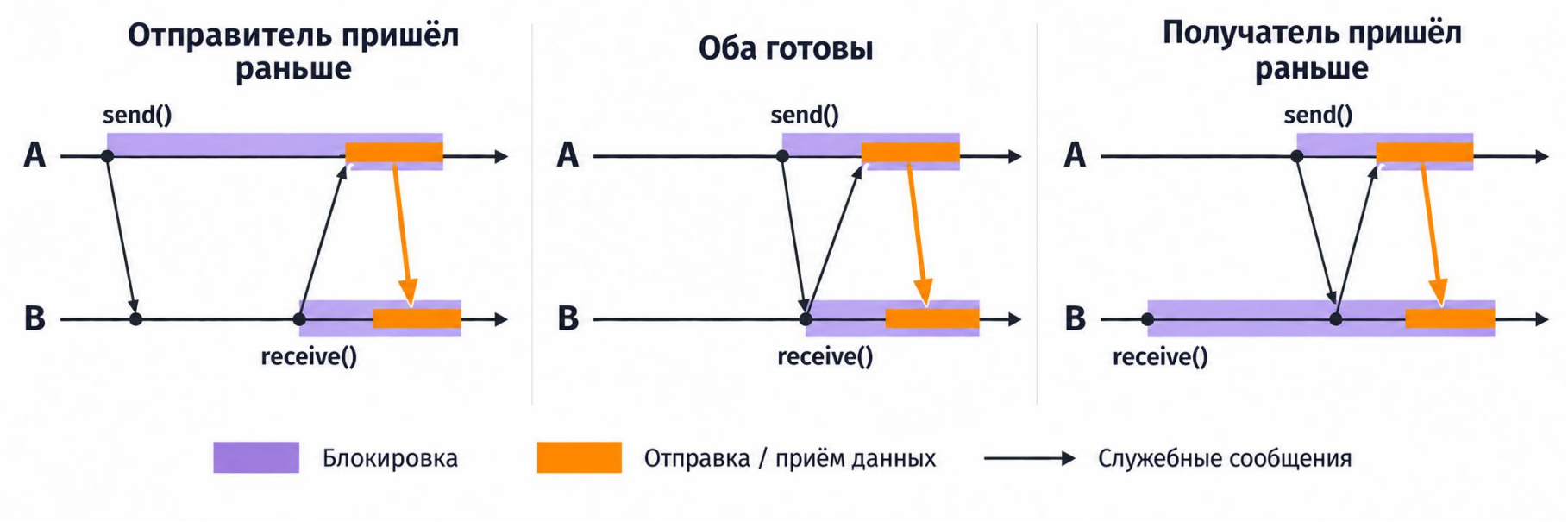
Момент завершения определяется семантикой операции.



Блокирующая передача без буферизации

Передача требует участия **обоих процессов**.

- `send()` ожидает соответствующий **`receive()`**
- `receive()` ожидает **сообщение**
- Пока операция не завершена, **вызывающий поток заблокирован**



Завершится ли такой обмен?

Блокирующие операции, **без промежуточной буферизации**

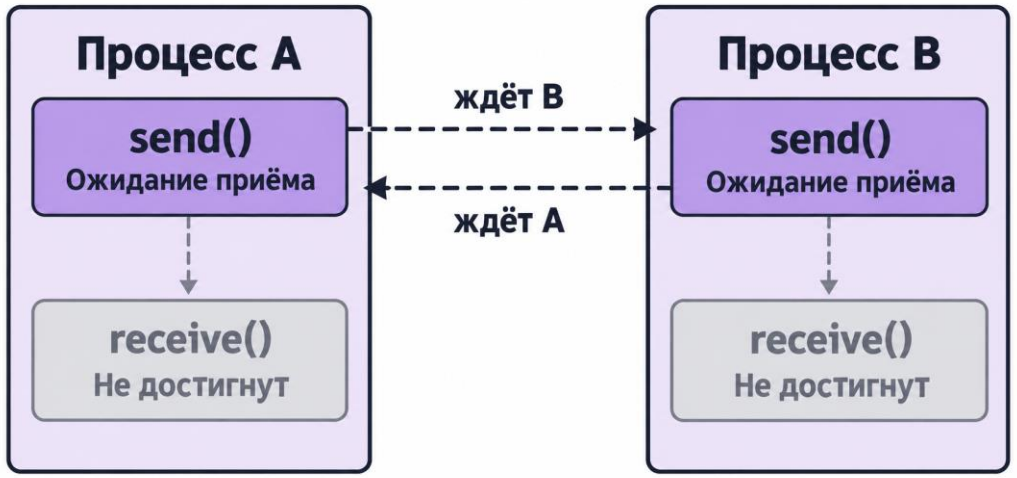
Процесс A	Процесс B
`send(&a, B)` `receive(&b, B)`	`send(&b, A)` `receive(&a, A)`

Завершится ли такой обмен?

Блокирующие операции, **без промежуточной буферизации**

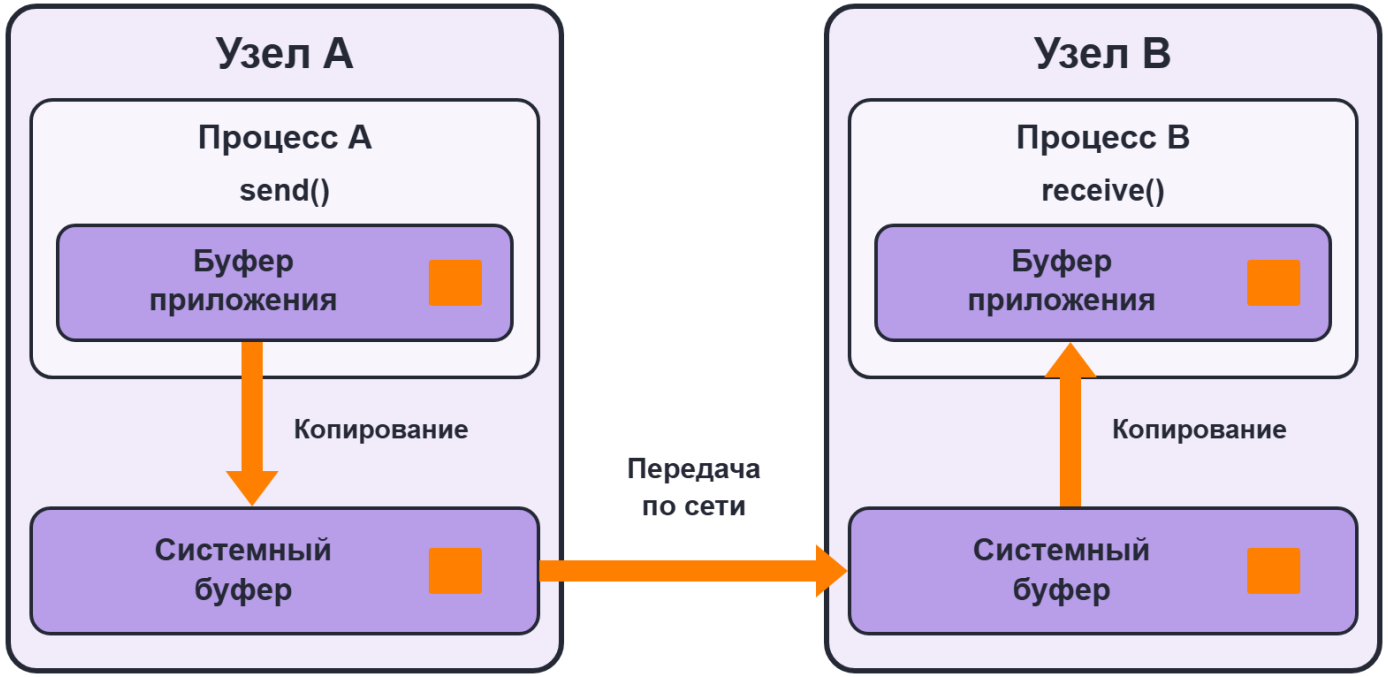
Процесс A	Процесс B
`send(&a, B)` `receive(&b, B)`	`send(&b, A)` `receive(&a, A)`

Взаимная блокировка (deadlock): оба ждут на send() и не доходят до receive()



Использование системных буферов

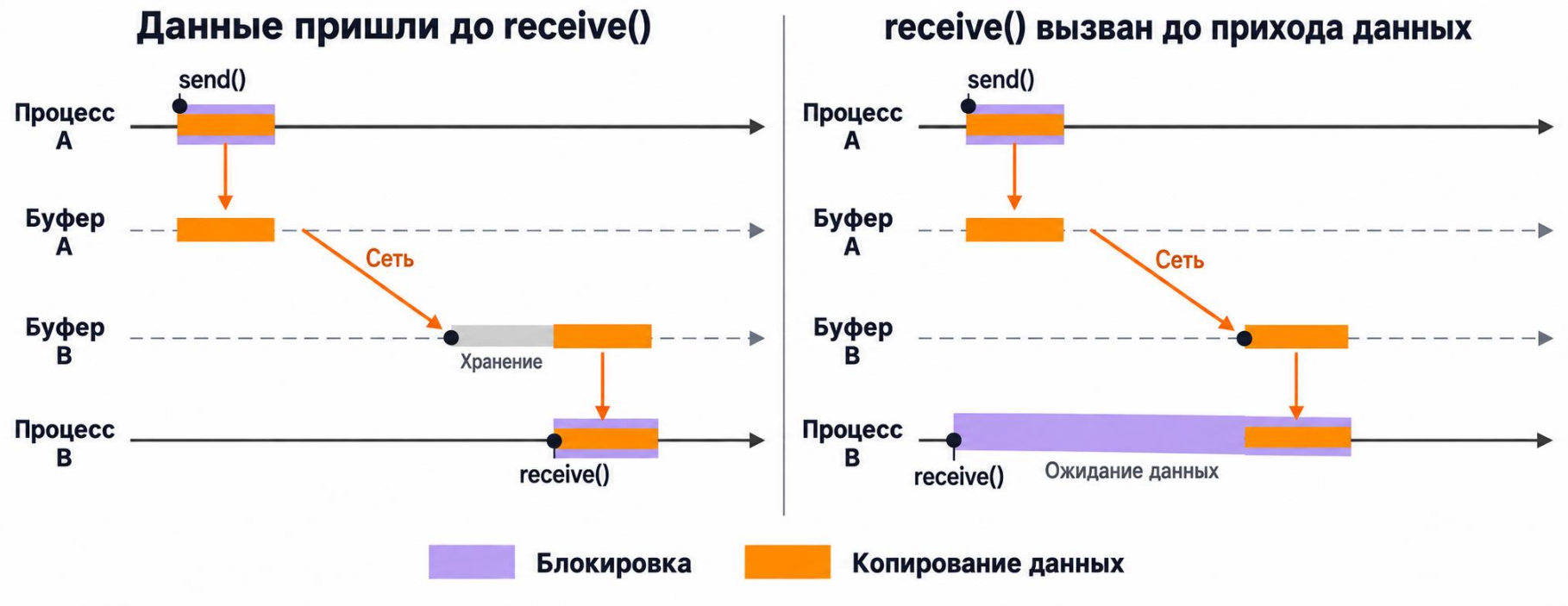
- Данные временно хранятся **в буферах ОС или коммуникационной библиотеки**
- `send()` **может завершиться до вызова** `receive()`
- Завершение отправки **не означает получение данных приложением**



Блокирующая передача с буферизацией

Рассматриваем случай, когда **в буферах достаточно места**.

- `send()` завершается после **копирования в системный буфер отправителя**
- Передача по сети продолжается **независимо от вызывающего потока**
- `receive()` завершается, когда данные **скопированы в буфер приложения**

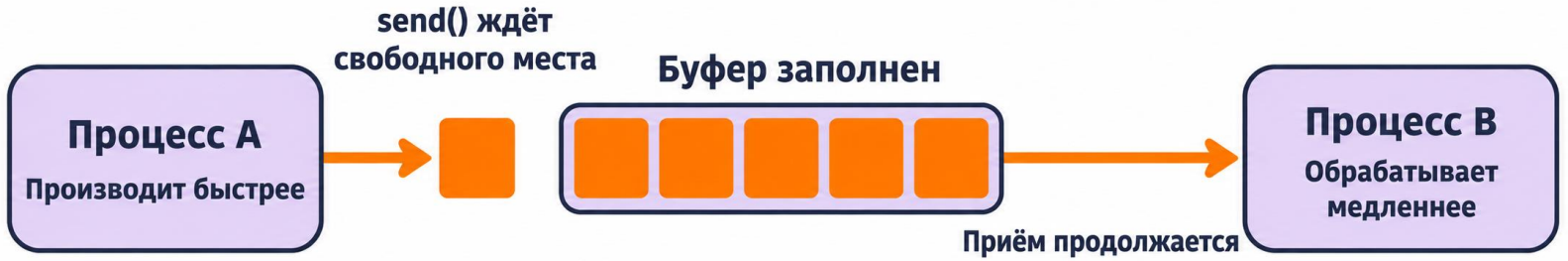


Если отправитель быстрее получателя

Процесс A	Процесс B

loop {
  produce_data(&data)
  send(&data, B)
}
	loop {
	  receive(&data, A)
	  consume_data(&data)
	}

- Если данные поступают быстрее, чем обрабатываются, **буферы заполняются**
- При нехватке места **send() может блокироваться**
- Буферизация **сглаживает временные всплески**, но не устраняет постоянную перегрузку



Взаимные блокировки при буферизации

1. Оба процесса сначала принимают

Процесс A	Процесс B
`receive(&a, B)` `send(&b, B)`	`receive(&a, A)` `send(&b, A)`

Взаимная блокировка независимо от размера буферов

2. Оба процесса сначала отправляют

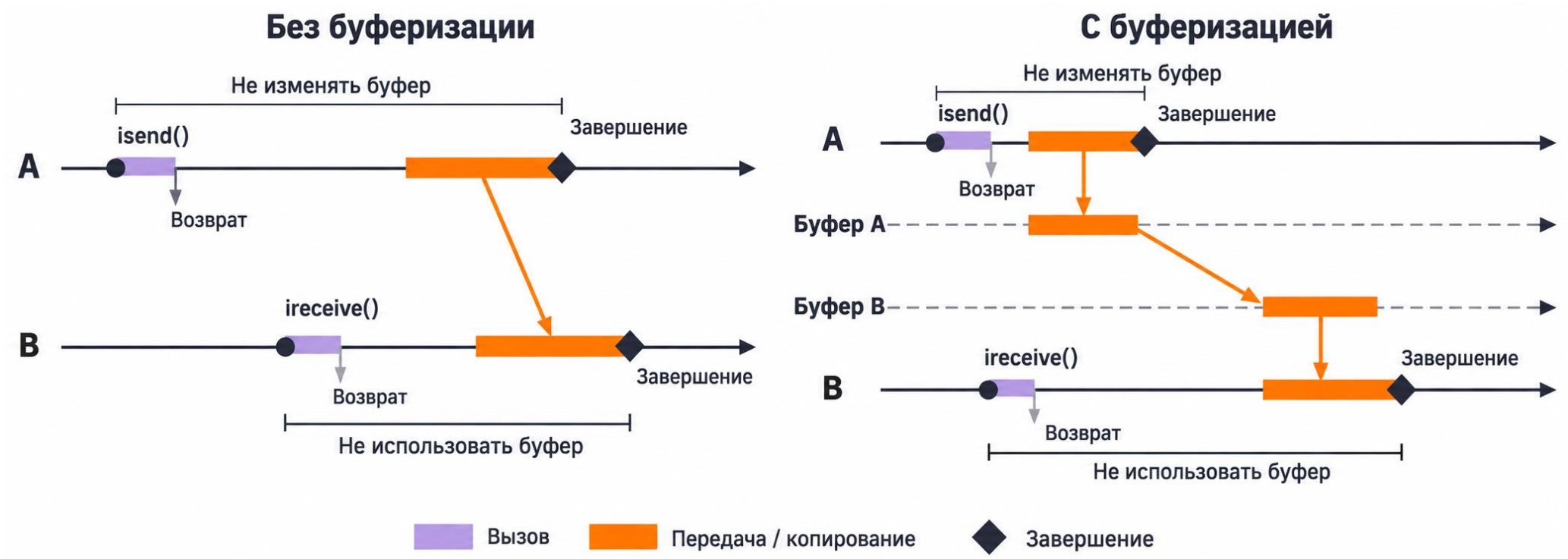
Процесс A	Процесс B
`send(&a, B)` `receive(&b, B)`	`send(&a, A)` `receive(&b, A)`

Взаимная блокировка возможна при нехватке буферов

Нельзя всегда рассчитывать на наличие свободных буферов.

Неблокирующие отправка и получение

- Вызов возвращает управление **до завершения операции**
- До завершения действуют **ограничения на буфер**
- Состояние операции можно **проверить** или **дождаться её завершения**



Блокировка вызова и синхронность передачи

Блокирующий / неблокирующий вызов
Ждёт ли вызывающий поток завершения операции внутри этого вызова?

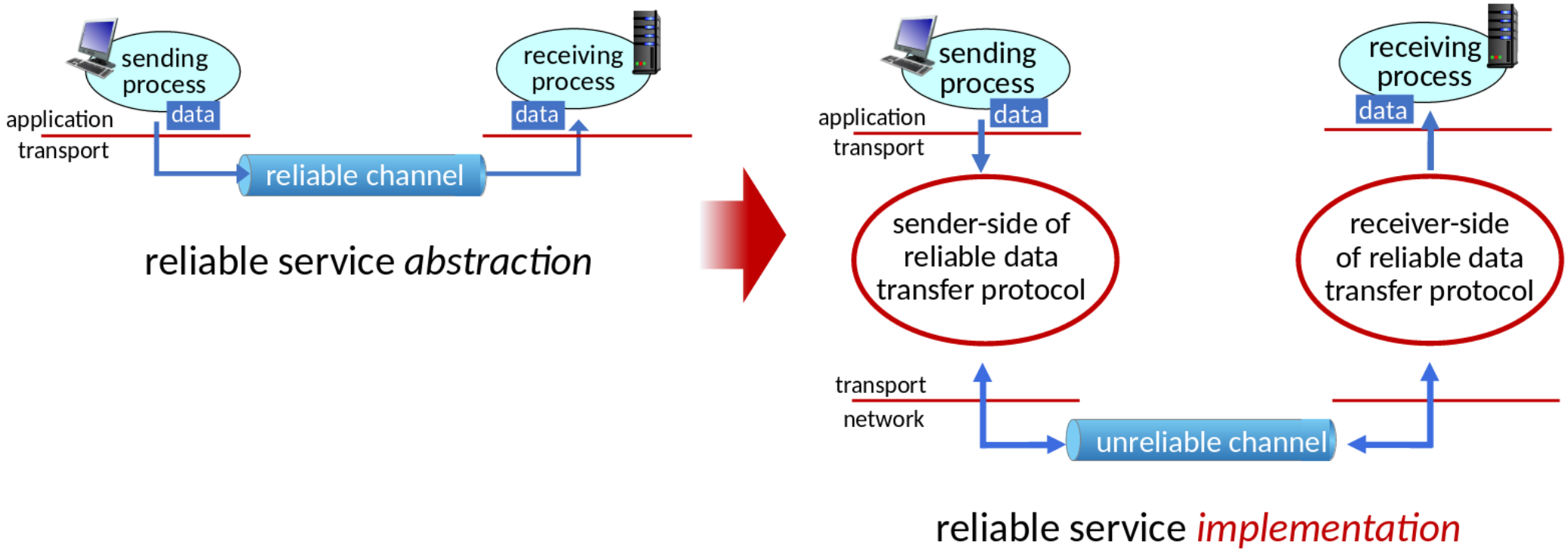
Синхронная / асинхронная передача
Требует ли завершение отправки участия получателя в приёме?

Неблокирующий вызов не обязательно означает асинхронную передачу.

	Блокирующий вызов	**Неблокирующий вызов**
Синхронная передача	Поток ждёт завершения отправки с участием получателя	Поток продолжает работу, но отправка ждёт участия получателя
Асинхронная передача	Поток ждёт передачи данных в промежуточное хранилище	Поток продолжает работу; отправка может завершиться до приёма

Надежная передача сообщений

Как обеспечить надёжную доставку поверх ненадёжного канала?

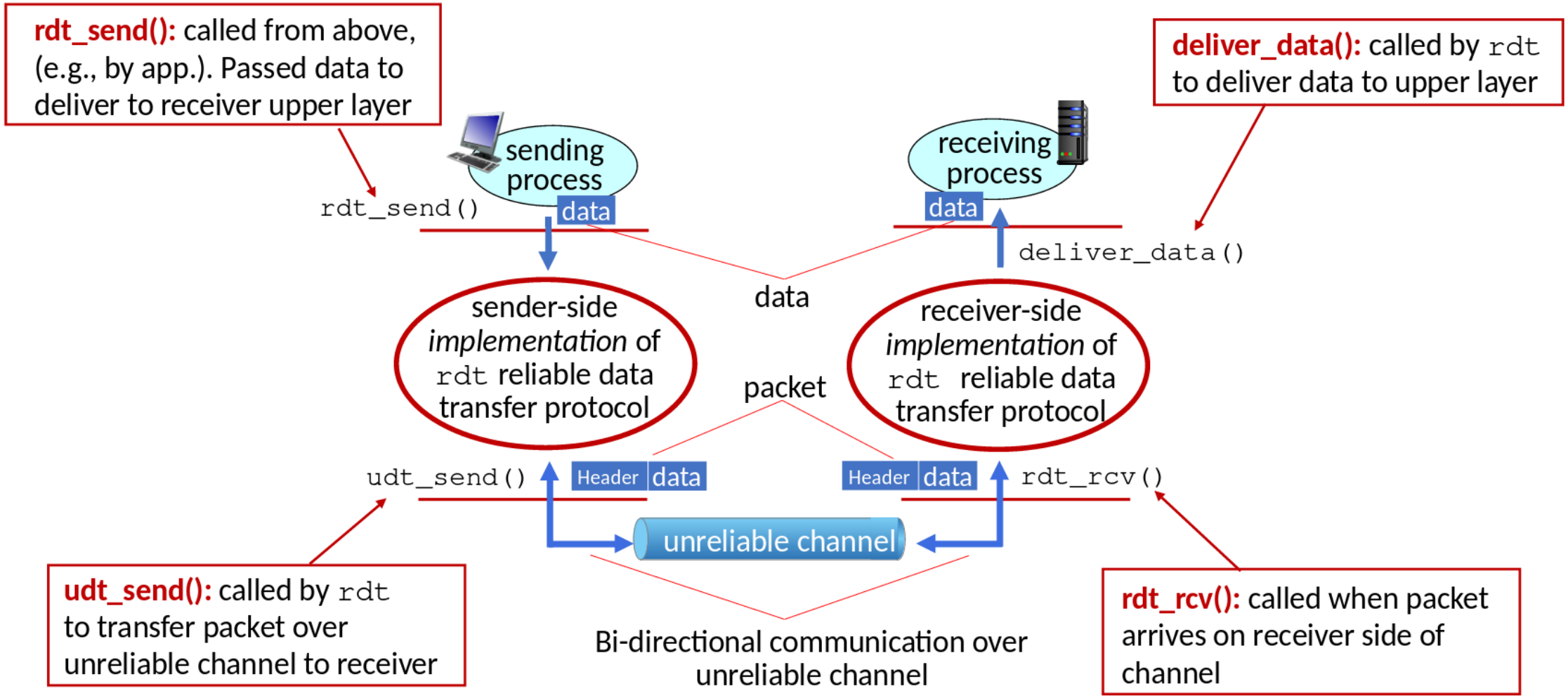


На основе материалов [Jim Kurose. Computer Networks: Principles of Reliable Data Transfer]

Модель и гарантии передачи

- Модель взаимодействия
  - Данные передаются от отправителя к получателю
  - Служебные сообщения могут идти в обе стороны
- Ненадёжный канал
  - Сообщения могут искажаться и теряться
  - Переупорядочивание пока исключаем
- Требуемые гарантии
  - Все отправленные сообщения доставляются без искажений и повторов, в порядке отправки

Интерфейсы передачи данных



Протокол передачи данных

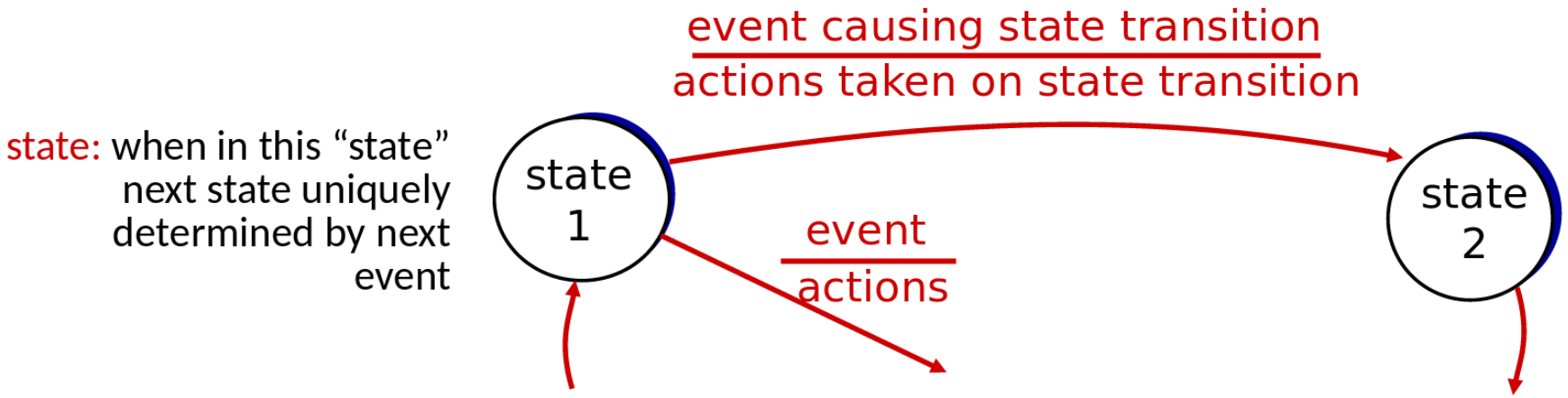
Набор **правил взаимодействия** между участниками передачи

- **Сообщения:** типы, структура и смысл
- **Отправка:** когда и какие сообщения передавать
- **Обработка событий:** как реагировать на полученные сообщения и истечение таймера

Описание протокола конечными автоматами

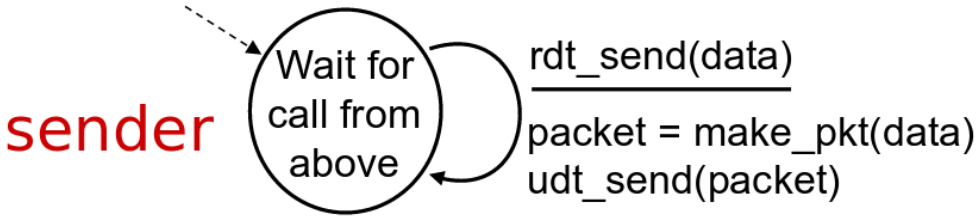
Поведение отправителя и получателя описываем отдельными **конечными автоматами** (finite-state machines, FSM)

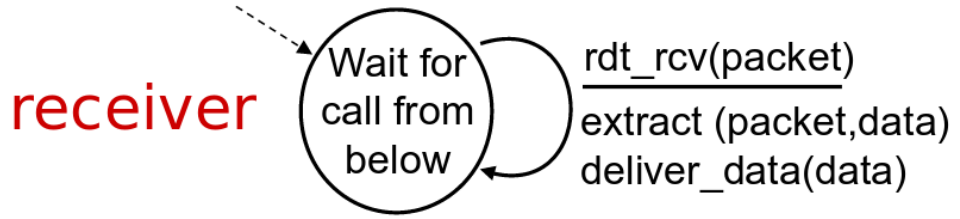
- **Состояния:** что участник помнит и чего ожидает
- **События и условия:** когда выполняется переход
- **Действия:** что выполняется при переходе



Протокол 1.0: надёжный канал

- Канал доставляет сообщения без искажений, потерь и повторов, в порядке отправки
- Отправитель формирует и отправляет пакет
- Получатель извлекает и передаёт данные приложению





Канал с искажениями данных

- При передаче биты пакета могут измениться
- Пакеты пока не теряются и не меняют порядок

Как обнаружить повреждение и доставить исходные данные?

RFC 3385 (2002) – Вероятность необнаруженных ошибок и сравнение алгоритмов контрольных сумм
When The CRC and TCP Checksum Disagree (2000) – Исследование ошибок в реальном сетевом трафике

Служебные сообщения ACK и NAK

ACK (*acknowledgement*): пакет получен без ошибок

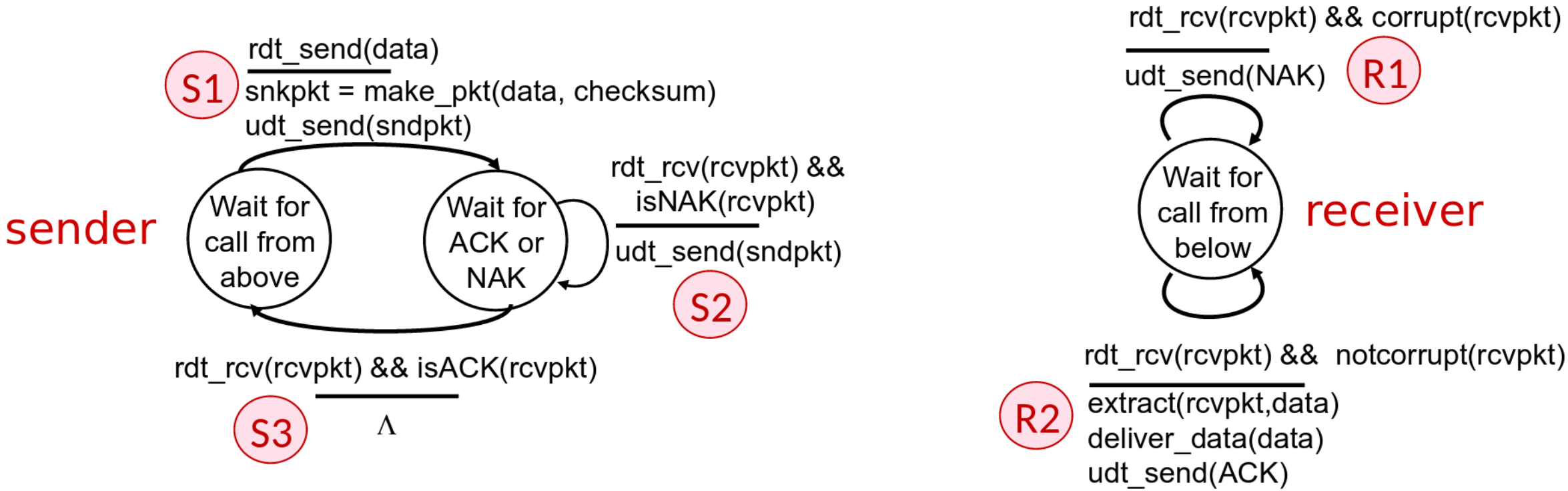
NAK (*negative acknowledgement*): обнаружено повреждение пакета

Отправитель ждёт ответа:

- ACK – можно отправить следующий пакет
- NAK – нужно повторить текущий пакет

Протокол 2.0

Пакеты данных могут повреждаться; ACK и NAK приходят без ошибок. Потерь нет.



Если повреждено подтверждение

Отправитель получил ответ, но не может определить: ACK это или NAK

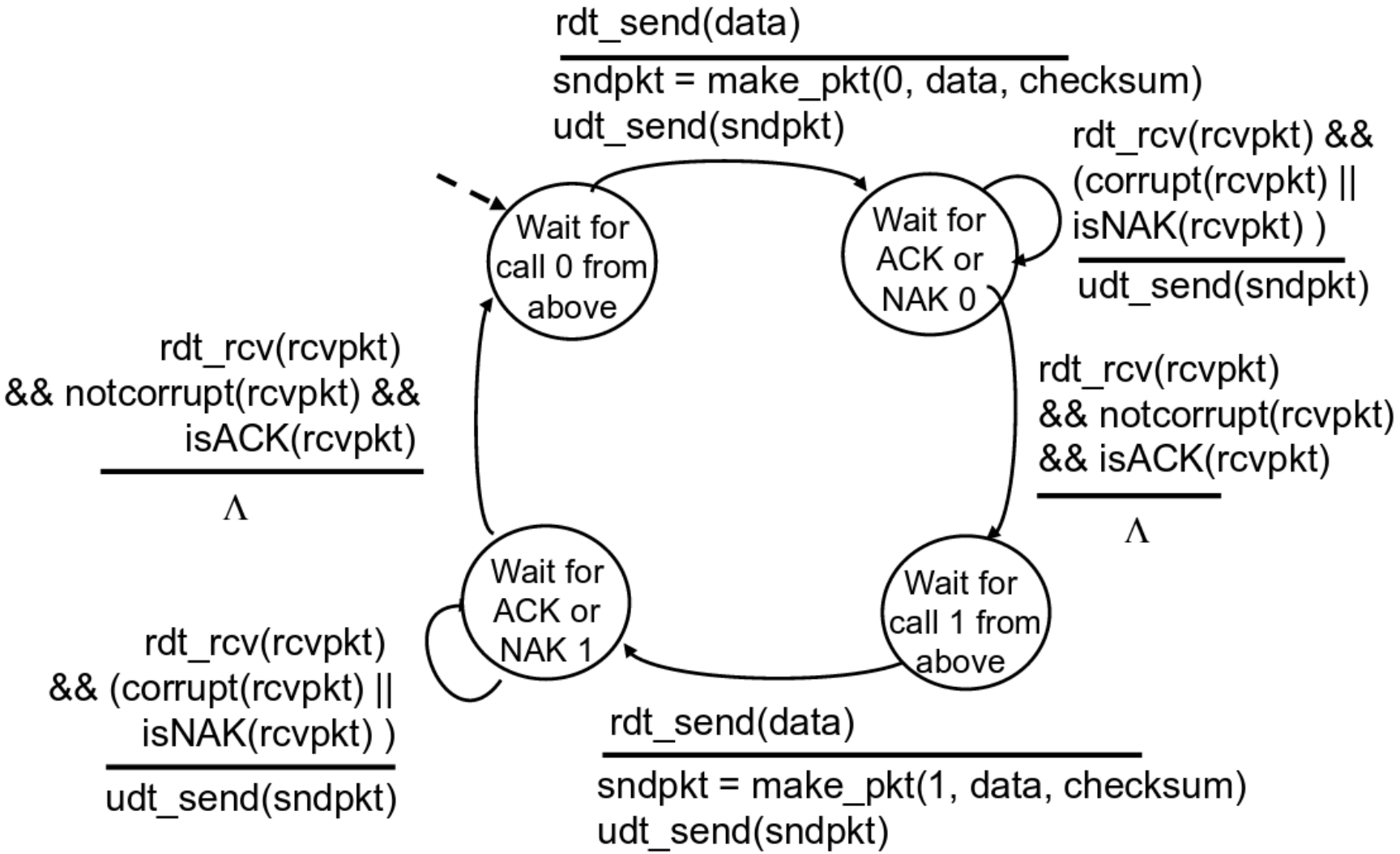
Можно ли безопасно отправить пакет повторно?

Дедупликация сообщений

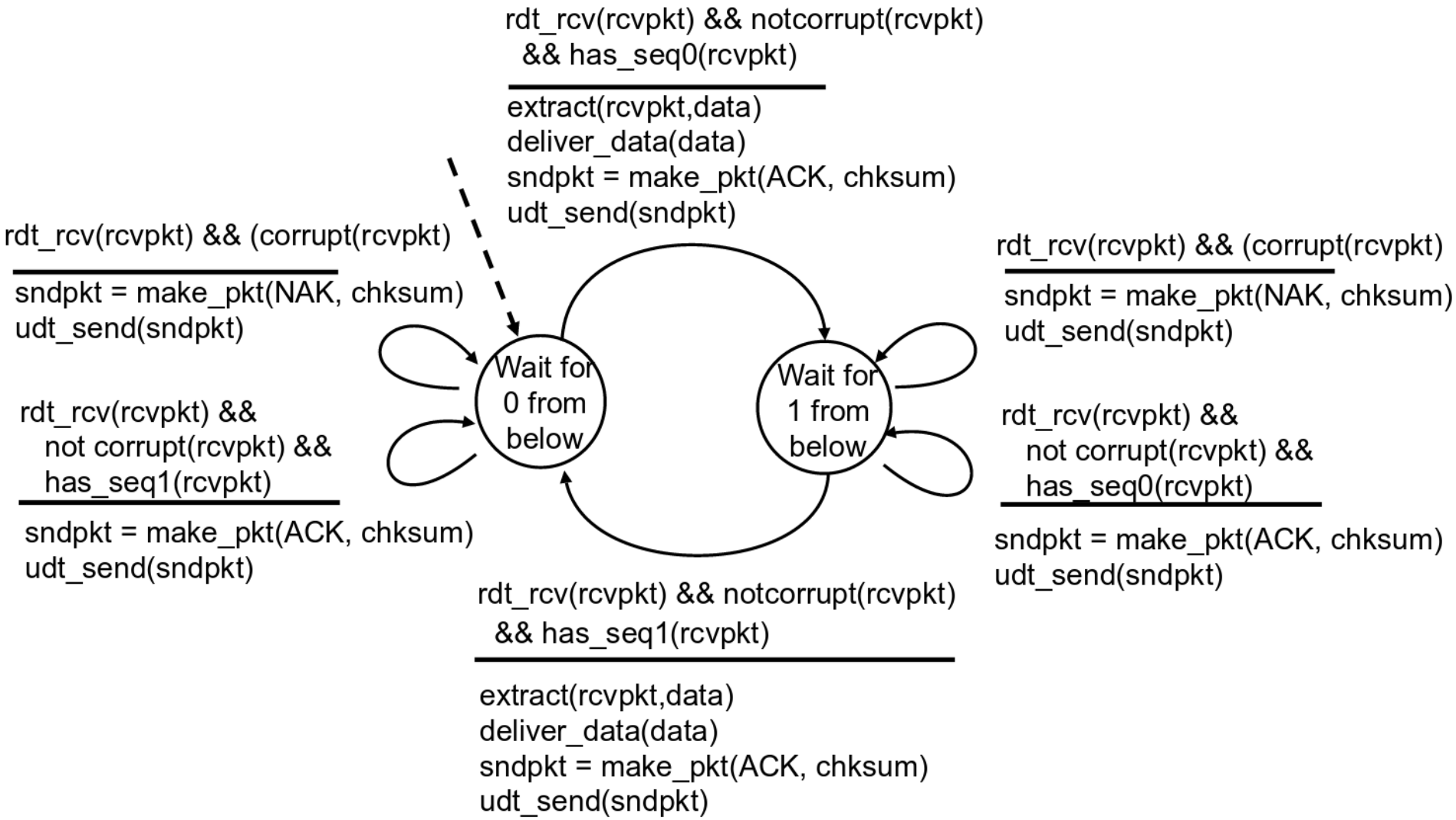
- Отправитель добавляет **номер пакета** (*sequence number, SN*), чередуя **0 и 1**
- Получатель принимает пакет с **ожидаемым номером** и меняет ожидаемый номер
- При получении **дубликата** он повторно отправляет ACK, но **не передаёт данные приложению**

Номер проверяется после проверки целостности пакета.

Протокол 2.1: отправитель



Протокол 2.1: получатель



Упражнение: протокол без NAK

Измените протокол 2.1 так, чтобы получатель отправлял **только ACK**, сохранив гарантии доставки.

- Какую информацию должны содержать подтверждения?
- Как изменятся автоматы отправителя и получателя?

Модель прежняя: сообщения могут повреждаться, но не теряются и не переупорядочиваются.

Подсказка и разбор – в конце презентации

Канал с повреждениями и потерями

- Теперь пакеты данных и подтверждения могут не только повреждаться, но и **теряться**
- Переупорядочивание по-прежнему исключаем

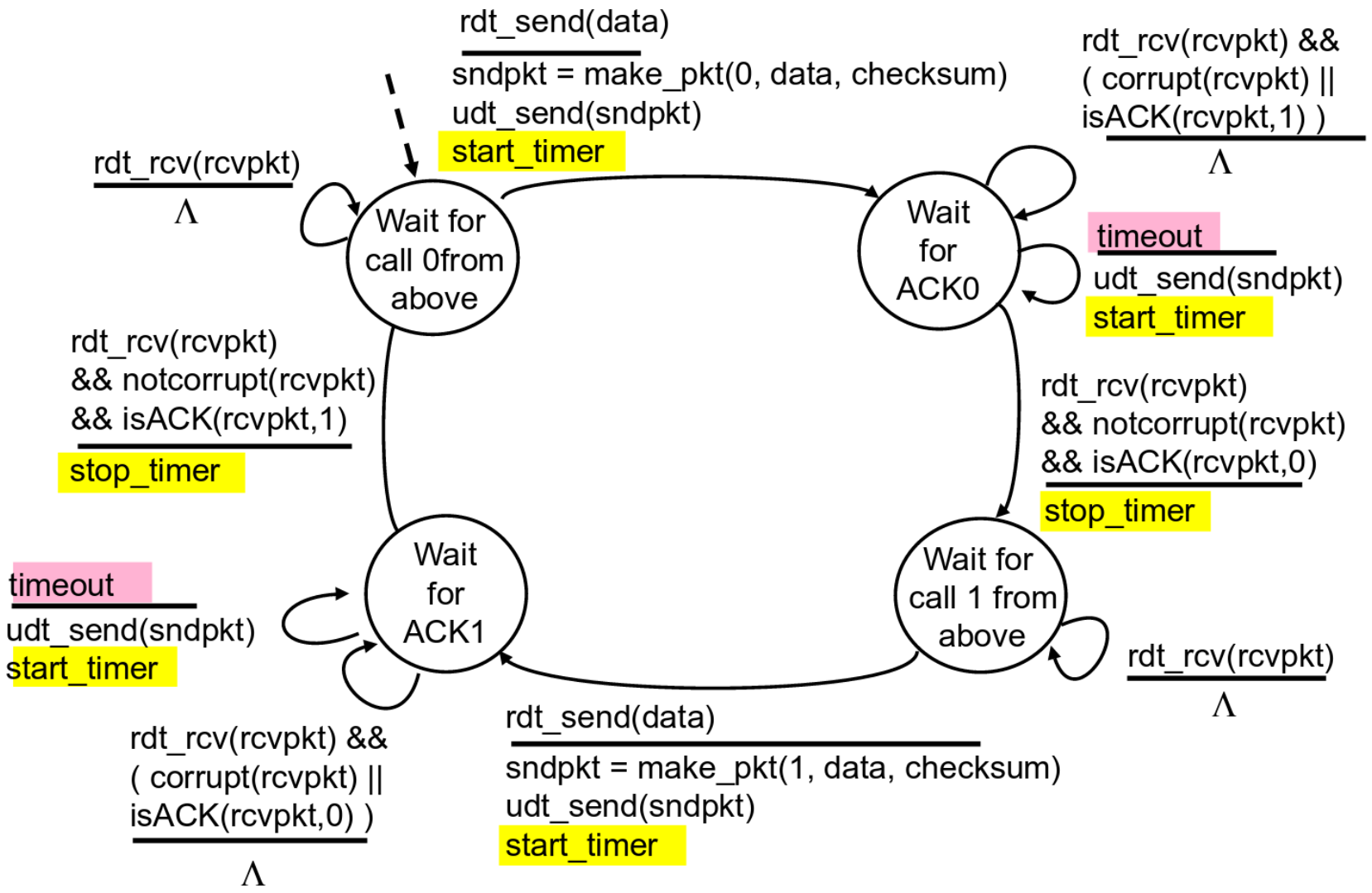
Как продолжить передачу, если ответ не приходит?

Тайм-аут и повторная отправка

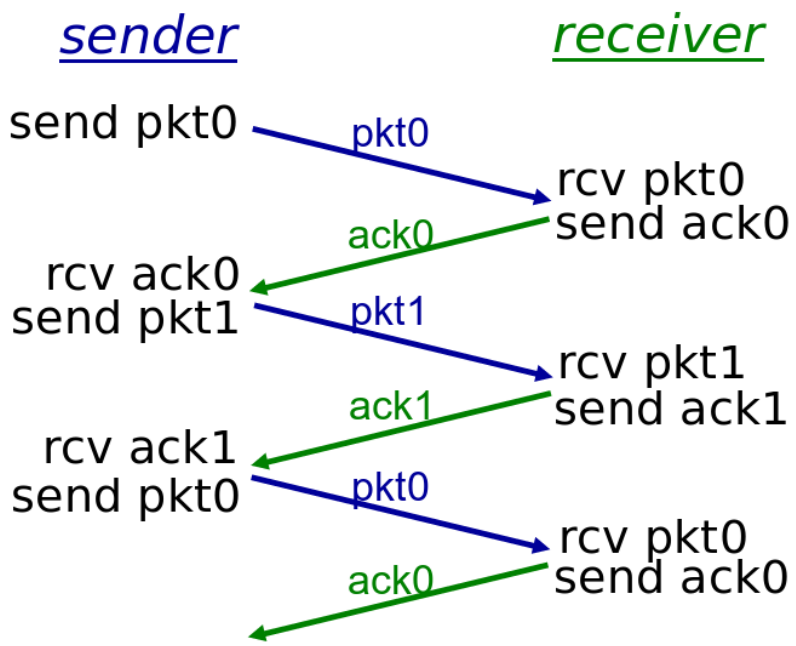
- После отправки пакета отправитель **запускает таймер**
- Если подтверждение текущего пакета не пришло вовремя, отправитель **повторяет пакет и перезапускает таймер**
- **Номер пакета в АСК** позволяет определить, какую передачу подтверждает ответ

Тайм-аут означает отсутствие своевременного ответа,
но не доказанную потерю

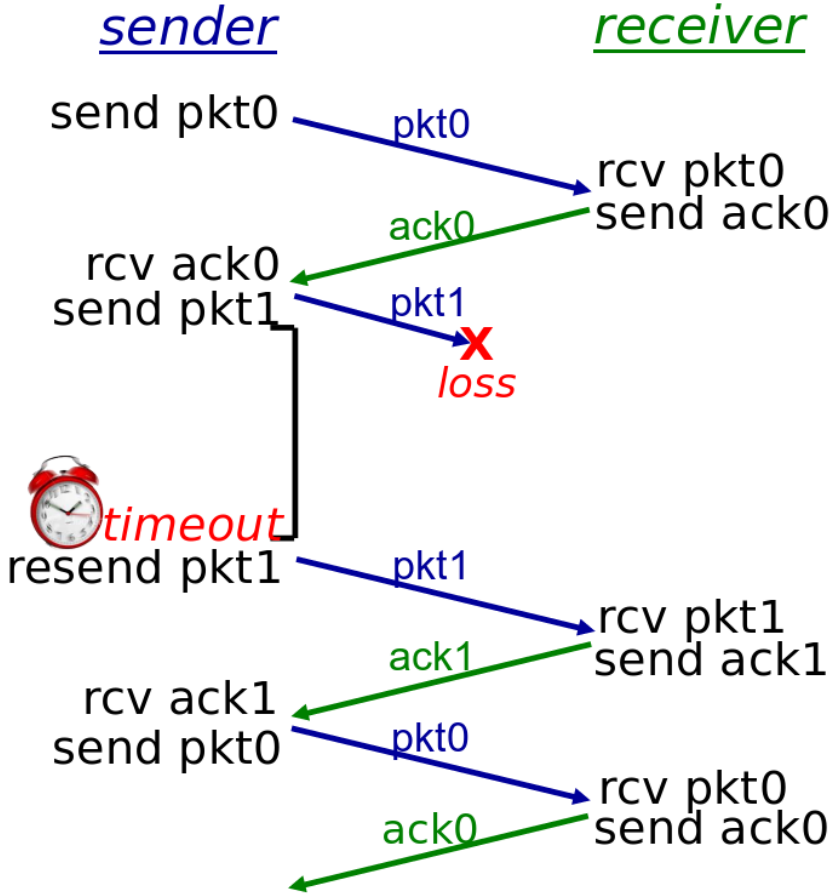
Протокол 3.0: отправитель



Протокол 3.0: передача без потерь и потеря пакета

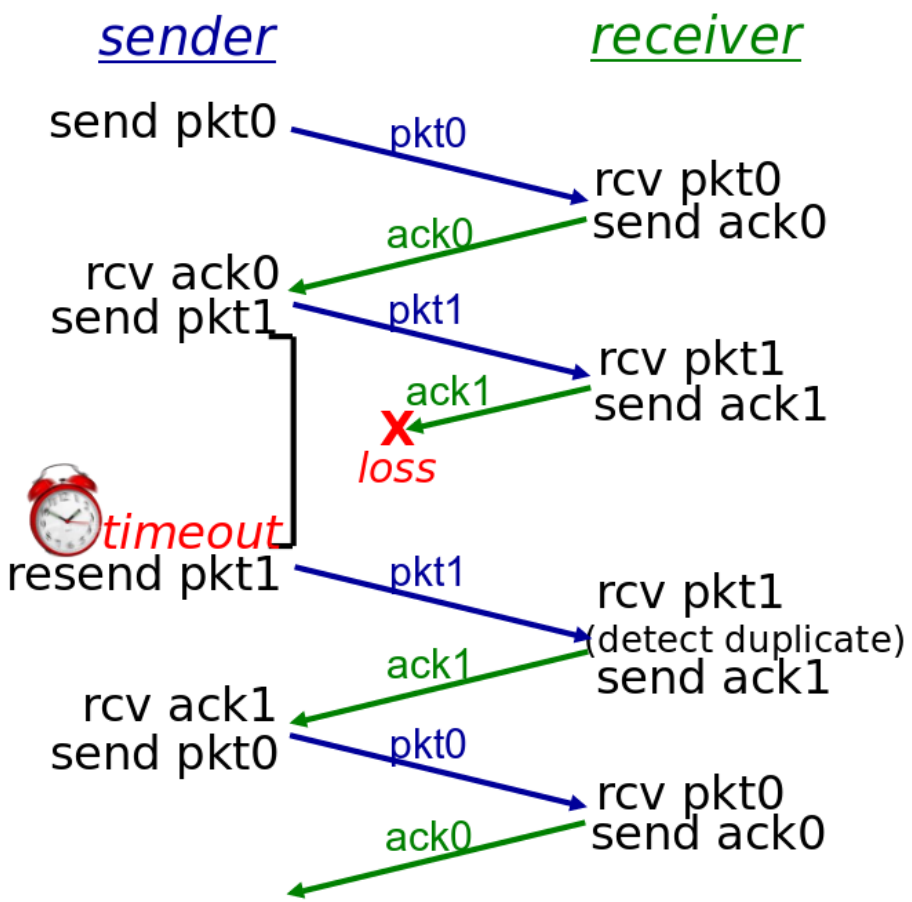


(a) no loss

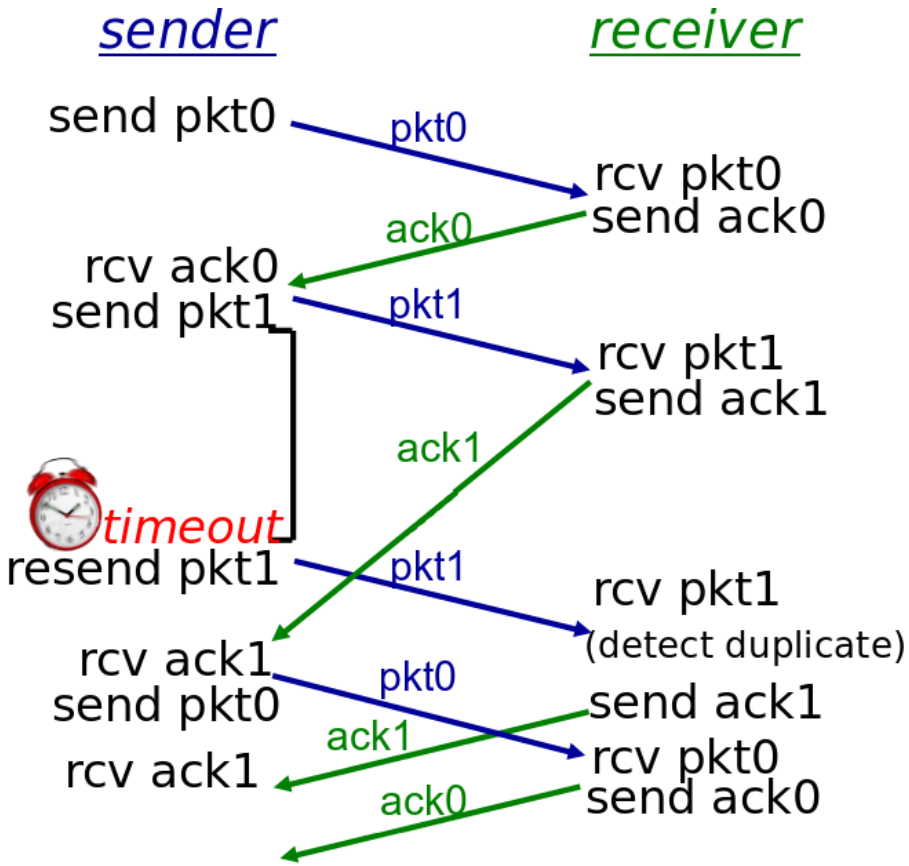


(b) packet loss

Протокол 3.0: потеря ACK и преждевременный тайм-аут



(c) ACK loss



(d) premature timeout/ delayed ACK

Упражнение: переупорядочивание сообщений

Теперь канал может **менять порядок доставки** пакетов данных и подтверждений.

Сохраняет ли протокол 3.0 гарантии доставки?

Постройте временную диаграмму, которая обосновывает ваш ответ.

Подсказка и разбор – в конце презентации

Гарантии передачи данных

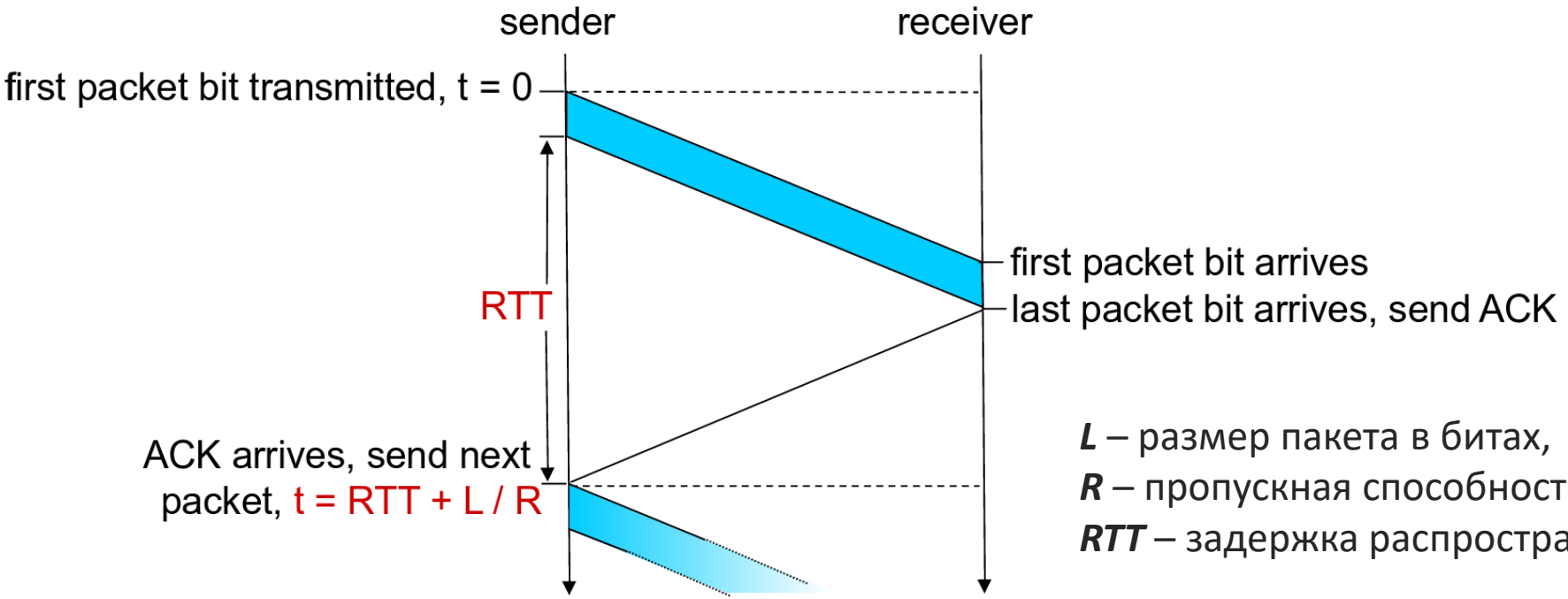
Целостность: обнаружение повреждений

Доставка сообщения приложению:

Семантика	**Число доставок**
Best effort	0 или более
At most once	0 или 1
At least once	1 или более
Exactly once	Ровно 1

Порядок: без гарантии порядка или в порядке отправки

Производительность stop-and-wait



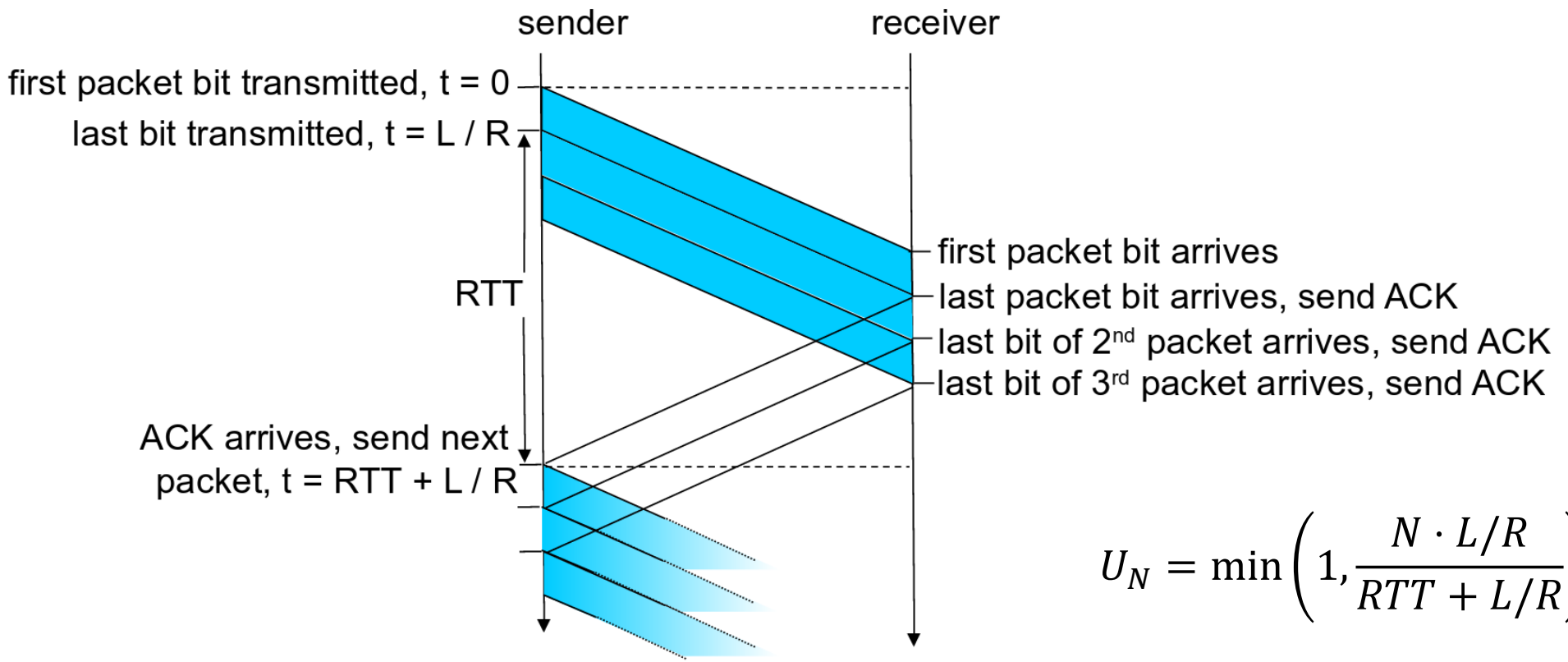
L – размер пакета в битах,
R – пропускная способность в бит/с,
RTT – задержка распространения туда и обратно.

Коэффициент использования канала: $U = \frac{L/R}{RTT + L/R}$

Если $RTT \gg L/R$, отправитель большую часть времени ждёт подтверждения.

(Допущения: нет потерь, временем обработки и передачи ACK пренебрегаем.)

Конвейерная передача (pipelining)

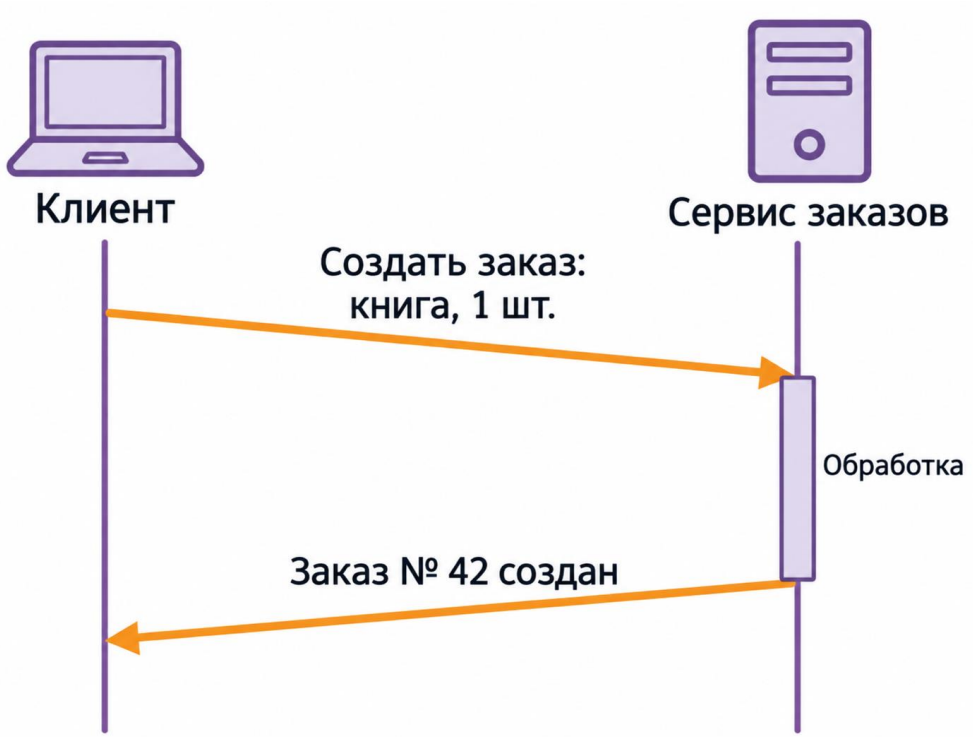


$$U_N = \min\left(1, \frac{N \cdot L/R}{RTT + L/R}\right)$$

- Отправитель может передать **до N пакетов, не дожидаясь подтверждений**
- Приход ACK позволяет продвинуть окно и отправить следующий пакет
- Коэффициент использования канала **увеличивается до N раз**, но не превышает 100 %.

Схема взаимодействия «запрос-ответ»

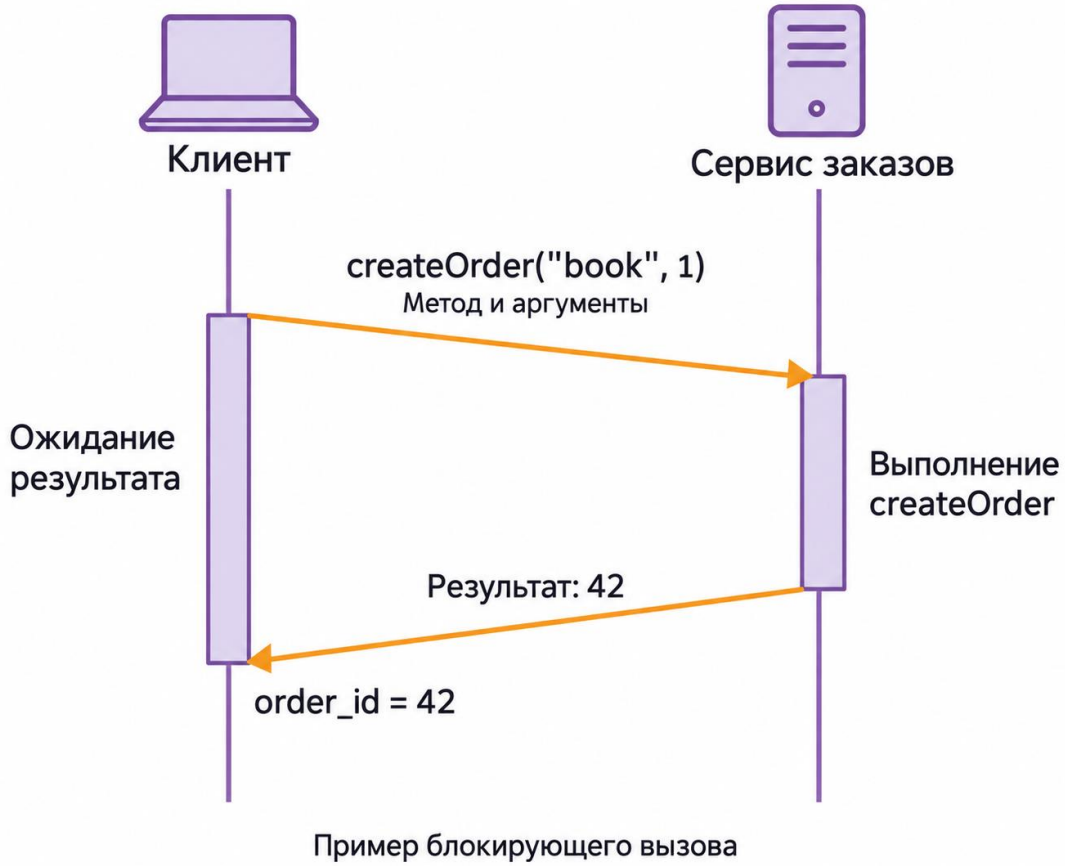
- **Клиент** отправляет запрос на выполнение операции
- **Сервер** обрабатывает запрос и возвращает результат или ошибку
- Каждый **ответ связан с конкретным запросом**



Удалённый вызов процедур (RPC)

- Вызов процедуры, выполняемой **в другом процессе**
- **Аргументы и результат** передаются в сообщениях
- **RPC-система** берёт на себя обмен сообщениями

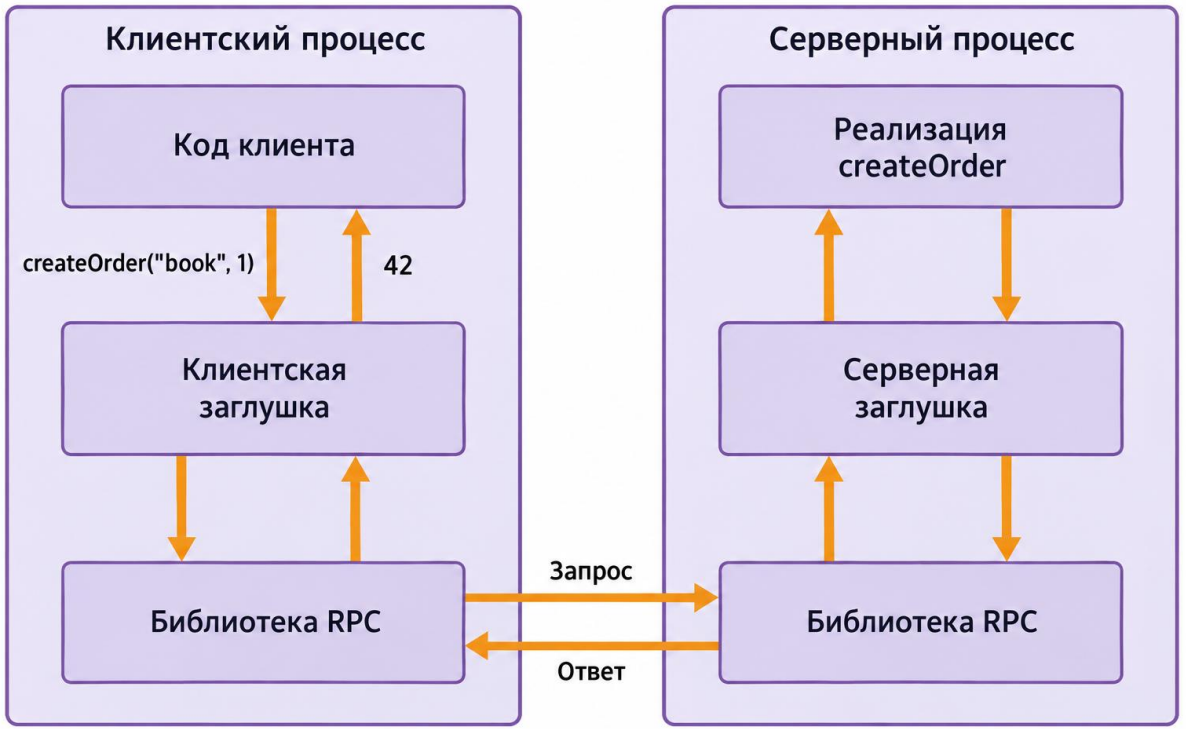
```
order_id = orders.createOrder("book", 1)
```



Пример блокирующего вызова

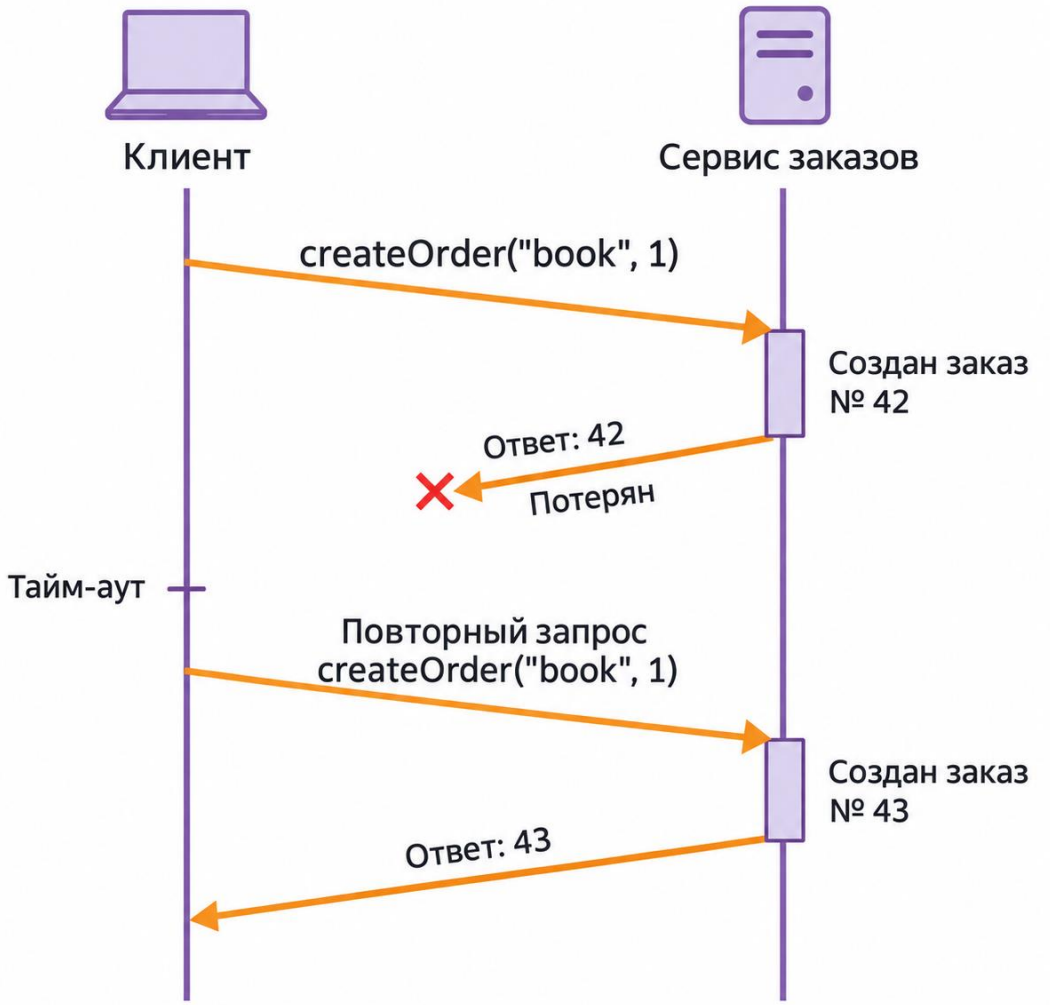
Реализация RPC

- **Клиентская заглушка** преобразует вызов в запрос и возвращает результат из ответа
- **Серверная заглушка** вызывает реализацию процедуры и формирует ответ
- **Сериализация** преобразует аргументы и результат в формат для передачи



Отказы и повторные вызовы RPC

- **Тайм-аут** не позволяет определить, выполнена ли операция
- **Повторный запрос** может привести к повторному выполнению
- Для обработки повторов используют **идентификаторы запросов и сохранённые ответы**



Без защиты от повторов

Гарантии выполнения RPC

Сколько раз выполняется операция для одного логического вызова?

Семантика	**Число выполнений**
Best effort	0 или более
At most once	0 или 1
At least once	1 или более
Exactly once	Ровно 1

Гарантии действуют в рамках принятой модели отказов.

Идемпотентные операции

Повторное выполнение с **теми же аргументами** имеет **тот же эффект**, что и однократное.

- Возвращаемые ответы могут различаться
- Идемпотентность упрощает безопасные повторы запросов

Отличия удалённых вызовов от локальных

- **Задержки и накладные расходы:** передача сообщений и сериализация
- **Частичные отказы:** результат вызова может остаться неизвестным
- **Раздельная память:** передача данных вместо обычных указателей
- **Независимое обновление:** совместимость версий клиента и сервера
- **Безопасность:** проверка участников и прав доступа, защита сообщений

RPC упрощает вызов, но не устраняет особенности распределённой системы.

Выводы

- **Модель взаимодействия** определяет зависимости между процессами
- **Завершение вызова** не всегда означает доставку или обработку сообщения
- **Гарантии доставки** зависят от модели отказов и требуют дополнительных ресурсов
- **Повторы RPC** требуют защиты от дублирования или идемпотентных операций

Что гарантирует завершение операции и что делать, если ответ не пришёл?

Материалы

- Distributed Systems (разделы 4.1-4.3)
- Computer Networking: A Top-Down Approach (глава 3)
- 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)

Протокол без NAK: решение

ACK(n) подтверждает пакет с номером n. Вместо NAK получатель повторяет последний ACK.

Состояние	Полученное сообщение	Действие
Отправитель ждёт ACK(n)	Неповреждённый ACK(n)	Перейти к следующим данным с номером 1−n
Отправитель ждёт ACK(n)	Повреждённый ACK или ACK(1−n)	Повторить текущий пакет n
Получатель ждёт пакет n	Неповреждённый пакет n	Передать данные приложению, отправить ACK(n), ждать 1−n
Получатель ждёт пакет n	Повреждённый пакет или дубликат 1−n	Повторить ACK(1−n), данные приложению не передавать

Начальное состояние: получатель ждёт пакет 0, сохранённое подтверждение – ACK1.
Модель: повреждения возможны, потерь и переупорядочивания нет.

Переупорядочивание: контрпример

- Пакет **A с номером 0 задерживается**. Его повторная копия приходит раньше
- После приёма **B с номером 1** получатель снова ожидает номер 0
- Старый пакет **A принимается как новый** и повторно доставляется приложению

Вывод: чередования 0 и 1 недостаточно при произвольном переупорядочивании.