

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ
ФЕДЕРАЦИИ ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

по выполнению лабораторных работ
по дисциплине «Распределённые системы управления в реальном времени»
для студентов
направления подготовки
09.03.04 Программная инженерия
Направленность (профиль)
«Инженерия сложных программных систем»

Ставрополь 2026

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ	3
Лабораторная работа 1. Установка и настройка среды для РВ-разработки.....	4
Лабораторная работа 2. Измерение задержек и джиттера в Linux (cyclictest).....	17
Лабораторная работа 3. Реализация периодической задачи в FreeRTOS.....	28
Лабораторная работа 4. Сравнение планировщиков RMA и EDF (симуляция).....	40
Лабораторная работа 5. Анализ CAN-трафика с помощью PCAN-View / Wireshark + CAN-USB.....	54
Лабораторная работа 6. Отправка и приём CAN-сообщений на STM32 с CAN-контроллером (или симуляция).....	67
Лабораторная работа 7. Программная эмуляция EtherCAT (SofEtherCAT или простейший симулятор).....	80
Лабораторная работа 8. Производительность EtherCAT vs стандартный TCP/IP.....	92
Лабораторная работа 9. Установка Fast DDS и создание Publisher/Subscriber.....	104
Лабораторная работа 10. Настройка QoS для гарантированной доставки (RELIABLE)....	118
Лабораторная работа 11. DDS с DEADLINE и LATENCY_BUDGET	131
Лабораторная работа 12. Распределённая система с несколькими участниками (Domain, Partition).....	143
Лабораторная работа 13. Синхронизация времени с PTP (Linux PTP).....	156
Лабораторная работа 14. Симуляция отказоустойчивой системы (Leader-Follower)	168
Лабораторная работа 15. Двухканальное резервирование (1oo2) на STM32 или симуляции	180
Лабораторная работа 16. Распределённая транзакция: Saga (хореография).....	190
Лабораторная работа 17. Интеграция DDS с ROS 2 (распределённая робототехника).....	202
Лабораторная работа 18. Итоговый проект: распределённая система управления группой симулированных роботов.....	220

ВВЕДЕНИЕ

Дисциплина «Распределённые системы управления в реальном времени» направлена на изучение принципов построения, анализа и практической реализации распределённых систем управления, функционирующих в условиях временных ограничений. В рамках лабораторных работ последовательно рассматриваются подготовка среды разработки реального времени, измерение задержек и джиттера, реализация периодических задач, анализ алгоритмов планирования, промышленный обмен по CAN и EtherCAT, middleware DDS, синхронизация времени, отказоустойчивость и интеграция распределённых компонентов в робототехнические системы.

Выполнение лабораторного практикума позволяет освоить полный цикл построения учебного решения для распределённой системы реального времени: настройку программного окружения, запуск измерительных инструментов, разработку симуляторов и узлов обмена сообщениями, настройку QoS-политик, анализ временных характеристик, проверку отказоустойчивости и оформление результатов экспериментов. Особое внимание уделяется воспроизводимости измерений, корректной интерпретации максимальных задержек, джиттера, потерь сообщений и поведения системы при отказах.

Настоящий документ объединяет лабораторные работы по дисциплине в единый файл. Сохранение исходной структуры и содержания отдельных работ необходимо для удобства последовательного изучения материала, выполнения индивидуальных заданий и подготовки к защите лабораторных работ.

Лабораторная работа 1. Установка и настройка среды для РВ-разработки

Цель работы: освоить установку и настройку рабочего окружения для разработки и тестирования приложений жёсткого реального времени на базе Linux. Подготовить дистрибутив Ubuntu 24.04 LTS с RT-ядром mainline-серии 6.12+ (с включённым CONFIG_PREEMPT_RT) либо виртуальную машину с выделенными CPU и закреплёнными прерываниями, собрать тулчейн GCC/CMake/GDB, развернуть POSIX-симулятор FreeRTOS 11.x, установить инструменты профилирования trace-cmd и cyclicttest и научиться интерпретировать первичные показатели задержек планировщика.

Теоретическое обоснование

1. Что такое среда РВ-разработки. Стандартный дистрибутив GNU/Linux является операционной системой общего назначения (GPOS) и оптимизирован под среднюю пропускную способность, а не под детерминизм отдельных задач. Для разработки приложений жёсткого реального времени требуется специально подготовленная инструментальная цепочка: вытесняющее ядро с предсказуемой реакцией на прерывания, набор средств измерения задержек, кросс-платформенный тулчейн для целевой ОСРВ и средства трассировки. Под средой РВ-разработки в рамках курса понимается комбинация из перечисленных ниже компонентов.

Таблица 1 – Состав рабочей среды для лабораторных работ курса

Компонент	Назначение	Пакет / источник
Ubuntu 24.04 LTS	Базовая ОС для хоста и виртуальных машин	ubuntu.com
Mainline-ядро Linux 6.12+ с CONFIG_PREEMPT_RT	Полностью вытесняемое ядро (RT в основной ветке с 12.2024)	linux-image-rt-generic либо сборка ядра 6.12+ с make menuconfig
GCC, binutils, make	Компилятор C/C++ и сборка нативных приложений	build-essential

Компонент	Назначение	Пакет / источник
CMake (≥ 3.22)	Кросс-платформенная система сборки	cmake
GDB	Отладчик нативных приложений	gdb
cyclicttest	Замер задержек пробуждения RT-таймера	rt-tests
stress-ng	Создание управляемой фоновой нагрузки	stress-ng
trace-cmd, KernelShark	Сбор и визуализация трасс ftrace	trace-cmd, kernelshark
FreeRTOS Kernel + POSIX-порт	Запуск FreeRTOS-задач в пользовательском пространстве Linux	github.com/FreeRTOS/FreeRTOS-Kernel

2. Ядро Linux с CONFIG_PREEMPT_RT. Основная цель режима PREEMPT_RT – минимизировать интервалы, в течение которых обработка задачи реального времени может быть отложена ядром. Достигается это за счёт трёх ключевых изменений: spinlock-и преобразуются в вытесняемые rt-mutex; обработчики прерываний выполняются в контексте отдельных потоков с настраиваемым приоритетом (threaded IRQs); пересмотрены критические секции RCU и таймерной подсистемы для допуска вытеснения. Исторически эти изменения распространялись отдельным patch-set-ом, однако начиная с Linux 6.12 (релиз декабря 2024 г.) код полностью принят в mainline-ядро и активируется одной конфигурационной опцией CONFIG_PREEMPT_RT=y. Никаких внешних патчей применять больше не нужно: достаточно собрать любое современное ядро 6.12+ с включённой опцией либо использовать готовый пакет дистрибутива (linux-image-rt-generic для Ubuntu 24.04, kernel-rt для RHEL/CentOS). Включение режима гарантирует ограниченную сверху worst-case-задержку планировщика, но не

уменьшает её среднюю величину – пропускная способность системы при этом, как правило, снижается на единицы процентов.

Если установить RT-ядро на хост-систему по каким-либо причинам нельзя (например, ноутбук с защищённой загрузкой и проприетарными драйверами), допустимо альтернативное решение – виртуальная машина с выделенными физическими ядрами. Хост-планировщик через `cpuset` или флаг `isolcpus=` закрепляет за ВМ набор ядер, на которые не приходят пользовательские задачи и прерывания. Внутри гостевой ОС включается тот же режим `CONFIG_PREEMPT_RT`. Подход даёт верхнюю границу latency порядка десятков микросекунд против единиц микросекунд на голом железе.

3. Основные показатели задержек. При оценке РВ-системы измеряют не среднюю производительность, а статистику задержек реакции на событие. Базовая величина – латентность пробуждения (`wake-up latency`) τ : интервал между моментом, когда таймер «должен» был разбудить задачу, и моментом, когда задача фактически получила процессор. По выборке N измерений строят набор показателей:

$$\tau_{min} = \min \tau_i, \quad \tau_{avg} = (1/N) \cdot \sum \tau_i, \quad \tau_{max} = \max \tau_i \quad (1)$$

Ключевым для жёсткого реального времени является именно τ_{max} – она задаёт минимально допустимый период задач и резерв на джиттер. Среднее значение τ_{avg} характеризует «обычное» поведение системы и обычно меняется слабо. Производная величина – джиттер J :

$$J = \tau_{max} - \tau_{min} \quad (2)$$

Опыт показывает, что для типового x86-сервера с ядром, собранным с `CONFIG_PREEMPT_RT`, τ_{max} при отсутствии нагрузки не превышает 30–50 мкс, под нагрузкой `stress-ng` – 80–150 мкс; ванильное ядро в тех же условиях легко выходит за миллисекунду.

4. `cyclictest`. Утилита `cyclictest` из пакета `rt-tests` эмулирует периодическую задачу: запускает один или несколько потоков с политикой `SCHED_FIFO`, каждый из которых засыпает на заданный интервал и после

пробуждения фиксирует фактическое отклонение от ожидаемого момента. Часто используемые ключи приведены в таблице 2.

Таблица 2 – Ключи cyclicttest, используемые в курсе

Ключ	Значение
-p PRIO	Приоритет SCHED_FIFO (1–99); 99 – максимум, 1 – минимум
-t N	Число измерительных потоков (по умолчанию равно числу CPU)
-i USEC	Базовый интервал пробуждения, мкс (по умолчанию 1000)
-l N	Сколько циклов выполнить и завершиться (0 – до прерывания)
-m	mlockall – заблокировать память процесса в RAM
-h MAX	Собирать гистограмму с бакетом 1 мкс до значения MAX
-q	Тихий режим: итоговая статистика только в конце
-a [LIST]	Привязать потоки к указанным CPU (affinity)
-D TIME	Длительность теста (например, 10m, 1h)

5. stress-ng. Утилита stress-ng (next generation) создаёт контролируемую фоновую нагрузку на различные подсистемы ядра и оборудования. В курсе используются несколько типичных «стрессоров»: --cpu N – N потоков с матричной арифметикой; --io N – дисковые операции через ядро; --vm N --vm-bytes 256M – аллокации и доступ к виртуальной памяти; --hdd N – запись и чтение временных файлов; --netdev N – нагрузка на сетевой стек. Эти стрессоры обнажают источники задержек ядра, которые в простое не видны.

6. FreeRTOS POSIX-порт. Отладка прикладной логики РВ-приложения далеко не всегда требует целевого микроконтроллера: для прохождения ранних этапов разработки достаточно симулятора. POSIX-порт FreeRTOS реализует планировщик и API ОСРВ как обычное пользовательское

приложение Linux, используя pthread и сигналы. Точные временные характеристики при этом не воспроизводятся, но функциональное поведение задач, очередей, семафоров и таймеров соответствует целевой платформе. Это позволяет проверить логику приложения на хосте, не дожидаясь поступления отладочной платы.

7. trace-cmd и ftrace. Подсистема ftrace, встроенная в ядро Linux, позволяет записывать события планировщика, прерываний и точек трассировки (tracpoint) ядра. Утилита trace-cmd – высокоуровневая обёртка над интерфейсом /sys/kernel/tracing. Минимально полезный набор команд: trace-cmd record -e sched_switch -e sched_wakeup <программа> – собрать события переключения контекста; trace-cmd report – вывести трассу в текстовом виде; KernelShark – графическая визуализация полученного файла trace.dat. Совместно с cyclicttest эти средства позволяют не только зафиксировать факт превышения задержки, но и установить её причину – например, длительный обработчик прерывания или работающий softirq.

Пример выполнения задания

Рассмотрим типовой сценарий подготовки рабочего места на чистой установке Ubuntu 24.04 LTS. Все команды выполняются от пользователя с правом sudo; перезагрузка после смены ядра обязательна.

Шаг 1. Установка базового тулчейна и средств измерения. Один вызов apt ставит все необходимые пакеты, включая отладчик, систему сборки и утилиты rt-tests/trace-cmd.

Листинг 1 – Установка тулчейна и инструментов профилирования

```
sudo apt update
sudo apt install -y \
    build-essential cmake gdb git \
    rt-tests stress-ng \
    trace-cmd kernelshark \
    linux-tools-common linux-tools-generic
```

Шаг 2. Установка RT-ядра. Начиная с Linux 6.12 (декабрь 2024 г.) поддержка реального времени принята в mainline-ядро, поэтому никаких

внешних патчей применять не требуется – нужно лишь ядро, собранное с CONFIG_PREEMPT_RT=y. В Ubuntu 24.04 LTS такое ядро поставляется мета-пакетом linux-image-rt-generic (доступен из штатных репозиториях, либо через подписку Ubuntu Pro для коммерческого использования). Альтернатива – собрать ядро 6.12+ из исходников kernel.org самостоятельно, выставив опцию Preemption Model → Fully Preemptible Kernel (Real-Time) в make menuconfig. После установки проверяем, что в загрузчике появилось RT-ядро, и перезагружаемся.

Листинг 2 – Установка готового RT-ядра и проверка

```
sudo apt install -y linux-image-rt-generic linux-headers-rt-generic
grep -E 'menuentry .*rt' /boot/grub/grub.cfg | head
sudo reboot
# после перезагрузки
uname -a # ожидаем строку, содержащую 'PREEMPT_RT'
```

Шаг 3. Настройка прав доступа для SCHED_FIFO. Чтобы cyclicttest и пользовательские RT-задачи могли запросить высокий приоритет, пользователь должен иметь право на rtprio и memlock. Стандартный способ – добавить пользователя в группу realtime и прописать лимиты:

Листинг 3 – Файл /etc/security/limits.d/realtime.conf

```
@realtime - rtprio 99
@realtime - memlock unlimited
@realtime - nice -20
```

После создания файла выполнить sudo groupadd -r realtime, sudo usermod -aG realtime \$USER и перелогиниться.

Шаг 4. Первичный замер задержек. Запускаем cyclicttest на 10 минут с одним потоком на каждом CPU, приоритетом 80 и шагом 1 мс. Ключ -m блокирует страницы памяти, -q подавляет промежуточный вывод.

Листинг 4 – Базовый прогон cyclicttest в простое

```
sudo cyclicttest -p 80 -t -m -i 1000 -D 10m -q
```

Ожидаемый вывод по завершении на типовом ноутбуке с RT-ядром приведён в листинге 5. Конкретные числа зависят от железа; критерий «среда настроена» – $\tau_{\max} \leq 100$ мкс в простое.

Листинг 5 – Пример итоговой строки cyclicttest

```
T: 0 ( 1234) P:80 I:1000 C: 600000 Min: 3 Act: 5 Avg: 6 Max: 42  
T: 1 ( 1235) P:80 I:1000 C: 600000 Min: 3 Act: 4 Avg: 6 Max: 51
```

Шаг 5. Сборка POSIX-симулятора FreeRTOS. Клонировем ядро и официальный пример Posix_GCC, собираем и запускаем простейший проект «Blinky», который попеременно выводит сообщения двух задач.

Листинг 6 – Получение и сборка FreeRTOS POSIX-демо

```
git clone --recurse-submodules \
  https://github.com/FreeRTOS/FreeRTOS.git ~/FreeRTOS
cd ~/FreeRTOS/FreeRTOS/Demo/Posix_GCC
make -j$(nproc)
./build/posix_demo
```

Если в выводе чередуются сообщения от задач Tx и Rx, симулятор FreeRTOS работает корректно.

Шаг 6. Снятие трассы планировщика. Запускаем trace-cmd на короткой загрузке (10 секунд stress-ng) и открываем результат в KernelShark для визуального анализа переключений контекста.

Листинг 7 – Снятие и визуализация трассы планировщика

```
sudo trace-cmd record -e sched_switch -e sched_wakeup \
  stress-ng --cpu 2 --timeout 10
sudo trace-cmd report | head -40
kernelshark trace.dat &
```

На этом базовая среда готова. В отчёте необходимо привести: вывод uname -a с подтверждением PREEMPT_RT, итоговую строку cyclicttest, вывод FreeRTOS POSIX-демо и фрагмент трассы sched_switch.

Задания по вариантам

Номер варианта определяется как $(N - 1) \bmod 15 + 1$, где N – порядковый номер студента в журнале группы. Конкретные числовые параметры задачи берутся из приведённой ниже таблицы. Составить отчет со скриншотами, показывающие выполнение заданий.

Вариант 1.

1. Установить на Ubuntu 24.04 пакет `linux-image-rt-generic`; перезагрузиться в RT-ядро и подтвердить наличие строки `PREEMPT_RT` в выводе `uname -a`.
2. Установить пакеты `rt-tests`, `stress-ng`, `trace-cmd`, `build-essential`, `cmake`, `gdb`; зафиксировать версии каждого инструмента в журнале эксперимента.
3. Снять $\tau_{\min}$, τ_{avg} , $\tau_{\max}$ в простое (`cyclicttest -p 80 -t -m -i 1000 -D 5m -q`) и под нагрузкой `stress-ng --cpu $(nproc)` длительностью 5 минут; сравнить значения.
4. Подготовить отчёт: лог `uname -a`, версии пакетов, две итоговые строки `cyclicttest`, вывод о пригодности среды к выполнению последующих лабораторных работ.

Вариант 2.

1. Подготовить виртуальную машину QEMU/KVM с двумя выделенными физическими CPU через параметр загрузки хоста `isolcpus=2,3`.
2. Внутри гостевой Ubuntu 24.04 включить `mainline`-ядро с `CONFIG_PREEMPT_RT` (`linux-image-rt-generic`) и установить `rt-tests`.
3. Снять `cyclicttest` на изолированных ядрах (ключ `-a 2,3`) и на общих ядрах; сравнить $\tau_{\max}$ в двух конфигурациях.
4. Построить таблицу: «конфигурация $\times \tau_{\text{avg}} \times \tau_{\max}$ »; сделать вывод о выигрыше от изоляции CPU.

Вариант 3.

1. Скачать исходники ядра Linux 6.12 (или новее) с kernel.org; включить в `make menuconfig` опцию Preemption Model → Fully Preemptible Kernel (Real-Time), убедиться, что задействован `CONFIG_PREEMPT_RT=y`.
2. Собрать ядро (`make -j$(nproc)` и `make modules_install, install`); зафиксировать продолжительность сборки.
3. Перезагрузиться в самосборное ядро; снять `cyclictst -p 80 -t -m -i 1000 -D 10m` в простое.
4. Сравнить $\tau_{\max}$ самосборного ядра с готовым пакетом `linux-image-rt-generic`; объяснить расхождение, если оно есть (разные `.config`, разные версии).

Вариант 4.

1. Установить две конфигурации одной и той же Ubuntu 24.04: одну с ванильным `linux-image-generic`, вторую с `linux-image-rt-generic`.
2. На обеих системах снять `cyclictst` с приоритетами `SCHED_OTHER`, `SCHED_FIFO 50` и `SCHED_FIFO 99` по 10 минут каждый.
3. Построить таблицу 2×3 «ядро × приоритет» с $\tau_{\max}$ в каждой ячейке.
4. Объяснить, почему на ванильном ядре переход с `SCHED_FIFO 50` на 99 почти не влияет на $\tau_{\max}$, а на RT-ядре – даёт ощутимый выигрыш.

Вариант 5.

1. Установить тулчейн `gcc-arm-none-eabi` (из пакета `gcc-arm-none-eabi`) и `openocd`; проверить версии.
2. Скачать FreeRTOS Kernel 11.x с github.com/FreeRTOS/FreeRTOS; выбрать демо для Cortex-M3 (`qemu_mps2_an385`) или аналогичную плату.
3. Собрать демо через `make` или `CMake`; запустить полученный `elf` под QEMU (`qemu-system-arm -M mps2-an385 -kernel ...`).
4. Привести вывод QEMU с диагностикой задач FreeRTOS; оформить отчёт с `CMakeLists.txt` либо `Makefile` минимального проекта.

Вариант 6.

1. Развернуть POSIX-порт FreeRTOS 11.x (FreeRTOS/Demo/Posix_GCC); собрать и запустить штатное демо.
2. Расширить демо третьей задачей, которая получает сообщения из новой очереди QueueHandle_t и печатает их через printf.
3. Подтвердить корректность работы: вывод чередуется во времени, очередь не переполняется при разных приоритетах задач.
4. Привести фрагменты main.c с созданием задач, очереди и vTaskStartScheduler; вывод программы за 10 секунд.

Вариант 7.

1. Настроить ftrace вручную через /sys/kernel/tracing (без trace-cmd): выбрать current_tracer=nop, включить события sched_switch, sched_wakeup.
2. На фоне работающего cyclicttest -p 80 -t -m -i 1000 -D 5m записать трассу в течение 5 секунд (echo 1 > tracing_on; sleep 5; echo 0 > tracing_on).
3. Выгрузить trace в файл; найти переключения контекста cyclicttest и сопоставить с пиками задержки.
4. Оформить фрагмент trace как приложение к отчёту; объяснить значения колонок (TASK-PID, CPU, TIMESTAMP, FUNCTION).

Вариант 8.

1. На системе с RT-ядром включить latency-tracer wakeup_rt (echo wakeup_rt > /sys/kernel/tracing/current_tracer).
2. Запустить cyclicttest на 10 минут под нагрузкой stress-ng --cpu \$(nproc); зафиксировать tracing_max_latency.
3. Извлечь содержимое /sys/kernel/tracing/trace для максимального выброса; расшифровать стек источника задержки.
4. В отчёте описать: значение максимальной задержки, имя процесса/функции, вызвавшей её, способ устранения (изоляция, повышение приоритета, отключение драйвера).

Вариант 9.

1. Запустить `cyclictest` с базовыми интервалами 100, 500, 1000, 5000, 10000 мкс на одном и том же RT-ядре в простое.
2. Каждый прогон – 10 минут; сохранить итоговую строку с τ_{avg} и τ_{max} .
3. Построить график зависимости τ_{avg} и τ_{max} от интервала (логарифмическая ось X).
4. Объяснить наблюдаемые тенденции: рост τ_{avg} при малых интервалах из-за насыщения, выход τ_{max} на плато на больших.

Вариант 10.

1. Установить Docker и базовый образ `ubuntu:24.04`; внутри контейнера поставить `rt-tests`, `stress-ng`, `trace-cmd`.
2. Запустить `cyclictest` внутри контейнера с привилегиями `--cap-add=SYS_NICE --ulimit rtprio=99 --ulimit memlock=-1`.
3. Подтвердить, что `cyclictest` получает `SCHED_FIFO` (`chrt -p PID`); снять τ_{max} за 10 минут.
4. Сравнить полученные числа с прямым запуском `cyclictest` на хосте; оценить, на сколько Docker увеличивает τ_{max} .

Вариант 11.

1. Установить TuneD (пакет `tuned`) и активировать профиль `realtime` либо `realtime-virtual-guest`; зафиксировать список изменённых параметров `sysctl`.
2. Привязать прерывания сетевой карты (`eth0` / `enp*`) к одному ядру через `/proc/irq/<N>/smp_affinity_list`, RT-нагрузку – к другому.
3. Подтвердить смену через `/proc/interrupts`: счётчики прерываний должны расти только на выбранном ядре.
4. Снять `cyclictest` до и после применения TuneD; сравнить τ_{max} и сделать вывод о вкладе настройки в стабильность системы.

Вариант 12.

1. Установить hwlatdetect (входит в rt-tests); ознакомиться с описанием тестируемых аппаратных задержек (SMI, NMI).
2. Запустить hwlatdetect --duration=30m --threshold=10 на хосте без полезной нагрузки.
3. По полученным выбросам отделить аппаратные задержки (видны через hwlat) от программных (видны через cyclicttest, но не видны через hwlat).
4. В отчёте привести: вывод hwlatdetect, тот же интервал cyclicttest, вывод о соотношении вкладов SMI/программных задержек на данной машине.

Вариант 13.

1. Собрать POSIX-порт FreeRTOS 11.x двумя способами: штатный make и CMake (написать CMakeLists.txt с нуля).
2. Зафиксировать время сборки для каждого способа (команда time make / time cmake --build).
3. Сравнить размеры итогового исполняемого файла и таблицу ld-секций (size, readelf -S).
4. Привести в отчёте полный CMakeLists.txt минимального проекта и сравнительную таблицу.

Вариант 14.

1. Развернуть Ubuntu 24.04 Server на Raspberry Pi 4; собрать или найти RT-ядро для arm64 (linux-image-raspi-rt либо самосборное 6.12+ с CONFIG_PREEMPT_RT).
2. Снять cyclicttest в простое и под нагрузкой stress-ng --cpu 4 на RPi 4 длительностью 10 минут.
3. Повторить те же замеры на x86-машине из ЛР 1.
4. Построить сводную таблицу «платформа × нагрузка × $\tau_{\max}$ »; объяснить, почему arm-машина даёт другую границу $\tau_{\max}$ при сопоставимых архитектурных частотах.

Вариант 15.

1. Установить пакеты `linux-tools-common` и `linux-tools-generic` для своего ядра; проверить доступность `perf` (`perf --version`).
2. Запустить параллельно `perf sched record` и `cyclicttest -p 80 -t -m -i 1000 -D 5m` под нагрузкой `stress-ng --cpu $(nproc)`.
3. Сформировать отчёт `perf sched latency`; найти максимальную задержку перепланирования.
4. Сопоставить максимум `perf sched latency` с $\tau_{\max}$ из `cyclicttest`; объяснить расхождение через различие методик измерения.

Лабораторная работа 2. Измерение задержек и джиттера в Linux (cyclicttest)

Цель работы: освоить методику количественной оценки задержек и джиттера планировщика Linux с помощью утилиты cyclicttest. Провести серию измерений на ванильном и RT-ядре (mainline 6.12+ с включённым CONFIG_PREEMPT_RT) при различных приоритетах и уровнях фоновой нагрузки, создаваемой stress-ng. Построить гистограммы распределения задержек, сравнить минимальное, среднее и максимальное значения для каждой конфигурации и сделать выводы об ограничениях ванильного ядра и о выигрыше, который даёт режим PREEMPT_RT.

Теоретическое обоснование

1. Источники задержек в Linux. Любая задача с приоритетом p в Linux может быть отложена ядром по нескольким независимым причинам: выполнение более приоритетной задачи (preemption), обработка аппаратного прерывания (hard IRQ), обработка отложенного прерывания (softirq, tasklet, workqueue), работа в критической секции, защищённой спинлоком, и операции миграции/планирования ядра. Совокупный вклад этих причин и определяет наблюдаемую задержку пробуждения, которую и измеряет cyclicttest.

2. Модель измерения. cyclicttest порождает один или несколько потоков с политикой SCHED_FIFO и заданным приоритетом. На итерации k поток просит ядро разбудить его в абсолютный момент $t_k = t_0 + k \cdot \Delta$ через `clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME)`. После возврата из системного вызова он читает текущее время t'_k и вычисляет фактическую задержку:

$$\tau_k = t'_k - t_k \quad (1)$$

Значение τ_k всегда неотрицательно. Программа поддерживает счётчик циклов C , ведёт скользящие min/avg/max и при включённом ключе `-h` –

собирает гистограмму $H[u]$ – число итераций, попавших в бакет u микросекунд.

3. Влияние приоритета. На ванильном ядре приоритет `SCHED_FIFO` позволяет вытеснять обычные `SCHED_OTHER`-задачи, но не вытесняет обработку прерываний и спинлок-секции ядра. В режиме `PREEMPT_RT` (включается опцией `CONFIG_PREEMPT_RT` в ядре 6.12+) обе категории становятся преемптируемыми, поэтому повышение приоритета имеет прямой эффект: чем выше p , тем меньше категорий, которые могут отложить задачу. На практике это означает, что разница $\tau_{\max}$ между $p = 50$ и $p = 99$ на ванильном ядре пренебрежима, а на `RT`-ядре может составлять десятки микросекунд.

4. Влияние фоновой нагрузки. Утилита `stress-ng` позволяет целенаправленно усиливать конкретный источник задержек. В лабораторной работе используются четыре характерных сценария, приведённые в таблице 1.

Таблица 1 – Сценарии нагрузки `stress-ng` для измерения задержек

Команда <code>stress-ng</code>	Какой источник нагружает
<code>--cpu N</code>	Вычислительные ядра; растёт давление на планировщик
<code>--io N</code>	Системные вызовы <code>sync</code> ; растёт работа в <code>kernel space</code>
<code>--vm N --vm-bytes 256M</code>	Управление виртуальной памятью; <code>TLB shootdown</code>
<code>--hdd N --hdd-bytes 1G</code>	Подсистема ввода-вывода; <code>softirq</code> , обработчики дисков
<code>--netdev N</code>	Сетевой стек; высокая частота <code>softirq</code> <code>NET_RX</code>

На ванильном ядре все эти сценарии способны вытолкнуть $\tau_{\max}$ за 1 мс. На `RT`-ядре с правильным закреплением аффинности ухудшение по сравнению с простым обычно не превышает 2-3 раз.

5. Гистограмма задержек. Гистограмма $H[u]$ позволяет отделить редкие выбросы от устойчивой границы. Для системы жёсткого реального времени важна не средняя величина, а правый «хвост» распределения: 99-й, 99.9-й и наибольший процентиля. Полезные производные показатели:

$$p99 = \min\{u : \sum_{v \leq u} H[v] \geq 0.99 \cdot N\}, \quad J = \tau_{\max} - \tau_{\min} \quad (2)$$

Дополнительно вычисляют стандартное отклонение σ и соотношение $\tau_{\max} / \tau_{\text{avg}}$ – грубая оценка худшего случая относительно типичного. Для RT-ядра в стабильной системе типичные значения: $\tau_{\text{avg}} \approx 5\text{--}10$ мкс, $p99 \approx 20\text{--}30$ мкс, $\tau_{\max} \leq 100$ мкс под нагрузкой.

6. Корректность методики. Чтобы измерения были воспроизводимы, необходимо: запускать `cyclictest` от `root` либо с правами на `rtprio/memlock`; всегда указывать ключ `-m (mlockall)` для исключения задержек из-за подкачки страниц; проводить тест длительностью не менее 10 минут для устойчивости статистики; в одной серии сравнивать только одинаковые длительности; снимать показания при одной и той же температуре процессора (после прогрева). Параметры теста, BIOS, частоту CPU, `governor` и список запущенных сервисов обязательно фиксировать в журнале эксперимента.

7. Визуализация. Гистограмма `cyclictest` выгружается ключом `-h MAX -q`, при этом строки вида «`<μs> <count_T0> <count_T1>`» пишутся в `stdout`. Эти данные удобно построить в виде ступенчатого графика или столбчатой диаграммы в логарифмическом масштабе по оси Y – это даёт возможность одновременно видеть и плотную среднюю часть распределения, и редкие выбросы. В курсе для визуализации используется Python + `matplotlib`.

Пример выполнения задания

В качестве разбора проведём сравнительный замер на одной и той же машине под ванильным и RT-ядром (`linux-image-rt-generic` в Ubuntu 24.04). Перезагрузка в нужное ядро выполняется через `grub`. Все запуски – с одинаковыми параметрами `cyclictest`, длительность 10 минут, гистограмма до 400 мкс.

Шаг 1. Калибровка в простое. Перед серией измерений нужно убедиться, что система действительно «тиха»: закрыты браузер, IDE, отключены автообновления. Запускаем `cyclictest` без нагрузки и сохраняем гистограмму в файл.

Листинг 1 – Базовый запуск `cyclictest` с выгрузкой гистограммы

```
sudo cyclictest -p 80 -t -m -i 1000 -D 10m -h 400 -q \  
> idle.txt
```

Шаг 2. Серия под нагрузкой. Запускаем фоновую нагрузку и сразу после неё – `cyclictest` с теми же параметрами. Длительность `stress-ng` выбирается больше длительности теста на 30 секунд, чтобы исключить эффект «начала и конца».

Листинг 2 – Замер под CPU-нагрузкой

```
stress-ng --cpu $(nproc) --timeout 11m &  
STRESS_PID=$!  
sudo cyclictest -p 80 -t -m -i 1000 -D 10m -h 400 -q \  
> load_cpu.txt  
kill $STRESS_PID
```

Шаг 3. Парсинг гистограммы. В первых строках вывода `cyclictest` идут итоговые Min/Avg/Max, далее с пустой строки – таблица гистограммы. Простой Python-скрипт извлекает данные и считает процентиля.

Листинг 3 – Скрипт `parse_cyclictest.py`

```
import sys  
from collections import defaultdict  
  
def parse(path):  
    hist = defaultdict(int)  
    with open(path) as f:  
        for line in f:  
            parts = line.split()  
            if len(parts) < 2:  
                continue  
            if not parts[0].isdigit():  
                continue  
            us = int(parts[0])  
            for c in parts[1:]:  
                hist[us] += int(c)
```

```

return hist

def percentile(hist, q):
    total = sum(hist.values())
    target = q * total
    acc = 0
    for us in sorted(hist):
        acc += hist[us]
        if acc >= target:
            return us
    return max(hist)

for path in sys.argv[1:]:
    h = parse(path)
    print(f'{path}: p50={percentile(h, 0.5)} '
          f'p99={percentile(h, 0.99)} '
          f'p999={percentile(h, 0.999)} '
          f'max={max(h)}')
```

Листинг 4 – Пример запуска парсера

```

$ python3 parse_cyclictest.py idle.txt load_cpu.txt
idle.txt:  p50=5 p99=14 p999=27 max=42
load_cpu.txt: p50=8 p99=32 p999=78 max=128
```

Шаг 4. Построение гистограммы. Полученные данные строятся столбчатой диаграммой в логарифмическом масштабе по оси Y, чтобы одновременно были видны и плотный центр распределения, и хвост редких выбросов.

Листинг 5 – Построение гистограммы matplotlib

```

import matplotlib.pyplot as plt
from parse_cyclictest import parse

for label, path in [('idle', 'idle.txt'),
                    ('cpu', 'load_cpu.txt')]:
    h = parse(path)
    xs = sorted(h)
    ys = [h[x] for x in xs]
    plt.bar(xs, ys, width=1.0, alpha=0.5, label=label)

plt.yscale('log')
plt.xlabel('latency, us')
plt.ylabel('count')
plt.legend()
plt.savefig('hist.png', dpi=150)
```

Шаг 5. Сравнение ядер. Та же серия повторяется после загрузки в ванильное ядро. Результаты сводятся в одну таблицу. Типичный вид для ноутбука с Intel i5 приведён в таблице 2.

Таблица 2 – Пример результатов сравнения ядер, мкс

Конфигурация	p50	p99	Max
Vanilla, idle	6	28	317
Vanilla, --cpu	9	120	2840
Vanilla, --hdd	8	180	5120
RT (PREEMPT_RT), idle	5	14	42
RT (PREEMPT_RT), -- cpu	8	32	128
RT (PREEMPT_RT), -- hdd	9	55	210

Шаг 6. Выводы. На ванильном ядре $\tau_{\max}$ уверенно выходит за 1 мс при любой нетривиальной нагрузке, что делает его непригодным для задач с дедлайнами в сотни микросекунд. RT-ядро обеспечивает верхнюю границу порядка 100–200 мкс под аналогичной нагрузкой, сохраняя при этом среднее значение на уровне единиц микросекунд. Полученные показатели p99 и max должны быть приведены в отчёте по каждой комбинации (ядро × нагрузка × приоритет) с гистограммами и коротким объяснением.

Задания по вариантам

Номер варианта определяется как $(N - 1) \bmod 15 + 1$, где N – порядковый номер студента в журнале группы. Конкретные числовые параметры задачи берутся из приведённой ниже таблицы. Составить отчет со скриншотами, показывающие выполнение заданий.

Вариант 1.

5. Снять базовые показатели `cyclictest -p 80 -t -m -i 1000 -D 10m -h 400 -q` в простое на ванильном ядре и на RT-ядре.

6. Повторить замер под нагрузкой `stress-ng --cpu $(nproc)` длительностью 11 минут.
7. Построить гистограммы (логарифмическая ось Y) для всех четырёх конфигураций на одном графике.
8. Сделать вывод: во сколько раз RT-ядро снижает $\tau_{\max}$ под нагрузкой по сравнению с ванильным.

Вариант 2.

5. Запустить `cyclictest` на RT-ядре в простое с приоритетами `SCHED_FIFO` 1, 20, 50, 80, 99 по 10 минут каждый.
6. Собрать в таблицу для каждой p тройку τ_{avg} , $p99$, $\tau_{\max}$.
7. Построить ступенчатый график $\tau_{\max}(p)$ и $p99(p)$ на одной оси.
8. Объяснить наблюдаемую тенденцию: с какого приоритета задача перестаёт «чувствовать» `kthread`-и ядра.

Вариант 3.

5. На RT-ядре снять `cyclictest -p 80 -t -m -i 1000 -D 10m` под четырьмя типами нагрузки: `stress-ng --cpu, --io, --vm --vm-bytes 256M, --hdd --hdd-bytes 1G`.
6. Извлечь $p50$, $p99$, $\tau_{\max}$ для каждой конфигурации; свести в таблицу.
7. Варьировать N – число потоков `stress-ng` – в значениях 1, 2, $nproc/2$, $nproc$; построить график $\tau_{\max}(N)$ для каждого типа.
8. Сделать вывод: какой подсистемой ядра определяется худший случай $\tau_{\max}$ на данной машине.

Вариант 4.

5. Зафиксировать поток `cyclictest` на одном CPU (например, -a 2), запустить `stress-ng --cpu N` на остальных ядрах через `taskset -c`.
6. Сравнить $\tau_{\max}$ с конфигурацией без явной аффинности (оба запущены по умолчанию).
7. Добавить параметр загрузки `isolcpus=2` в командную строку ядра, перезагрузиться, повторить замер.

8. Построить таблицу «конфигурация $\times \tau_{\max}$ »; объяснить вклад аффинности и изоляции CPU отдельно.

Вариант 5.

5. Снять `cyclictest` с базовыми интервалами 100, 500, 1000, 5000, 10000 мкс под одинаковой фоновой нагрузкой `stress-ng --cpu $(nproc)` по 10 минут каждый.

6. Извлечь τ_{avg} и τ_{max} ; построить график зависимости от интервала (логарифмическая ось X).

7. Подтвердить, что при интервале 100 мкс ядро `fttrace` начинает конкурировать с `rt`-поток (через `perf top`).

8. Сделать вывод: какой минимальный период задачи практически достижим на данной машине.

Вариант 6.

5. Переключить `cpufreq governor` в режим `performance` (`cpupower frequency-set -g performance`); снять `cyclictest` 10 минут.

6. Переключить в `powersave`; повторить замер.

7. Дополнительно проверить режимы `ondemand` и `schedutil`.

8. Свести в таблицу «`governor` $\times \tau_{\text{max}}$ $\times$ средняя частота CPU»; объяснить, почему `powersave` ухудшает τ_{max} в несколько раз (время выхода из P-state).

Вариант 7.

5. Запустить `cyclictest` на 1 час в простое; сохранить полный лог с ключом `-v` (каждое измерение).

6. Разбить выборку на 10-минутные окна, посчитать τ_{max} в каждом окне.

7. Построить график $\tau_{\text{max}}(\text{окно})$ и оценить разброс между окнами.

8. Сделать вывод: устоявшаяся ли это статистика, либо требуются более длительные прогоны для стабильных оценок худшего случая.

Вариант 8.

5. На RT-ядре под нагрузкой `stress-ng --cpu $(nproc)` запустить `cyclicttest` с политикой `SCHED_OTHER`, `SCHED_RR` (приоритет 50), `SCHED_FIFO` (приоритет 50).
6. Зафиксировать $\tau_{\max}$ для каждой политики при одной и той же длительности 10 минут.
7. Объяснить разницу через формальную семантику классов: квант `SCHED_OTHER`, переключение по таймеру у `SCHED_RR`.
8. Сделать вывод о пригодности `SCHED_OTHER` для задач с жёсткими дедлайнами на основе экспериментальных данных.

Вариант 9.

5. Подготовить два раздела с файловыми системами `ext4` и `xfs` (можно через `файлы-loop`).
6. Запустить `stress-ng --hdd 2 --hdd-bytes 1G --temp-path <точка_монтирования>` для каждой ФС; параллельно `cyclicttest` 10 минут.
7. Сравнить $\tau_{\max}$ для `ext4` и `xfs`; зафиксировать частоту `fsync` и размер журнала.
8. Объяснить разницу через активность журналирования (`jbd2/kworker-` потоки); привести команду `iostat` в подтверждение.

Вариант 10.

5. Снять `cyclicttest` 10 минут в простое с настройками питания по умолчанию.
6. Перезагрузиться с параметром ядра `intel_idle.max_cstate=1` (или `processor.max_cstate=1` для AMD); повторить замер.
7. Дополнительно отключить C-states через BIOS, если есть возможность; снять третий замер.
8. Построить столбчатую диаграмму «настройка C-state $\times$ $\tau_{\max}$ »; объяснить, во сколько раз глубокие C-state увеличивают латентность пробуждения и почему.

Вариант 11.

5. На 8-ядерной (или большей) машине запустить `cyclictest` при $N = 1, 2, 4, 8$ потоках (ключ `-t N`).
6. Для каждого N снять $\tau_{\max}$ каждого потока отдельно и общий $p99$ по выборке.
7. Построить таблицу « $N \times \max\text{-по-потокам} \times \text{среднее } \max \times p99$ ».
8. Объяснить, почему общий $p99$ ухудшается медленнее, чем максимум по потокам, при росте N .

Вариант 12.

5. Под нагрузкой `stress-ng --netdev N` снять `cyclictest` с двумя конфигурациями приоритетов: (а) $p = 30$ (ниже `chrt -p $(pidof ksoftirqd/0)`); (б) $p = 80$ (выше `ksoftirqd`).
6. Зафиксировать $\tau_{\max}$ и количество `irqs/sec` через `/proc/interrupts` для обеих конфигураций.
7. Привязать сетевые прерывания к выделенному CPU через `smp_affinity_list`; повторить замер.
8. Сделать вывод о роли `threaded IRQs` и приоритета `ksoftirqd` на RT-ядре.

Вариант 13.

5. Развернуть KVM-гостя с Ubuntu 24.04 и RT-ядром; провести `cyclictest` 10 минут внутри ВМ при `isolcpus=2,3` на хосте и `vsrpin` гостя к ядрам 2,3.
6. Снять те же замеры на хосте без виртуализации.
7. Снять третий вариант: гость без `isolcpus`, `vsrpi` плавают по хосту.
8. Построить таблицу «среда $\times \tau_{\max}$ »; оценить «налог виртуализации» на латентность.

Вариант 14.

5. Запустить `cyclictest` 5 минут под нагрузкой `stress-ng --cpu $(nproc)`; параллельно записать `trace-cmd record -e sched_switch -e irq_handler_entry -e irq_handler_exit`.

6. По завершении найти момент максимального выброса $\tau_{\max}$ в выводе `cyclictest`; сопоставить с `timestamp` в `trace-cmd report`.
7. Открыть `trace.dat` в `KernelShark`, визуально найти цепочку событий, которая привела к выбросу.
8. Описать в отчёте: какой `irq` / какой процесс задержал `rt`-поток на $\tau_{\max}$ микросекунд; предложить меру устранения.

Вариант 15.

5. Установить пакет `rt-app` (`apt install rt-app`); ознакомиться с форматом JSON-описания задач.
6. Сформулировать одну и ту же задачу (10 мс период, 1 мс вычислений) для `rt-app` и для `cyclictest`; запустить обе по 10 минут на `RT`-ядре под нагрузкой.
7. Сравнить распределения задержек: гистограмма из `cyclictest` и `perf_thread` из `rt-app`.
8. Сделать вывод: какие дополнительные характеристики (блокировки, очереди, `IPC`) можно моделировать `rt-app`, и почему `cyclictest` остаётся базовым инструментом для оценки ядра.

Лабораторная работа 3. Реализация периодической задачи в FreeRTOS

Цель работы: освоить корректную реализацию периодических задач в FreeRTOS 11.x на примере POSIX-симулятора. Реализовать задачу с фиксированным периодом, используя API `xTaskDelayUntil`, измерить фактический период между активациями и оценить его джиттер. Сравнить поведение `xTaskDelayUntil` и `vTaskDelay` при возникновении дополнительной нагрузки на планировщик и показать, что `vTaskDelay` приводит к накапливающемуся сдвигу фазы (drift), а `xTaskDelayUntil` – нет.

Теоретическое обоснование

1. Периодическая задача в ОСРВ. Периодической называется задача, которая должна активироваться с фиксированным интервалом T и укладывать своё выполнение в этот же интервал. Если k -я активация ожидается в момент $t_k = t_0 + k \cdot T$, то идеальная задача удовлетворяет двум условиям одновременно: фактический момент начала очередной итерации $t'_k \rightarrow t_k$ (нулевой джиттер фазы), и сумма времени работы одной итерации $S_k \leq T$ (нет накопления отставания). На практике t'_k всегда чуть больше t_k из-за задержек планировщика; задача разработчика – не давать этому отставанию накапливаться от итерации к итерации.

2. Тики ядра FreeRTOS. Время в FreeRTOS измеряется в тиках – целочисленных интервалах длиной $\text{configTICK_RATE_HZ}^{-1}$. По умолчанию $\text{configTICK_RATE_HZ} = 1000$, то есть один тик равен 1 мс. Текущее значение счётчика тиков возвращает функция `xTaskGetTickCount()`; пересчёт миллисекунд в тики выполняется макросом `pdMS_TO_TICKS(ms)`. Все функции задержки оперируют только тиками, поэтому минимально различимый период равен одному тiku и точность периода ограничена сверху частотой тика.

3. `vTaskDelay` против `vTaskDelayUntil`. FreeRTOS предоставляет две функции для приостановки задачи на заданный интервал. Сигнатуры приведены ниже, а семантика – в таблице 1.

Листинг 1 – Сигнатуры функций задержки FreeRTOS

```
void vTaskDelay( const TickType_t xTicksToDelay );
BaseType_t xTaskDelayUntil( TickType_t * const pxPreviousWakeTime,
                             const TickType_t xTimeIncrement );
```

Таблица 1 – Сравнение функций задержки FreeRTOS

Свойство	vTaskDelay	xTaskDelayUntil
Опорный момент	Момент вызова функции	Сохранённый момент предыдущего пробуждения
Действие	Спать xTicksToDelay тиков, начиная с «сейчас»	Спать до момента *pxPreviousWakeTime + xTimeIncrement
Поведение при перегрузке итерации	Период де-факто = T + ε; ε накапливается	Период удерживается равным T; ε компенсируется
Допустимая длительность работы	Любая, но дрейф растёт	C _k < T строго; иначе функция вернёт pdFALSE
Назначение	«Отдохнуть немного», аperiodические паузы	Периодические задачи с жёстким периодом

4. Накопление дрейфа при vTaskDelay. Пусть очередная итерация начинается в момент t_k и работает C_k тиков, после чего вызывается vTaskDelay(T). Следующая активация произойдёт не в момент $t_k + T$, а в момент $t_k + C_k + T$. Через N итераций сдвиг фазы относительно идеального расписания составит:

$$\Delta_N = \sum_{k=1..N} C_k \quad (1)$$

Это означает, что любая, даже сколь угодно малая, длительность полезной работы превращает vTaskDelay в систему с линейно растущим отставанием. Через минуту работы при $T = 10$ мс и $C_k \approx 200$ мкс фаза задачи уходит на 6 секунд, что в системах реального времени неприемлемо.

5. Корректная схема vTaskDelayUntil. Функция xTaskDelayUntil получает указатель на переменную, в которой хранится момент последнего

планового пробуждения, и инкремент периода. После возврата переменная обновляется до значения $*pxPreviousWakeTime + xTimeIncrement$. Канонический шаблон периодической задачи:

Листинг 2 – Канонический шаблон периодической задачи

```
void vPeriodicTask( void *pv )
{
    TickType_t xLastWakeTime = xTaskGetTickCount();
    const TickType_t xPeriod = pdMS_TO_TICKS( 10 );

    for( ;; )
    {
        /* полезная работа */
        doWork();

        xTaskDelayUntil( &xLastWakeTime, xPeriod );
    }
}
```

В отличие от `vTaskDelay`, накопления нет: если итерация затянулась на ε , следующая будет короче на ε , а среднее значение периода сходится к T точно.

6. Джиттер периода. На реальной аппаратуре и в POSIX-симуляторе фактические моменты пробуждения смещены относительно идеальных на величину δ_k . Для оценки качества задачи измеряют интервалы между пробуждениями $P_k = t'_k - t'_{k-1}$ и считают:

$$P_{avg} = (1/N) \cdot \sum P_k, \quad J = \max P_k - \min P_k \quad (2)$$

Хорошая периодическая задача удовлетворяет двум условиям: $P_{avg} \approx T$ (нет систематического дрейфа) и J мал по сравнению с T . На POSIX-симуляторе FreeRTOS типичные значения при $T = 10$ мс: $P_{avg} \approx 10000$ мкс с точностью до микросекунды; $J \approx 50\text{--}200$ мкс под нагрузкой хоста; при `vTaskDelay` P_{avg} растёт пропорционально C_k , а J заметно увеличивается.

7. POSIX-порт FreeRTOS. Симулятор реализует планировщик FreeRTOS поверх pthread Linux: каждая FreeRTOS-задача становится pthread, тики генерируются сигналом SIGALRM, переключение контекста –

стандартными примитивами pthread. Точные временные характеристики целевого микроконтроллера симулятор не воспроизводит – он наследует джиттер хост-планировщика Linux. Тем не менее, функциональное поведение API FreeRTOS совпадает с реальной платформой: правильная реализация периодической задачи на симуляторе будет правильной и на STM32 или ESP32.

Пример выполнения задания

В качестве разбора реализуем периодическую задачу с периодом 10 мс, на каждой итерации печатающую timestamp фактического пробуждения. По собранному журналу вычислим P_{avg} и джиттер, затем повторим тот же эксперимент с vTaskDelay вместо vTaskDelayUntil.

Шаг 1. Подготовка проекта. За основу берётся каталог FreeRTOS/Demo/Posix_GCC, клонированный в ЛР 1. Создаём в нём копию main.c (например, main_periodic.c), правим Makefile и заменяем демо-задачи на одну периодическую.

Шаг 2. Реализация задачи. Файл main_periodic.c содержит точку входа, создание задачи и саму периодическую функцию. Время измеряется CLOCK_MONOTONIC, чтобы не зависеть от перевода системных часов.

Листинг 3 – Файл main_periodic.c (фрагмент)

```
#include <stdio.h>
#include <time.h>
#include "FreeRTOS.h"
#include "task.h"

#define PERIOD_MS    10U
#define ITERATIONS  1000U

static long long now_us( void )
{
    struct timespec ts;
    clock_gettime( CLOCK_MONOTONIC, &ts );
    return ( long long )ts.tv_sec * 1000000LL
        + ts.tv_nsec / 1000LL;
}
```

```

static void vPeriodicTask( void *pv )
{
    ( void )pv;
    TickType_t xLastWake = xTaskGetTickCount();
    const TickType_t xPeriod = pdMS_TO_TICKS( PERIOD_MS );

    for( unsigned i = 0; i < ITERATIONS; ++i )
    {
        printf( "%u %lld\n", i, now_us() );
        xTaskDelayUntil( &xLastWake, xPeriod );
    }
    vTaskEndScheduler();
}

int main( void )
{
    xTaskCreate( vPeriodicTask, "per",
                configMINIMAL_STACK_SIZE * 4,
                NULL, tskIDLE_PRIORITY + 2, NULL );
    vTaskStartScheduler();
    return 0;
}

```

Шаг 3. Сборка и запуск. Из каталога Posix_GCC запускаем сборку, выходные данные перенаправляем в файл.

Листинг 4 – Сборка и снятие журнала

```

make -j$(nproc)
./build/posix_demo > delay_until.log

```

Шаг 4. Анализ журнала. Простой Python-скрипт считает разности между соседними строками и выводит P_avg, P_min, P_max и джиттер.

Листинг 5 – Скрипт analyze_period.py

```

import sys

ts = []
with open(sys.argv[1]) as f:
    for line in f:
        _, t = line.split()
        ts.append(int(t))

diffs = [ts[i] - ts[i - 1] for i in range(1, len(ts))]
avg = sum(diffs) / len(diffs)
print(f'N    = {len(diffs)}')
print(f'min  = {min(diffs)} us')
print(f'avg  = {avg:.1f} us')

```

```

print(f'max = {max(diffs)} us')
print(f'J   = {max(diffs) - min(diffs)} us')
print(f'drift= {ts[-1] - ts[0] - (len(ts) - 1) * 10000} us')

```

Листинг 6 – Пример вывода для vTaskDelayUntil

```

$ python3 analyze_period.py delay_until.log
N   = 999
min  = 9856 us
avg  = 10000.1 us
max  = 10184 us
J    = 328 us
drift= 76 us

```

Шаг 5. Контрольный эксперимент с vTaskDelay. Заменяем вызов xTaskDelayUntil на vTaskDelay(xPeriod), пересобираем, запускаем и сохраняем журнал в delay.log.

Листинг 7 – Замена API и повторный прогон

```

// в main_periodic.c:
vTaskDelay( xPeriod );
// сборка и запуск
make -j$(nproc) && ./build/posix_demo > delay.log
python3 analyze_period.py delay.log

```

Листинг 8 – Пример вывода для vTaskDelay

```

N   = 999
min  = 10003 us
avg  = 10247.3 us
max  = 10612 us
J    = 609 us
drift= 247000 us

```

Шаг 6. Интерпретация. У xTaskDelayUntil средний период равен номиналу 10000 мкс с точностью до микросекунды и итоговый дрейф за 1000 итераций не превышает сотен микросекунд. У vTaskDelay средний период систематически больше номинала, а накопленный сдвиг фазы за то же количество итераций исчисляется уже сотнями миллисекунд – ровно те самые накапливающиеся S_k из формулы (1). Это и есть причина, по которой в периодических задачах используется только xTaskDelayUntil.

В отчёте необходимо привести: исходник `main_periodic.c`, оба журнала с выводом скрипта анализа, гистограммы распределения интервалов P_k для обоих API и краткий вывод о причинах наблюдаемого дрейфа.

Задания по вариантам

Номер варианта определяется как $(N - 1) \bmod 15 + 1$, где N – порядковый номер студента в журнале группы. Конкретные числовые параметры задачи берутся из приведённой ниже таблицы. Составить отчет со скриншотами, показывающие выполнение заданий.

Вариант 1.

9. Реализовать на POSIX-порту FreeRTOS 11.x одну периодическую задачу с $T = 10$ мс на 1000 итераций; журналировать `timestamp` каждого пробуждения через `CLOCK_MONOTONIC`.
10. Снять P_{avg} , P_{min} , P_{max} , J и накопленный дрейф для версии с `xTaskDelayUntil`.
11. Заменить вызов на `vTaskDelay(xPeriod)`; повторить эксперимент.
12. Сравнить два API в одной таблице; объяснить разницу через формулу (1) накопления дрейфа.

Вариант 2.

9. Реализовать задачу с $T = 5$ мс; внутри итерации – `busy-loop` длительностью ≈ 1 мс (откалибровать по `clock_gettime`).
10. Запустить версии с `xTaskDelayUntil` и `vTaskDelay` по 2000 итераций каждая.
11. Подтвердить, что версия с `vTaskDelay` даёт дрейф ≈ 1 мс на итерацию (итоговый ≈ 2 с), а `xTaskDelayUntil` удерживает среднее = 5 мс.
12. Построить график «номер итерации $\rightarrow$ разность фактический – идеальный момент пробуждения» для обоих API.

Вариант 3.

9. Реализовать две независимые задачи с $T_1 = 10$ мс и $T_2 = 25$ мс, обе на `xTaskDelayUntil`, с одинаковым приоритетом.
10. Журналировать пробуждения обеих задач в одном файле с меткой ID; разделить логи в Python.
11. Снять P_{avg} и J для каждой задачи; убедиться, что взаимное влияние минимально (T_2 – не кратно T_1).
12. Изменить периоды на $T_1 = 10$ мс, $T_2 = 20$ мс (кратные); снова снять J – объяснить, почему джиттер уменьшается.

Вариант 4.

9. При `configTICK_RATE_HZ = 1000` реализовать задачу с заявленным $T = 2$ мс (`pdMS_TO_TICKS(2) = 2` тика).
10. Снять распределение фактических интервалов P_k за 5000 итераций; построить гистограмму с бакетом 100 мкс.
11. Подтвердить, что значения сосредоточены около $2000 \text{ мкс} \pm 1 \text{ тик}$ (1000 мкс).
12. Объяснить эффект через дискретизацию таймера ядра FreeRTOS и момент начала отсчёта относительно ближайшего тика.

Вариант 5.

9. Собрать симулятор с `configTICK_RATE_HZ = 200` (тик = 5 мс); запустить ту же задачу $T = 10$ мс.
10. Собрать с `configTICK_RATE_HZ = 1000` и `configTICK_RATE_HZ = 10000`; запустить тот же эксперимент.
11. Зафиксировать минимально достижимый период (когда P_{avg} стабильно близок к `pdMS_TO_TICKS`) и наблюдаемый J для каждой частоты тика.
12. Построить таблицу «частота тика $\times T_{min} \times J$ »; сделать вывод о компромиссе между разрешением и оверхедом тика.

Вариант 6.

9. Реализовать задачу A с $T = 10$ мс, приоритет `tskIDLE_PRIORITY + 1`, `xTaskDelayUntil`.
10. Параллельно запустить задачу B с тем же приоритетом и бесконечным `busy-loop` (без задержек).
11. Снять J задачи A; повторить, переведя A на приоритет `tskIDLE_PRIORITY + 2`.
12. Объяснить рост джиттера при равных приоритетах через `round-robin` внутри одного приоритетного уровня в FreeRTOS.

Вариант 7.

9. Реализовать задачу с $T = 10$ мс и случайным $C_k \in [1, 9]$ мс (`rand() % 9 + 1`, `busy-loop` на нужную длительность).
10. Снять `P_avg` и J на 1000 итерациях с `xTaskDelayUntil`.
11. Подтвердить, что среднее `P_avg` ≈ 10 мс ровно до тех пор, пока $C_k < T$; зафиксировать дрейф = 0.
12. Расширить диапазон C_k до $[1, 15]$ мс; показать переход в режим срыва дедлайнов и рост `P_avg` выше номинала.

Вариант 8.

9. Реализовать задачу с $T = 10$ мс и заведомо избыточной длительностью $C \approx 12$ мс (постоянный `busy-loop`).
10. Зафиксировать возвращаемое значение `xTaskDelayUntil` (типа `BaseType_t`): должен возвращать `pdFALSE`.
11. Подсчитать долю итераций, где задача переходит в режим «гонки» (`pdFALSE`) за 1000 циклов.
12. Описать в отчёте механизм: что происходит с `xLastWakeTime` после возврата `pdFALSE` и почему задача далее работает «без сна».

Вариант 9.

9. Реализовать две задачи с одинаковым $T = 10$ мс на `xTaskDelayUntil`, фазы стартуют с разницей 5 мс (вторая задача после `xTaskCreate` вызывает `vTaskDelay(pdMS_TO_TICKS(5))` перед циклом).
10. Журналировать оба пробуждения; визуально подтвердить, что разность фаз стабильно ≈ 5 мс на 2000 итераций.
11. Снять P_{avg} и J для каждой задачи; убедиться, что обе удерживают период.
12. Повторить с разницей фазы 7 мс; показать, что устойчивость сохраняется при любом начальном сдвиге.

Вариант 10.

9. Реализовать задачу с $T = 1$ мс (`configTICK_RATE_HZ = 1000`) на 5000 итераций.
10. Снять P_{avg} , P_{max} , J на хосте без фоновой нагрузки.
11. Запустить параллельно `stress-ng --cpu $(nproc)` длительностью 10 секунд; повторить замер.
12. Сравнить два прогона: построить две гистограммы P_k на одном графике; оценить рост J под нагрузкой и сделать вывод о минимальном применимом периоде в данной среде.

Вариант 11.

9. Базовая версия: задача с $T = 10$ мс, внутри которой – `busy-loop` 5 мс; снять P_{avg} , J , дрейф.
10. Заменить `busy-loop` на `vTaskDelay(pdMS_TO_TICKS(5))` для имитации блокирующей операции; повторить замер.
11. Заменить на `vTaskDelayUntil` от собственного локального `xLastWake`; повторить замер.
12. Объяснить, почему накопленный дрейф у задачи с внутренней блокировкой меньше, чем у чистого `busy-loop`, даже при внешнем `vTaskDelay` на основном цикле.

Вариант 12.

9. Реализовать задачу-продюсер с $T = 20$ мс, пишущую timestamp пробуждения в очередь QueueHandle_t (xQueueSend, timeout = 0).
10. Реализовать задачу-логгер пониженного приоритета, вычитывающую очередь (xQueueReceive, бесконечный wait) и пишущую в файл через fprintf.
11. Снять P_{avg} и J продюсера; зафиксировать, что вывод файла не вносит джиттер.
12. Уменьшить размер очереди до 1; показать, что задача-продюсер начинает блокироваться, дрейф растёт.

Вариант 13.

9. Реализовать задачу с $T = 10$ мс; через каждые 100 итераций искусственно «потерять» период – вызвать busy-loop на 15 мс перед xTaskDelayUntil.
10. Журналировать все пробуждения; подсчитать число пропусков ($P_k > 1.5 \cdot T$).
11. Подтвердить, что после пропуска следующий P_k короче номинала – задача догоняет график.
12. Построить график $P_k(k)$ на интервале $[k_{\text{пропуска}} - 5, k_{\text{пропуска}} + 5]$; описать форму восстановления.

Вариант 14.

9. Реализовать вариант А: задача с xTaskDelayUntil, $T = 10$ мс.
10. Реализовать вариант В: software-таймер xTimerCreate с тем же периодом 10 мс, autoReload = pdTRUE, колбэк той же длительности.
11. Снять P_{avg} и J для обоих вариантов по 1000 итераций.
12. Сравнить два подхода в таблице; объяснить, в каких случаях программный таймер предпочтительнее задачи (короткий колбэк, много мелких периодов).

Вариант 15.

9. Реализовать «строгую» схему: задача с $T = 10$ мс, каждая итерация перед `xTaskDelayUntil` проверяет, не превышает ли $(xTaskGetTickCount() - xLastWakeTime)$ значения $2 \cdot xPeriod$; при превышении – инкрементировать счётчик пропусков дедлайна.
10. Снять статистику пропусков без фоновой нагрузки на 5000 итераций.
11. Запустить параллельно `stress-ng --cpu $(nproc)`; снять статистику пропусков под нагрузкой.
12. Построить таблицу «нагрузка $\times$ число пропусков $\times \max(P_k)$ »; сделать вывод о применимости POSIX-порта FreeRTOS для жёстких дедлайнов 10 мс на нагруженной системе.

Лабораторная работа 4. Сравнение планировщиков RMA и EDF (симуляция)

Цель работы: сравнить два классических алгоритма планирования задач в системах реального времени – Rate Monotonic (RMA) и Earliest Deadline First (EDF) – на наборе периодических задач. Реализовать симулятор однопроцессорного планировщика на языке Python, выполнить schedulability-анализ, провести моделирование на интервале до наименьшего общего кратного периодов, подсчитать число пропущенных дедлайнов и сделать выводы об условиях применимости каждого алгоритма.

Теоретическое обоснование

1. Модель задачи в системе жёсткого реального времени. Периодическая задача τ_i описывается тройкой (C_i, T_i, D_i) , где C_i – наихудшее время выполнения (WCET), T_i – период активации, D_i – относительный дедлайн. В классической постановке принимается $D_i = T_i$. Активация k -го экземпляра задачи τ_i происходит в момент $k \cdot T_i$, дедлайн наступает в момент $k \cdot T_i + D_i$. Считается, что задачи независимы, вытесняемы в любой момент времени и не используют общих ресурсов.

Коэффициент загрузки процессора набора из n задач определяется как сумма отношений времени выполнения к периоду:

$$U = C_1/T_1 + C_2/T_2 + \dots + C_n/T_n \quad (1)$$

Величина U является ключевой характеристикой нагрузки: при $U > 1$ набор не может быть выполнен ни одним планировщиком на одном ядре; при $U \leq 1$ вопрос планируемости зависит от выбранного алгоритма.

2. Алгоритм Rate Monotonic (RMA). RMA относится к классу алгоритмов со статическими приоритетами: каждой задаче назначается приоритет один раз и навсегда. Правило RMA – чем короче период задачи, тем выше её приоритет. На каждом шаге планировщик выбирает на исполнение самую приоритетную из готовых задач. Если в момент t активируется задача с более высоким приоритетом, она вытесняет текущую.

Достаточным (но не необходимым) условием планируемости n задач под RMA является оценка Лиу – Лейленда:

$$U \leq n \cdot (2^{1/n} - 1) \quad (2)$$

Для $n = 2$ граница составляет ≈ 0.828 , для $n = 3 - \approx 0.780$, при $n \rightarrow \infty$ величина стремится к $\ln 2 \approx 0.693$. Если U превышает эту границу, но не превышает 1, набор может быть как планируемым, так и нет – требуется точный анализ времени отклика (Response Time Analysis, RTA).

3. Алгоритм Earliest Deadline First (EDF). EDF относится к классу алгоритмов с динамическими приоритетами: на каждом шаге планировщик отдаёт процессор той готовой задаче, у экземпляра которой ближайший абсолютный дедлайн. Приоритет пересчитывается при каждой активации.

Для набора независимых периодических задач с $D_i = T_i$ алгоритм EDF является оптимальным на одном процессоре: достаточным и необходимым условием планируемости является

$$U \leq 1 \quad (3)$$

Это означает, что EDF способен планировать набор задач при загрузке вплоть до 100 % процессорного времени. Цена – большой оверхед на пересчёт приоритетов и более сложный анализ в присутствии общих ресурсов и накладных расходов на переключение контекста.

4. Сравнительная таблица алгоритмов приведена в таблице 1.

Таблица 1 – Сравнение алгоритмов планирования RMA и EDF

Критерий	RMA	EDF
Тип приоритетов	Статические	Динамические (по дедлайну)
Условие планируемости	$U \leq n \cdot (2^{1/n} - 1)$ – достаточное	$U \leq 1$ – точное
Оверхед планировщика	Низкий: приоритеты заданы заранее	Выше: пересчёт при каждой активации
Поведение при	Низкоприоритетные	Каскадный отказ:

Критерий	RMA	EDF
перегрузке	задачи стабильно теряют дедлайны	пропускаются дедлайны у многих задач
Поддержка в OCPB	FreeRTOS, Zephyr, RT-Linux (стандарт)	SCHED_DEADLINE в Linux, эксперимент. в FreeRTOS
Анализ инверсии приоритетов	Хорошо изучен (PIP, PCP)	Сложнее – требует расширений (SRP, EDF-DM)

5. Понятие гиперпериода и схема моделирования. Для анализа поведения планировщика на ограниченном промежутке достаточно рассмотреть интервал длиной в гиперпериод H , равный наименьшему общему кратному всех периодов: $H = \text{НОК}(T_1, T_2, \dots, T_n)$. За этот интервал каждая задача активируется H/T_i раз, а расписание далее повторяется. Если на интервале $[0, H]$ не было пропущенных дедлайнов, расписание корректно бесконечно.

Симуляция дискретного времени строится с шагом $\Delta t = 1$ (приведённое к целым числам время). На каждом шаге: (а) обновляется список готовых экземпляров; (б) выбирается задача с наивысшим приоритетом по выбранному правилу; (в) у неё уменьшается оставшееся время выполнения; (г) при истечении дедлайна экземпляра, который ещё не завершён, фиксируется пропуск.

Пример выполнения задания

Рассмотрим набор из трёх задач, заданный таблицей 2. Все значения приведены в условных единицах времени; принято $D_i = T_i$.

Таблица 2 – Параметры эталонного набора задач для разбора

Задача	C_i	$T_i = D_i$
τ_1	1	4
τ_2	2	6

Задача	C_i	$T_i = D_i$
τ_3	3	12

Шаг 1. Расчёт загрузки и проверка условий. Подставив значения в формулу (1), получим:

$$U = 1/4 + 2/6 + 3/12 = 0.25 + 0.333 + 0.25 = 0.833$$

Граница Лиу – Лейленда для $n = 3$ равна $3 \cdot (2^{1/3} - 1) \approx 0.780$. Полученное значение $U \approx 0.833$ превышает её, поэтому достаточного условия RMA не выполнено и требуется точный анализ либо прогон симулятора. Для EDF выполнено условие (3): $U \leq 1$, значит набор планируется.

Шаг 2. Гиперпериод. Вычисляем $H = \text{НОК}(4, 6, 12) = 12$. На интервале $[0, 12)$ задача τ_1 активируется 3 раза, τ_2 – 2 раза, τ_3 – 1 раз.

Шаг 3. Реализация симулятора. Симулятор представлен в листинге 1. Конструктор принимает список задач и имя политики; метод run выполняет пошаговое моделирование и возвращает количество пропущенных дедлайнов вместе с журналом исполнения.

Листинг 1 – Симулятор однопроцессорного планировщика с политиками RMA и EDF

```

from dataclasses import dataclass
from math import gcd
from functools import reduce

@dataclass
class Task:
    name: str
    c: int # WCET
    t: int # period == deadline

@dataclass
class Job:
    task: Task
    release: int
    remaining: int
    abs_deadline: int

def lcm(values):
    return reduce(lambda a, b: a * b // gcd(a, b), values)

```

```

class Scheduler:
    def __init__(self, tasks, policy):
        self.tasks = tasks
        self.policy = policy # 'RMA' | 'EDF'
        self.h = lcm([t.t for t in tasks])

    def _priority(self, job, now):
        if self.policy == 'RMA':
            return job.task.t # меньше период - выше приоритет
        if self.policy == 'EDF':
            return job.abs_deadline # ближе дедлайн - выше приоритет
        raise ValueError(self.policy)

    def run(self):
        active, log, misses = [], [], 0
        for now in range(self.h):
            for task in self.tasks:
                if now % task.t == 0:
                    active.append(Job(task, now, task.c,
                                     now + task.t))
            active = [j for j in active if j.remaining > 0]
            ready = [j for j in active if j.release <= now]
            if ready:
                ready.sort(key=lambda j: self._priority(j, now))
                chosen = ready[0]
                chosen.remaining -= 1
                log.append((now, chosen.task.name))
            else:
                log.append((now, 'idle'))
            for j in active:
                if j.abs_deadline == now + 1 and j.remaining > 0:
                    misses += 1
                    j.remaining = 0
        return misses, log

tasks = [Task('t1', 1, 4), Task('t2', 2, 6), Task('t3', 3, 12)]
for policy in ('RMA', 'EDF'):
    m, log = Scheduler(tasks, policy).run()
    print(policy, 'misses=', m,
          'schedule:', ''.join(name[-1] if name != 'idle' else ' '
                               for _, name in log))

```

Шаг 4. Ожидаемый вывод программы приведён в листинге 2.

Листинг 2 – Ожидаемый вывод симулятора на эталонном наборе задач

RMA misses= 0 schedule: 12313122.131.
EDF misses= 0 schedule: 12313122.131.

Оба алгоритма успешно укладываются в дедлайны. В лабораторных вариантах необходимо подобрать такие наборы задач, в которых поведение RMA и EDF разойдётся, и сравнить их по числу пропущенных дедлайнов, числу переключений контекста и среднему времени отклика.

Задания по вариантам

Номер варианта определяется как $(N - 1) \bmod 15 + 1$, где N – порядковый номер студента в журнале группы. Конкретные числовые параметры задачи берутся из приведённой ниже таблицы. Составить отчет со скриншотами, показывающие выполнение заданий.

Вариант 1.

13. Реализовать симулятор RMA и EDF для набора задач (C, T) : (1, 5), (2, 8), (2, 10). Принять $D = T$. Прогнать симуляцию на интервале $[0, H]$, где H – гиперпериод периодов.
14. Построить диаграмму Ганта расписания для каждого алгоритма; выделить моменты вытеснения и завершения экземпляров.
15. Подсчитать число пропущенных дедлайнов и среднее время отклика по каждой задаче для RMA и EDF.
16. Изменить C_1 с 1 до 3 с шагом 1; построить график зависимости числа пропусков от C_1 для обоих алгоритмов на одной координатной плоскости.
17. Объяснить, при каком C_1 оценка Лиу – Лейленда перестаёт быть достаточной для RMA, и подтвердить это симуляцией.
18. Подготовить отчёт: исходный код, расчёт U , диаграммы Ганта, числовые результаты в виде таблицы, выводы об условиях применимости каждого алгоритма к данному набору.

Вариант 2.

13. Реализовать симулятор RMA и EDF для набора задач (C, T) : (1, 3), (2, 7), (3, 14). Принять $D = T$. Прогнать симуляцию на интервале $[0, H]$, где H – гиперпериод периодов.

14. Построить диаграмму Ганта расписания для каждого алгоритма; выделить моменты вытеснения и завершения экземпляров.
15. Вычислить границу планируемости Лиу – Лейленда для $n = 3$ и сравнить её с фактической загрузкой U .
16. Подтвердить симуляцией, что при $U \leq 1$, но $U > U_{LL}$, RMA всё ещё планирует набор; найти такое $C_2 \in \{2, 3, 4, 5\}$, при котором RMA впервые срывает дедлайн.
17. Применить точный анализ времени отклика (RTA) к τ_3 и сравнить результат R_3 с фактическим временем отклика в симуляции.
18. Подготовить отчёт: исходный код, расчёт U , диаграммы Ганта, числовые результаты в виде таблицы, выводы об условиях применимости каждого алгоритма к данному набору.

Вариант 3.

13. Реализовать симулятор RMA и EDF для набора задач (C, T) : (2, 6), (3, 9), (4, 12). Принять $D = T$. Прогнать симуляцию на интервале $[0, H]$, где H – гиперпериод периодов.
14. Построить диаграмму Ганта расписания для каждого алгоритма; выделить моменты вытеснения и завершения экземпляров.
15. Подсчитать число пропусков для каждой задачи отдельно (а не суммарно) и показать, какие задачи страдают первыми под RMA.
16. Снять зависимость общего числа пропусков от увеличения C_3 на 10, 20, 30 % и построить график.
17. Заменить периоды на (5, 9, 12) – проверить, как уход от гармоничности периодов меняет результат.
18. Подготовить отчёт: исходный код, расчёт U , диаграммы Ганта, числовые результаты в виде таблицы, выводы об условиях применимости каждого алгоритма к данному набору.

Вариант 4.

13. Реализовать симулятор RMA и EDF для набора задач (C, T): (1, 4), (3, 8), (5, 16). Принять $D = T$. Прогнать симуляцию на интервале $[0, H]$, где H – гиперпериод периодов.
14. Построить диаграмму Ганта расписания для каждого алгоритма; выделить моменты вытеснения и завершения экземпляров.
15. Реализовать политику Round-Robin (квант = 1) как третий алгоритм для сравнения с RMA и EDF.
16. Снять зависимость числа пропусков от увеличения C_3 на $\pm 20\%$ (шаг 1) для всех трёх алгоритмов.
17. Построить сводную таблицу: алгоритм $\times C_3 \times$ число пропусков; сделать вывод о применимости RR к PB-системам.
18. Подготовить отчёт: исходный код, расчёт U , диаграммы Ганта, числовые результаты в виде таблицы, выводы об условиях применимости каждого алгоритма к данному набору.

Вариант 5.

13. Реализовать симулятор RMA и EDF для набора задач (C, T): (1, 2), (1, 5), (2, 10). Принять $D = T$. Прогнать симуляцию на интервале $[0, H]$, где H – гиперпериод периодов.
14. Построить диаграмму Ганта расписания для каждого алгоритма; выделить моменты вытеснения и завершения экземпляров.
15. Подсчитать число переключений контекста за гиперпериод для RMA и EDF; сравнить с теоретической нижней оценкой $H/\min(T_i)$.
16. Внести оверхед переключения контекста $\varepsilon = 0.1$ у. е. времени в симулятор; пересчитать пропуски с учётом $\varepsilon \in \{0, 0.1, 0.2, 0.3\}$.
17. Показать, при каком ε EDF начинает терять дедлайны раньше RMA.
18. Подготовить отчёт: исходный код, расчёт U , диаграммы Ганта, числовые результаты в виде таблицы, выводы об условиях применимости каждого алгоритма к данному набору.

Вариант 6.

13. Реализовать симулятор RMA и EDF для набора задач (C, T): (3, 10), (4, 15), (5, 20). Принять $D = T$. Прогнать симуляцию на интервале $[0, H]$, где H – гиперпериод периодов.
14. Построить диаграмму Ганта расписания для каждого алгоритма; выделить моменты вытеснения и завершения экземпляров.
15. Построить распределение моментов завершения каждого экземпляра по обоим алгоритмам (гистограмма).
16. Вычислить максимальное и среднее время отклика для каждой задачи; сравнить с теоретическим R_i по RTA.
17. Добавить четвёртую задачу (C_i, T_i) на свой выбор так, чтобы RMA сорвал, а EDF успел; обосновать выбор.
18. Подготовить отчёт: исходный код, расчёт U , диаграммы Ганта, числовые результаты в виде таблицы, выводы об условиях применимости каждого алгоритма к данному набору.

Вариант 7.

13. Реализовать симулятор RMA и EDF для набора задач (C, T): (2, 5), (2, 7), (3, 11). Принять $D = T$. Прогнать симуляцию на интервале $[0, H]$, где H – гиперпериод периодов.
14. Построить диаграмму Ганта расписания для каждого алгоритма; выделить моменты вытеснения и завершения экземпляров.
15. Найти гиперпериод H аналитически и подтвердить расчётом из симулятора.
16. Прогнать симуляцию на интервалах $H, 2H, 5H$ – подтвердить, что расписание периодически с периодом H .
17. Оценить, во сколько раз вырастет время моделирования при увеличении одного из периодов на 1 (нарушение взаимной простоты не должно сильно влиять).

18. Подготовить отчёт: исходный код, расчёт U , диаграммы Ганта, числовые результаты в виде таблицы, выводы об условиях применимости каждого алгоритма к данному набору.

Вариант 8.

13. Реализовать симулятор RMA и EDF для набора задач (C, T) : (1, 4), (2, 6), (4, 8). Принять $D = T$. Прогнать симуляцию на интервале $[0, H]$, где H – гиперпериод периодов.

14. Построить диаграмму Ганта расписания для каждого алгоритма; выделить моменты вытеснения и завершения экземпляров.

15. Вычислить U аналитически; убедиться, что $U = 1.0$ и для RMA, и для EDF граница планируемости достигнута.

16. Показать симуляцией, что RMA срывает дедлайны на τ_3 , а EDF – нет; указать конкретный момент срыва.

17. Уменьшить C_3 на 0.1 шагом 0.05 до тех пор, пока RMA не перестанет срывать дедлайны; найти граничное C_3 .

18. Подготовить отчёт: исходный код, расчёт U , диаграммы Ганта, числовые результаты в виде таблицы, выводы об условиях применимости каждого алгоритма к данному набору.

Вариант 9.

13. Реализовать симулятор RMA и EDF для набора задач (C, T) : (1, 3), (1, 4), (2, 6), (3, 12). Принять $D = T$. Прогнать симуляцию на интервале $[0, H]$, где H – гиперпериод периодов.

14. Построить диаграмму Ганта расписания для каждого алгоритма; выделить моменты вытеснения и завершения экземпляров.

15. Построить диаграмму Ганта на интервале $[0, 24]$ (двойной гиперпериод).

16. Сравнить среднее время отклика по τ_4 для RMA и EDF; объяснить, почему длинная низкоприоритетная задача страдает сильнее под RMA.

17. Заменить порядок приоритетов RMA на обратный (Deadline-Monotonic с $D = 2T$ для τ_1 и $D = T$ для остальных) – пересчитать пропуски.

18. Подготовить отчёт: исходный код, расчёт U , диаграммы Ганта, числовые результаты в виде таблицы, выводы об условиях применимости каждого алгоритма к данному набору.

Вариант 10.

13. Реализовать симулятор RMA и EDF для набора задач (C, T) : $(2, 8)$, $(3, 12)$, $(5, 20)$, $(1, 6)$. Принять $D = T$. Прогнать симуляцию на интервале $[0, H]$, где H – гиперпериод периодов.

14. Построить диаграмму Ганта расписания для каждого алгоритма; выделить моменты вытеснения и завершения экземпляров.

15. Сравнить среднее, максимальное и 95-перцентильное время отклика по каждой задаче для двух алгоритмов.

16. Добавить в симулятор поддержку приоритетов наследования (PIP) для модельной критической секции у τ_2 длительностью 1; оценить изменение R_1 .

17. Объяснить, в каких случаях PIP не помогает (контрпример с тремя задачами).

18. Подготовить отчёт: исходный код, расчёт U , диаграммы Ганта, числовые результаты в виде таблицы, выводы об условиях применимости каждого алгоритма к данному набору.

Вариант 11.

13. Реализовать симулятор RMA и EDF для набора задач (C, T) : $(1, 5)$, $(3, 10)$, $(4, 20)$, $(2, 25)$. Принять $D = T$. Прогнать симуляцию на интервале $[0, H]$, где H – гиперпериод периодов.

14. Построить диаграмму Ганта расписания для каждого алгоритма; выделить моменты вытеснения и завершения экземпляров.

15. Построить распределение пропусков дедлайнов по экземплярам (сколько раз пропустила первая, вторая, третья активация каждой задачи).

16. Снять зависимость числа пропусков от длительности интервала моделирования (от $1 \cdot H$ до $10 \cdot H$) – убедиться, что после первого гиперпериода картина стабилизируется.

17. Преобразовать набор: умножить все C_i на 1.1; пересчитать U и пропуски, подтвердить ожидаемый рост.
18. Подготовить отчёт: исходный код, расчёт U , диаграммы Ганта, числовые результаты в виде таблицы, выводы об условиях применимости каждого алгоритма к данному набору.

Вариант 12.

13. Реализовать симулятор RMA и EDF для набора задач (C, T) : (2, 4), (3, 8), (5, 16). Принять $D = T$. Прогнать симуляцию на интервале $[0, H]$, где H – гиперпериод периодов.
14. Построить диаграмму Ганта расписания для каждого алгоритма; выделить моменты вытеснения и завершения экземпляров.
15. Показать конкретный момент пропуска дедлайна под RMA; восстановить цепочку вытеснений, которая привела к пропуску.
16. Применить точный RTA: рассчитать R_i для всех задач, сопоставить с симуляцией.
17. Преобразовать набор так, чтобы периоды были гармоничны (каждый делится на предыдущий нацело) – показать, что RMA становится планируемым при $U > U_{LL}$.
18. Подготовить отчёт: исходный код, расчёт U , диаграммы Ганта, числовые результаты в виде таблицы, выводы об условиях применимости каждого алгоритма к данному набору.

Вариант 13.

13. Реализовать симулятор RMA и EDF для набора задач (C, T) : (1, 3), (2, 5), (1, 7), (2, 9). Принять $D = T$. Прогнать симуляцию на интервале $[0, H]$, где H – гиперпериод периодов.
14. Построить диаграмму Ганта расписания для каждого алгоритма; выделить моменты вытеснения и завершения экземпляров.
15. Подсчитать U ; подобрать дополнительную пятую задачу (C_5, T_5) так, чтобы общая U выросла до 0.95; пересчитать пропуски.

16. Оценить отношение число пропусков RMA / EDF на полученном наборе.
17. Подтвердить аналитически: для EDF при $U \leq 1$ пропусков быть не должно; объяснить расхождение, если оно возникнет в симуляции (например, из-за округлений).
18. Подготовить отчёт: исходный код, расчёт U , диаграммы Ганта, числовые результаты в виде таблицы, выводы об условиях применимости каждого алгоритма к данному набору.

Вариант 14.

13. Реализовать симулятор RMA и EDF для набора задач (C, T) : (3, 6), (5, 12), (6, 24). Принять $D = T$. Прогнать симуляцию на интервале $[0, H]$, где H – гиперпериод периодов.
14. Построить диаграмму Ганта расписания для каждого алгоритма; выделить моменты вытеснения и завершения экземпляров.
15. Подсчитать среднее и максимальное время отклика τ_1 ; показать, как длинная τ_3 вытесняет (или не вытесняет) короткую τ_1 .
16. Добавить четвёртую короткую задачу (1, 3) с высшим приоритетом; проверить устойчивость планирования.
17. Сравнить с EDF: совпадают ли моменты вытеснений и завершений?
18. Подготовить отчёт: исходный код, расчёт U , диаграммы Ганта, числовые результаты в виде таблицы, выводы об условиях применимости каждого алгоритма к данному набору.

Вариант 15.

13. Реализовать симулятор RMA и EDF для набора задач (C, T) : (1, 4), (2, 6), (3, 8), (5, 16). Принять $D = T$. Прогнать симуляцию на интервале $[0, H]$, где H – гиперпериод периодов.
14. Построить диаграмму Ганта расписания для каждого алгоритма; выделить моменты вытеснения и завершения экземпляров.
15. Подобрать минимальное $\Delta \in \{0.1, 0.2, \dots\}$, при котором замена C_4 на $C_4 + \Delta$ приводит к первому пропуску дедлайна под RMA.

16. Для найденного Δ подтвердить, что EDF продолжает успевать.
17. Построить график « $\Delta \rightarrow$ число пропусков» для RMA в диапазоне $\Delta \in [0, 2]$; найти граничные значения, где число пропусков скачком увеличивается.
18. Подготовить отчёт: исходный код, расчёт U , диаграммы Ганта, числовые результаты в виде таблицы, выводы об условиях применимости каждого алгоритма к данному набору.

Лабораторная работа 5. Анализ CAN-трафика с помощью PCAN-View / Wireshark + CAN-USB

Цель работы: освоить методику захвата и анализа кадров протокола CAN с помощью адаптера CAN-USB и инструментов PCAN-View / Wireshark 4.x (плагин SocketCAN). Подключить два узла к общей шине, разобрать структуру кадра данных (SOF, идентификатор, RTR, IDE, DLC, поле данных, CRC, ACK, EOF), пронаблюдать механизм неразрушающего арбитража по доминантному биту, записать и воспроизвести фрагмент шинного трафика, оценить загрузку шины при заданной битрейт-настройке. Дополнительно сопоставить классический CAN 2.0 с расширениями CAN FD (ISO 11898-1:2015, до 8 Мбит/с) и CAN XL (CiA 610-1, до 20 Мбит/с), которые применяются в современных автомобильных и промышленных сетях.

Теоретическое обоснование

1. Физический уровень и стек CAN. CAN (Controller Area Network) – последовательный полудуплексный протокол с дифференциальной передачей по двухпроводной шине (CAN_H и CAN_L), стандартизованный ISO 11898. На физическом уровне различают два состояния линии: доминантное (логический 0, $V_{diff} \approx 2$ В) и рецессивное (логическая 1, $V_{diff} \approx 0$ В). Каждый узел подключён к шине через приёмопередатчик (CAN-трансивер), который преобразует логические уровни контроллера в напряжения линии и обратно. На концах шины обязательны согласующие резисторы 120 Ом.

Стандартные битрейты – 125, 250, 500 кбит/с и 1 Мбит/с. Длина шины и битрейт связаны обратно: при 1 Мбит/с длина не должна превышать ≈ 40 м, при 125 кбит/с допускается до 500 м. Время бита t_{bit} разбивается контроллером на 4 сегмента (SYNC, PROP, PHASE1, PHASE2); точка выборки обычно располагается на отметке 75 – 87.5 % длительности бита.

2. Структура стандартного кадра данных CAN 2.0A. Кадр данных – основная единица передачи. Графически его состав показан на рисунке 1; поля идут в следующем порядке.

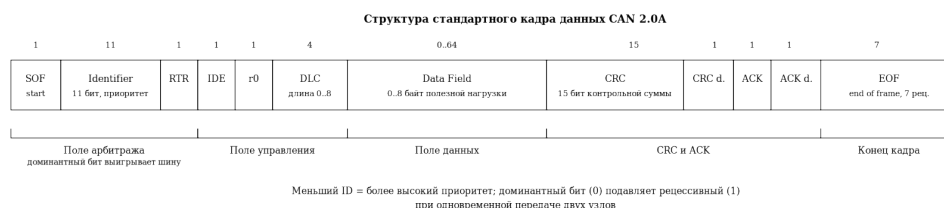


Рисунок 1 – Структура стандартного кадра данных CAN 2.0A

19. SOF (Start Of Frame) – 1 доминантный бит, синхронизация всех узлов на фронте.
20. Identifier – 11 бит идентификатора сообщения; одновременно выполняет роль приоритета (меньшее значение – выше приоритет).
21. RTR (Remote Transmission Request) – 1 бит; 0 для кадра данных, 1 для кадра запроса данных.
22. IDE – 1 бит, признак расширенного идентификатора (для CAN 2.0B).
23. r0 – 1 зарезервированный доминантный бит.
24. DLC (Data Length Code) – 4 бита, длина поля данных в байтах (0...8).
25. Data Field – от 0 до 8 байт полезной нагрузки.
26. CRC – 15 бит контрольной суммы по полиному $x^{15} + x^{14} + x^{10} + x^8 + x^7 + x^4 + x^3 + 1$, далее разделитель CRC (1 рецессивный бит).
27. ACK Slot (1 бит) + ACK Delimiter (1 рецессивный бит) – место, куда любой принявший узел вставляет доминантный бит подтверждения целостности.
28. EOF (End Of Frame) – 7 рецессивных бит, конец кадра; затем межкадровый интервал (IFS) не менее 3 рецессивных бит.

При штатной передаче применяется также bit stuffing: после 5 одинаковых бит подряд передатчик вставляет один бит противоположной полярности, который удаляется приёмником. Стаффинг защищает от потери синхронизации, но увеличивает реальную длину кадра до 30 – 40 % от номинала.

3. Битрейт, время бита и загрузка шины. Длительность одного бита обратно пропорциональна битрейту:

$$t_{bit} = 1 / R_{bit} \quad (1)$$

Максимальная длина стандартного кадра данных с 8 байтами полезной нагрузки – около 132 бит с учётом стаффинга. Время передачи такого кадра при $R_{bit} = 500$ кбит/с составляет $132 / 500\,000 \approx 264$ мкс. Загрузка шины при N кадрах в секунду оценивается как:

$$B = N \cdot L_{\text{кадр}} / R_{bit} \cdot 100 \% \quad (2)$$

где $L_{\text{кадр}}$ – средняя длина кадра в битах. На практике рабочую загрузку шины ограничивают значением 30 – 50 %, чтобы оставить запас на пиковые всплески и обеспечить детерминированность задержек у высокоприоритетных сообщений.

4. Арбитраж по доминантному биту. CAN использует механизм CSMA/CD-AMP (Carrier Sense Multiple Access / Collision Detection with Arbitration on Message Priority). Если два узла начинают передачу одновременно (после периода тишины IFS), они побитно сравнивают передаваемый идентификатор с уровнем шины. При несовпадении узел, передавший рецессивный бит, обнаруживает на шине доминантный, проигрывает арбитраж и немедленно прекращает передачу, переходя в режим приёма. Узел с меньшим ID (более доминантным) продолжает передачу без потери времени и без повторной отправки. Арбитраж неразрушающий: ни один бит уже переданного префикса не теряется.

5. Сравнение типичных инструментов анализа CAN-шины приведено в таблице 1.

Таблица 1 – Инструменты анализа CAN-трафика

Инструмент	Платформа	Особенности
PCAN-View	Windows, адаптер PEAK PCAN-USB	Простой GUI, передача и приём, запись в .trc, статистика загрузки шины
candump / cansend (can-utils)	Linux + SocketCAN	CLI, дампы в текст или .log, фильтры по ID, поддержка виртуального интерфейса vcan0
Wireshark +	Linux	Полноценный анализатор с

Инструмент	Платформа	Особенности
SocketCAN		фильтрами и диссектором CAN, сохранение в .pcapng
BUSMASTER	Windows, Vector/Kvaser/PEAK	Открытый аналог CANalyzer, скрипты на Python для генерации трафика

6. Виртуальная шина SocketCAN. Для отладки без физического адаптера в Linux используется виртуальный интерфейс vcan: после загрузки модуля vcan и поднятия интерфейса (`ip link add dev vcan0 type vcan; ip link set up vcan0`) все стандартные утилиты can-utils и Wireshark 4.x работают с ним так же, как с реальной шиной. Это позволяет отрабатывать сценарии и форматы кадров до подключения железа. Wireshark начиная с версии 3.6 и далее в 4.x поддерживает CAN-FD «из коробки» – через интерфейс vcan можно увидеть в дампе и классические, и FD-кадры.

7. Расширения CAN: CAN FD и CAN XL. Помимо классического CAN 2.0 на 1 Мбит/с в актуальных стандартах закреплены две высокоскоростные модификации, с которыми приходится сталкиваться в современных автомобильных и промышленных сетях:

19. CAN FD (Flexible Data-rate, ISO 11898-1:2015) – расширение, при котором поле данных и CRC передаются на повышенной скорости до 8 Мбит/с (типичные значения 2 и 5 Мбит/с), длина поля данных – до 64 байт. Сохранена обратная совместимость с CAN 2.0 на этапе арбитража. В Linux поддерживается socketCAN при наличии FD-трансивера и драйвера контроллера; для виртуального интерфейса используется параметр fd on: `ip link set vcan0 mtu 72, ip link set vcan0 type vcan; cansend` поддерживает FD через ключ -fd.

20. CAN XL (CiA 610-1, релиз 2022 – 2024) – самая новая спецификация, скорость поля данных до 20 Мбит/с, длина – до 2048 байт. Ориентирована на замену классической CAN-шины в автомобильных bus-системах и интеграцию с Ethernet через 10BASE-T1S. На рынке доступны первые

контроллеры (NXP, Bosch, Microchip); в учебной лаборатории чаще используется как тема обзора, без аппаратной реализации.

21. Обычный CAN 2.0 остаётся базовым уровнем: классический трансивер ISO 11898-2 и адаптеры PCAN-USB / CANable 2.0 работают на 1 Мбит/с с длиной кадра до 8 байт – именно с ним выполняется основная часть лабораторной.

Пример выполнения задания

Рассмотрим разбор «нулевого» варианта: битрейт 500 кбит/с, два виртуальных узла на интерфейсе vcan0, первый узел периодически отправляет ID = 0x100 с полезной нагрузкой счётчика, второй – ID = 0x101 с константной нагрузкой. Требуется снять дамп, определить загрузку шины и пронаблюдать арбитраж.

Шаг 1. Поднятие виртуального CAN-интерфейса. Действия выполняются под пользователем с правами sudo. Команды листинга 1 создают виртуальный интерфейс vcan0.

Листинг 1 – Поднятие виртуального CAN-интерфейса под Linux

```
sudo modprobe vcan  
sudo ip link add dev vcan0 type vcan  
sudo ip link set up vcan0  
ip -details link show vcan0
```

Шаг 2. Запуск двух периодических передатчиков. Скрипт листинга 2 на python-can отправляет с двух разных идентификаторов кадры с разной частотой. Запускается в двух терминалах с разными параметрами.

Листинг 2 – Эмуляция двух CAN-узлов на python-can

```
import argparse, time, can  
  
ap = argparse.ArgumentParser()  
ap.add_argument('--id', type=lambda s: int(s, 0), required=True)  
ap.add_argument('--period', type=float, default=0.010)  
ap.add_argument('--count', type=int, default=1000)  
args = ap.parse_args()  
  
bus = can.interface.Bus(channel='vcan0', bustype='socketcan')
```

```

for i in range(args.count):
    payload = [(i >> (8 * k)) & 0xFF for k in range(8)]
    msg = can.Message(arbitration_id=args.id, data=payload,
                      is_extended_id=False)
    bus.send(msg)
    time.sleep(args.period)

```

Запуск двух узлов в отдельных терминалах:

Листинг 3 – Параллельный запуск двух эмулируемых узлов

```

# терминал 1 - узел с высоким приоритетом, 100 Гц
python3 node.py --id 0x100 --period 0.010

# терминал 2 - узел с низким приоритетом, 50 Гц
python3 node.py --id 0x101 --period 0.020

```

Шаг 3. Захват трафика и сохранение дампа. В третьем терминале запускается `candump` с записью в файл и одновременно Wireshark – оба видят один и тот же поток кадров.

Листинг 4 – Захват CAN-трафика средствами `can-utils` и Wireshark

```

# текстовый дамп в файл, фильтр по ID
candump -L vcan0 > capture.log &

# параллельный анализ в Wireshark
sudo wireshark -k -i vcan0

```

Шаг 4. Разбор формата кадра по дампу. В формате `candump` одна строка соответствует одному кадру и содержит метку времени, имя интерфейса, ID, DLC и байты данных. Пример вывода – листинг 5.

Листинг 5 – Фрагмент дампа `candump -L vcan0`

```

(1716895123.123456) vcan0 100#0000000000000000
(1716895123.133612) vcan0 100#0100000000000000
(1716895123.143810) vcan0 101#DEADBEEFCAFEBAFE
(1716895123.153790) vcan0 100#0200000000000000
(1716895123.163955) vcan0 101#DEADBEEFCAFEBAFE
(1716895123.173812) vcan0 100#0300000000000000

```

Здесь 100 и 101 – идентификаторы в шестнадцатеричном виде, после знака # идут восемь байт данных. DLC определяется по числу байт. Скорость

кадров с ID 0x100 равна 100 Гц, с ID 0x101 – 50 Гц, что соответствует заданным периодам.

Шаг 5. Расчёт загрузки шины. При битрейте 500 кбит/с средняя длина кадра с 8 байтами данных и стаффингом составляет около 130 бит. Общее число кадров в секунду – 150 (100 + 50). По формуле (2):

$$B = 150 \cdot 130 / 500\,000 \cdot 100 \% = 3.9 \%$$

Загрузка $\approx 3.9 \%$, что подтверждает прогноз: на 500 кбит/с теоретически помещается до ≈ 3800 кадров/с.

Шаг 6. Наблюдение арбитража. При одновременной отправке двух кадров с ID 0x100 и 0x101 захват покажет, что первым на шине появится кадр с ID 0x100, второй пройдёт сразу после него (без потерь, без повторов). На реальном адаптере это видно по полю «Bus load» в PCAN-View; на vcan – по метке времени: между двумя кадрами интервал ≈ 270 мкс, равный длительности первого кадра. Это и есть проявление неразрушающего арбитража: кадр с меньшим ID выигрывает шину «бесплатно».

Шаг 7. Воспроизведение записанного трафика. Сохранённый дамп можно проиграть обратно командой canplayer, например для стресс-теста приёмника.

Листинг 6 – Воспроизведение записанного дампа

```
canplayer -l capture.log vcan0=vcan0
```

Опция I задаёт входной файл, vcan0=vcan0 – соответствие имени интерфейса в дампе и в текущей системе. Параметр -l N задаёт число повторов проигрывания.

Задания по вариантам

Номер варианта определяется как $(N - 1) \bmod 15 + 1$, где N – порядковый номер студента в журнале группы. Конкретные числовые параметры задачи берутся из приведённой ниже таблицы. Составить отчет со скриншотами, показывающие выполнение заданий.

Вариант 1.

19. Поднять виртуальный интерфейс vcan0 и запустить два узла на python-can с битрейтом 125 кбит/с: ID 0x080 с периодом 10 мс (100 Гц), ID 0x200 с периодом 50 мс (20 Гц).
20. Снять 30-секундный дамп через candump -L и параллельно через Wireshark 4.x; сохранить .log и .pcapng.
21. Рассчитать теоретическую загрузку шины по формуле (2) и сравнить с показаниями PCAN-View / candump-статистики (canbusload).
22. Подготовить отчёт: команды поднятия интерфейса, фрагменты дампов, таблица «ID – частота – расчёт – факт», выводы.

Вариант 2.

19. Поднять vcan0 и три узла с ID 0x050, 0x051, 0x052, период 5 мс, битрейт 250 кбит/с, полезная нагрузка 8 байт.
20. Записать 60-секундный дамп через candump; разобрать его скриптом на python (можно scapy + диссектор) и построить гистограмму числа кадров по каждому ID.
21. Снять статистику загрузки шины через canbusload vcan0@250000 и сравнить с теоретической.
22. В отчёт включить скрипт-парсер, таблицу «ID – кадров/с – процент загрузки» и вывод о приоритете доставки.

Вариант 3.

19. Битрейт 500 кбит/с, два узла: ID 0x010 и ID 0x011 стартуют синхронно с одинаковым периодом 1 мс (запуск из одного управляющего скрипта).

20. Снять 5-секундный дамп; восстановить по меткам времени очередность кадров и зафиксировать межкадровый интервал.
21. Объяснить полученный порядок через механизм неразрушающего арбитража и подсчитать «штраф» по времени, который получает ID 0x011.
22. В отчёте привести фрагмент дампа, расчёт интервала и обоснование, почему ID 0x010 выигрывает шину.

Вариант 4.

19. Битрейт 1 Мбит/с. Подобрать комбинацию периодов для пяти узлов (ID 0x100...0x104) так, чтобы фактическая загрузка шины составила $70 \pm 2 \%$.
20. Подтвердить расчёт прогоном на `vsan0` в течение 60 с и снятием статистики `canbusload`.
21. Подсчитать число и максимальную задержку доставки кадра с ID 0x104 (низший приоритет); построить гистограмму задержек.
22. В отчёт включить таблицу теоретической и фактической загрузки, гистограмму задержки ID 0x104, вывод об устойчивости конфигурации.

Вариант 5.

19. Реализовать на python-can обмен RTR-кадрами: узел-инициатор шлёт remote frame с ID 0x123 и DLC = 8 раз в 100 мс, узел-ответчик возвращает кадр данных с тем же ID и 8 байтами полезной нагрузки.
20. Снять дамп через `sandump -L` и проверить, что в дампе присутствуют оба типа кадров (R-флаг у RTR).
21. Сравнить длительность RTR-кадра и кадра данных по меткам времени; объяснить разницу через структуру кадра.
22. В отчёт включить код обоих узлов, фрагмент дампа с выделенными RTR-кадрами и расчёт длительностей.

Вариант 6.

19. Битрейт 250 кбит/с. Поднять три узла: ID 0x180, ID 0x280 (стандартный 11-битный формат) и ID 0x18FF0000 (расширенный 29-битный, признак IDE = 1).
20. Снять дамп Wireshark 4.x с включённым CAN-диссектором; выделить в дампе кадры с IDE = 0 и IDE = 1.
21. Подсчитать длины кадров (в битах) для обоих форматов и сравнить с теоретическими значениями (44 и 64 бит соответственно без учёта стаффинга).
22. В отчёт включить скриншот фильтра Wireshark по IDE, таблицу длин кадров и вывод о цене перехода на расширенный формат.

Вариант 7.

19. Битрейт 500 кбит/с. Поднять периодический поток ID 0x100 с периодом 1 мс и аperiodический поток ID 0x050 (вручную или по cangen) с большей нагрузкой данных (8 байт).
20. Прогнать обмен 60 с; по дампу определить максимальную задержку доставки ID 0x100, вызванную блокировкой кадром ID 0x050 (хотя 0x050 имеет более высокий приоритет – именно поэтому он блокирует уже идущий кадр 0x100).
21. Сравнить замеренный максимум с теоретической оценкой худшего случая (длительность одного 8-байтного кадра).
22. В отчёт включить гистограмму времени отклика ID 0x100, теоретический расчёт и комментарий о приоритетной инверсии в CAN.

Вариант 8.

19. Поднять три узла разной частоты (1, 5, 20 мс) на vcan0; записать 30-секундный дамп через sandump -L > capture.log.
20. На второй машине (или второй vcan-интерфейс vcan1 в той же системе) проиграть дамп через canplayer -I capture.log vcan0=vcan1.

21. Параллельно с воспроизведением снять второй дамп через sandump vcan1; сравнить число кадров и распределение по ID с оригиналом.
22. В отчёт включить команды записи и воспроизведения, таблицу сравнения двух дампов по числу кадров на каждый ID, вывод о потерях (если есть).

Вариант 9.

19. Битрейт 500 кбит/с. Передать одиночный кадр с DLC = 0 (без данных) с ID 0x000 (наивысший приоритет) и одиночный кадр DLC = 8 с тем же ID.
20. Снять дамп Wireshark 4.x и измерить длительность каждого кадра по меткам времени или через диссектор.
21. Подтвердить расчётом разницу: $\Delta T = 8 \cdot 8 / 500\,000 = 128$ мкс (плюс возможный стаффинг).
22. В отчёт включить два скриншота-разбора кадра в Wireshark, измеренные и теоретические значения длительностей.

Вариант 10.

19. Битрейт 125 кбит/с. Принять для модели длину шины 200 м (влияет на максимально допустимый битрейт – укладывается в лимит).
20. Подобрать число одинаковых узлов с периодом 20 мс и нагрузкой 8 байт так, чтобы загрузка шины не превышала 30 %.
21. Подтвердить теоретический расчёт прогоном на vcan0 в течение 60 с и снятием canbusload.
22. В отчёт включить расчёт максимального числа узлов, таблицу «N узлов – загрузка, %» и обоснование, почему предел 30 % – разумная инженерная норма.

Вариант 11.

19. Битрейт 500 кбит/с, три узла с периодами 1, 5, 10 мс. Записать 60-секундный дамп одновременно через sandump (.log) и Wireshark 4.x (.pcapng).
20. Преобразовать оба дампа в текст со счётчиком кадров; сравнить число кадров и распределение по ID.

21. Подсчитать число кадров, расхождение которых отличается между двумя инструментами (потери одного из них).

22. В отчёт включить команды записи, парсер с подсчётом, таблицу сравнения и вывод о пригодности обоих инструментов для промышленной диагностики.

Вариант 12.

19. Битрейт 500 кбит/с, узел-передатчик ID 0x100, узел-приёмник слушает шину.

20. Через `cangen -e` (генерация ошибочных кадров) или ручную посылку с битом ошибки спровоцировать ошибку CRC у одного из кадров.

21. Снять дамп `sandump -e` (вывод error frames) и зафиксировать, как принимающий узел инициирует Error Frame и инкрементирует счётчики ошибок (TEC / REC из регистра CAN_ESR).

22. В отчёт включить фрагмент дампа с Error Frame, выводы по значениям TEC / REC и обоснование, в какой момент узел переходит в состояние Error Passive.

Вариант 13.

19. Битрейт 1 Мбит/с. С помощью `cangen vcan0 -g 1 -I i -L i` поднять случайный поток кадров и довести загрузку шины до 90 % (контроль – `canbusload`).

20. Параллельно запустить приёмник, который пишет дамп `sandump` и ведёт счётчик ожидаемых / реально принятых кадров.

21. По истечении 60 с подсчитать число пропущенных кадров и среднюю задержку доставки самых низкоприоритетных ID.

22. В отчёт включить параметры `cangen`, статистику загрузки, число пропусков по ID и вывод о применимости такой загрузки в промышленных системах.

Вариант 14.

1. Битрейт 250 кбит/с. Два узла с ID 0x100 и ID 0x101, период 2 мс, синхронный старт.
2. Снять 10-секундный дамп; восстановить по меткам времени очередность кадров.
3. Проверить и обосновать, что ID 0x100 всегда передаётся первым (почему именно так, исходя из значений идентификаторов).
4. В отчёт включить таблицу «время – ID», расчёт фактического периода появления каждого ID и вывод о детерминированности CAN-арбитража.

Вариант 15.

1. Битрейт 500 кбит/с. Поднять шесть узлов с ID 0x010, 0x020, 0x030, 0x040, 0x050, 0x060, синхронный старт каждые 10 мс (нагрузка 8 байт у каждого).
2. Снять 30-секундный дамп; для каждого периода 10 мс зафиксировать порядок прохода кадров и измерить задержку доставки самого низкоприоритетного (ID 0x060) относительно момента старта.
3. Сравнить с теоретическим худшим случаем (≈ 5 длительностей кадра по $130 \text{ бит} / 500 \text{ кбит/с} \approx 1.3 \text{ мс}$).
4. В отчёт включить гистограмму задержки ID 0x060, теоретический расчёт, вывод об устойчивости расписания при синхронном старте узлов.

Лабораторная работа 6. Отправка и приём CAN-сообщений на STM32 с CAN-контроллером (или симуляция)

Цель работы: освоить настройку контроллера bxCAN (или FDCAN на новых STM32) микроконтроллера STM32 для приёма и отправки сообщений по шине CAN. Сгенерировать каркас проекта в STM32CubeIDE 1.16+ (встроенный CubeMX), инициализировать тактирование, фильтры приёма и прерывания на стороне HAL, реализовать отправку кадра по нажатию пользовательской кнопки и приём сообщений от второго узла, вывести идентификатор и содержимое полученных кадров на терминал по UART. При отсутствии второй платы – использовать программную симуляцию второго узла на python-can через виртуальный интерфейс vcan0 и адаптер CAN-USB (CANable 2.0 / PCAN-USB).

Теоретическое обоснование

1. Архитектура bxCAN и FDCAN в семействах STM32. Контроллер bxCAN (basic extended CAN) – встроенный аппаратный модуль классических STM32 F1, F4, L4. Соответствует спецификации CAN 2.0A и B (стандартный 11-бит и расширенный 29-бит ID), поддерживает битрейт до 1 Мбит/с. Содержит три почтовых ящика передачи (TX mailboxes 0...2) с приоритетной очередью, две FIFO-очереди приёма (FIFO0, FIFO1) по три кадра каждая, и блок из 14 (для bxCAN1) или 28 фильтров приёма. В новых семействах STM32 G0 / G4 / H7 (актуальный мейнстрим 2024 – 2026 годов) bxCAN заменён на FDCAN с поддержкой CAN FD (до 8 Мбит/с в поле данных), 64-байтным payload и дедушевым (configurable) Message RAM объёмом 2 – 10 КБ; HAL-API отличается префиксом HAL_FDCAN_*. Классические STM32 F1 / F4 формально считаются legacy, но широко применяются в учебных платах Nucleo и Discovery, поэтому именно с bxCAN построен пример этой лабораторной.

Для работы на стороне HAL (STM32 HAL Driver) используются структуры `CAN_HandleTypeDef`, `CAN_TxHeaderTypeDef`,

CAN_RxHeaderTypeDef, CAN_FilterTypeDef и функции HAL_CAN_Init, HAL_CAN_ConfigFilter, HAL_CAN_Start, HAL_CAN_ActivateNotification, HAL_CAN_AddTxMessage, HAL_CAN_GetRxMessage.

2. Тактирование и расчёт битрейта. Контроллер CAN тактируется от шины APB1 (PCLK1). Время бита делится на сегменты: SYNC_SEG (1 квант), BS1 (1...16 квантов, включает PROP и PHASE1) и BS2 (1...8 квантов, PHASE2). Битрейт определяется формулой:

$$R_bit = f_PCLK1 / (Prescaler \cdot (1 + BS1 + BS2)) \quad (1)$$

Точка выборки – на границе BS1 / BS2 – обычно ставится на 75 – 87.5 % длительности бита. Пример: $f_PCLK1 = 42$ МГц, $Prescaler = 6$, $BS1 = 11$, $BS2 = 2$; тогда $R_bit = 42\,000\,000 / (6 \cdot 14) = 500\,000$ бит/с, точка выборки $= (1 + 11) / 14 = 85.7 \%$.

3. Фильтры приёма. Каждый аппаратный фильтр представлен парой 32-битных регистров (FR1, FR2) и работает в одном из режимов: Mask или List, по 16 или 32 бит. В режиме Mask регистр FR1 задаёт ожидаемое значение ID, FR2 – маску значащих бит. Любое сообщение, у которого все ненулевые биты маски совпали с ожидаемым ID, попадает в назначенный FIFO. Чтобы пропустить все ID 0x000...0x7FF без фильтрации, выставляют FilterIdHigh = 0, FilterMaskIdHigh = 0 (любой ID совпадает с любым значением).

Таблица 1 – Поля структуры CAN_FilterTypeDef (HAL)

Поле	Назначение
FilterBank	Номер банка фильтра (0...13 для bxCAN1)
FilterMode	CAN_FILTERMODE_IDMASK или CAN_FILTERMODE_IDLIST
FilterScale	CAN_FILTERSCALE_16BIT или 32BIT
FilterIdHigh / FilterIdLow	Ожидаемое значение идентификатора (биты ID + IDE + RTR)
FilterMaskIdHigh / FilterMaskIdLow	Маска значащих битов идентификатора
FilterFIFOAssignment	CAN_FILTER_FIFO0 или

Поле	Назначение
	CAN_FILTER_FIFO1
FilterActivation	ENABLE / DISABLE – активация банка

4. Передача кадра. Передача инициируется функцией HAL_CAN_AddTxMessage(hcan, &tx_header, data, &tx_mailbox). Структура CAN_TxHeaderTypeDef содержит идентификатор (StdId или ExtId), признак расширенного формата (IDE), признак RTR, длину (DLC) и опцию TransmitGlobalTime для аппаратной метки времени. Контроллер сам ставит сообщение в свободный почтовый ящик, выполняет арбитраж на шине, при потере арбитража – автоматическую повторную передачу (если не отключено).

5. Приём и обработка прерываний. После HAL_CAN_Start вызывается HAL_CAN_ActivateNotification с маской CAN_IT_RX_FIFO0_MSG_PENDING; при поступлении кадра HAL вызывает пользовательский callback HAL_CAN_RxFifo0MsgPendingCallback, в котором следует прочитать кадр через HAL_CAN_GetRxMessage. Для отладочного вывода удобно использовать UART2 на USART2 (115200 8N1) – на большинстве плат Nucleo он соединён со встроенным ST-LINK virtual COM-портом.

6. Подключение второго узла. Возможны три варианта проверки:

29. Вторая STM32 с такой же прошивкой, два узла соединены через трансиверы (например, TJA1050) с согласующими 120 Ом на концах шины.

30. Один STM32 + USB-CAN-адаптер (PCAN-USB, CANable). На стороне ПК запускается candump / cansend / python-can для генерации и приёма кадров.

31. Полностью программная симуляция: bxCAN на STM32 заменяется обёрткой на ПК; обмен идёт через виртуальный vcan0 между двумя процессами python-can. Используется для отработки логики прикладного слоя без аппаратуры.

Пример выполнения задания

Рассмотрим разбор «нулевого» варианта: плата Nucleo-F446RE, два узла. Узел А (STM32) отправляет по нажатию USER-кнопки кадр с ID 0x123 и 8 байтами счётчика. Узел В (ПК + USB-CAN-адаптер) отвечает кадром с ID 0x321; STM32 принимает его в FIFO0 и выводит содержимое в UART2 (115200 8N1). Битрейт – 500 кбит/с.

Шаг 1. Настройка проекта в STM32CubeIDE 1.16+ (встроенный графический CubeMX). Включаются периферии: CAN1 (Mode = Activated, Tx = PA12, Rx = PA11), USART2 (Asynchronous, 115200), GPIO кнопки PC13 (input, EXTI15_10), GPIO светодиода PA5 (output). Bit Timing для PCLK1 = 42 МГц: Prescaler = 6, Time Quanta in BS1 = 11, BS2 = 2 – итог 500 кбит/с при точке выборки 85.7 %. NVIC: разрешаются прерывания CAN1 RX0, USART2 (опционально), EXTI15_10.

Шаг 2. Инициализация фильтра приёма. Фильтр банка 0 настраивается в режиме Mask 32 бит, в FIFO0, маска нулевая – принимаются все кадры. Код листинга 1 вставляется в функцию main() сразу после MX_CAN1_Init.

Листинг 1 – Инициализация фильтра CAN и запуск контроллера

```
CAN_FilterTypeDef filter = {0};
filter.FilterBank = 0;
filter.FilterMode = CAN_FILTERMODE_IDMASK;
filter.FilterScale = CAN_FILTERSCALE_32BIT;
filter.FilterIdHigh = 0x0000;
filter.FilterIdLow = 0x0000;
filter.FilterMaskIdHigh = 0x0000;
filter.FilterMaskIdLow = 0x0000;
filter.FilterFIFOAssignment = CAN_FILTER_FIFO0;
filter.FilterActivation = ENABLE;

HAL_CAN_ConfigFilter(&hcan1, &filter);
HAL_CAN_Start(&hcan1);
HAL_CAN_ActivateNotification(&hcan1,
    CAN_IT_RX_FIFO0_MSG_PENDING);
```

Шаг 3. Отправка кадра по нажатию кнопки. В EXTI-callback формируется заголовок и вызывается HAL_CAN_AddTxMessage. Счётчик хранится в файловой переменной.

Листинг 2 – Отправка CAN-кадра по нажатию кнопки USER

```
static uint32_t tx_counter = 0;

void HAL_GPIO_EXTI_Callback(uint16_t pin)
{
    if (pin != GPIO_PIN_13) return;

    CAN_TxHeaderTypeDef tx = {0};
    tx.StdId = 0x123;
    tx.IDE = CAN_ID_STD;
    tx.RTR = CAN_RTR_DATA;
    tx.DLC = 8;

    uint8_t data[8];
    for (int i = 0; i < 4; ++i)
        data[i] = (tx_counter >> (8 * i)) & 0xFF;
    for (int i = 4; i < 8; ++i)
        data[i] = 0xA5;

    uint32_t mailbox;
    HAL_CAN_AddTxMessage(&hcan1, &tx, data, &mailbox);
    tx_counter++;
    HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_5);
}
```

Шаг 4. Обработка приёма и вывод в UART. Callback HAL_CAN_RxFifo0MsgPendingCallback читает кадр из FIFO0 и форматирует строку для UART2.

Листинг 3 – Приём CAN-кадра и вывод содержимого по UART2

```
void HAL_CAN_RxFifo0MsgPendingCallback(CAN_HandleTypeDef *hcan)
{
    CAN_RxHeaderTypeDef rx;
    uint8_t data[8] = {0};

    if (HAL_CAN_GetRxMessage(hcan, CAN_RX_FIFO0,
                             &rx, data) != HAL_OK) {
        return;
    }

    char buf[80];
    int n = snprintf(buf, sizeof(buf),
        "RX id=0x%03lX dlc=%lu data=%02X %02X %02X %02X "
        "%02X %02X %02X %02X\r\n",
        rx.StdId, rx.DLC,
        data[0], data[1], data[2], data[3],
        data[4], data[5], data[6], data[7]);
}
```

```

    HAL_UART_Transmit(&huart2, (uint8_t*)buf, n, 100);
}

```

Шаг 5. Второй узел на стороне ПК. На ПК через USB-CAN-адаптер (или vcan для отладки) запускается скрипт python-can, который слушает ID 0x123 и в ответ передаёт ID 0x321 с фиксированными данными.

Листинг 4 – Второй узел на python-can (ПК с USB-CAN-адаптером)

```

import can

bus = can.interface.Bus(channel='can0',
                        bustype='socketcan',
                        bitrate=500000)
print('node B started, waiting for 0x123...')
for msg in bus:
    if msg.arbitration_id != 0x123:
        continue
    print('RX', hex(msg.arbitration_id),
          msg.data.hex())
    reply = can.Message(arbitration_id=0x321,
                        data=bytes(range(8)),
                        is_extended_id=False)
    bus.send(reply)

```

Шаг 6. Ожидаемый вывод. После прошивки и подключения по USB-COM-порту терминал ПК (например, picocom /dev/ttyACM0 -b 115200) при каждом нажатии кнопки покажет приём ответного кадра 0x321.

Листинг 5 – Ожидаемый вывод в терминале ПК

```

RX id=0x321 dlc=8 data=00 01 02 03 04 05 06 07
RX id=0x321 dlc=8 data=00 01 02 03 04 05 06 07
RX id=0x321 dlc=8 data=00 01 02 03 04 05 06 07

```

Светодиод РА5 переключает состояние при каждом нажатии кнопки – подтверждение, что обработчик EXTI отработал и отправка прошла успешно. Если ответ не приходит, нужно проверить согласующие 120 Ом, фактическую полярность CAN_H / CAN_L и совпадение битрейта на обоих узлах.

Задания по вариантам

Номер варианта определяется как $(N - 1) \bmod 15 + 1$, где N – порядковый номер студента в журнале группы. Конкретные числовые параметры задачи берутся из приведённой ниже таблицы. Составить отчет со скриншотами, показывающие выполнение заданий.

Вариант 1.

- 22. В STM32CubeIDE 1.16+ создать проект для Nucleo-F446RE: включить CAN1 (битрейт 125 кбит/с, точка выборки 87.5 %), TIM2 (период 100 мс, прерывание по обновлению), USART2 (115200 8N1).
- 23. Реализовать периодическую отправку кадра ID 0x100 (8 байт – значение SysTick) в обработчике TIM2.
- 24. Реализовать на ПК узел-ответчик на python-can: при приёме ID 0x100 он сразу отправляет в ответ ID 0x101 с тем же содержимым.
- 25. Замерить задержку «запрос – ответ» по разнице меток времени входящего ID 0x101 и исходящего ID 0x100 на стороне STM32 (вывод в UART); усреднить по 1000 циклам.
- 26. В отчёт включить конфигурацию CubeMX (скриншот периферий), ключевые фрагменты прошивки, статистику задержки и вывод о её разбросе.

Вариант 2.

- 23. Создать проект для STM32, битрейт 250 кбит/с, USART2 для отладки.
- 24. Настроить два аппаратных банка фильтров bxCAN в режиме FILTERMODE_IDLIST 16-bit: принимать только ID 0x200 и ID 0x201, остальные отбрасывать.
- 25. На ПК через cangen vcan0 -I i (или адаптер CAN-USB) сгенерировать поток кадров с ID от 0x100 до 0x300; в callback приёма выводить через UART только принятые ID.
- 26. Подсчитать через UART-лог число принятых и отброшенных кадров; подтвердить, что в логе встречаются только 0x200 и 0x201.

27. В отчёт включить настройку фильтра (значения регистров), лог UART, таблицу «ID – число входящих – число принятых».

Вариант 3.

23. Битрейт 500 кбит/с. На STM32 реализовать передатчик RTR-кадра с ID 0x300 (DLC = 8, RTR = CAN_RTR_REMOTE) по нажатию USER-кнопки.

24. На ПК через python-can настроить узел-ответчик, который при приёме RTR-запроса ID 0x300 возвращает кадр данных с тем же ID и нагрузкой – значением счётчика обработанных запросов.

25. STM32 при получении кадра данных ID 0x300 выводит содержимое в UART.

26. Замерить полное время «запрос RTR – приём данных» на стороне STM32 через DWT_CYCNT.

27. В отчёт включить код двух узлов, фрагменты лога UART, статистику замеров за 200 запросов.

Вариант 4.

23. Битрейт 1 Мбит/с. Настроить ADC1 IN0 (вход PA0) в режиме непрерывной выборки через DMA с периодом усреднения 10 мс (таймер TIM2).

24. Отправлять усреднённое значение АЦП (12-битный код, упакован в первые 2 байта) в кадре ID 0x150 с периодом 10 мс.

25. На ПК через python-can принимать поток и в matplotlib строить график «время – значение АЦП» в реальном времени (FuncAnimation, окно 5 секунд).

26. Подать на вход PA0 пилообразный сигнал от внешнего генератора (или потенциометр) и убедиться визуально, что график соответствует входу.

27. В отчёт включить настройку ADC + DMA в CubeMX, скриншот графика на ПК и оценку фактической частоты передачи кадров.

Вариант 5.

23. Битрейт 500 кбит/с, расширенный 29-битный формат, ID = 0x18FF1234.

24. Настроить фильтр приёма `bxCAN` в режиме `IDMASK` 32-bit: маска выставлена так, чтобы пропускать только этот конкретный 29-битный ID, все остальные кадры (в том числе 11-битные) отбрасывать. Учесть, что в регистре фильтра ID сдвигается на 3 бита влево и активируются биты `IDE` + `RTR`.
25. С ПК через `cangen` попеременно слать 11-битные кадры (ID `0x100`, `0x200`) и расширенные (`0x18FF1234`, `0x18FF1235`); подтвердить по `UART`, что приходит только `0x18FF1234`.
26. В отчёт включить расчёт значений `FilterIdHigh` / `FilterMaskIdHigh`, лог `UART`, обоснование «почему фильтр работает».

Вариант 6.

23. Битрейт 500 кбит/с. Реализовать на `STM32` узел-наблюдатель: принимать любые входящие кадры в `FIFO0`.
24. Каждую секунду по таймеру `TIM2` выводить в `UART` статистику: число принятых кадров за последнюю секунду, общее число кадров с начала работы, значения регистров `CAN_ESR` (`REC`, `TEC`, текущее состояние error: `Active` / `Passive` / `Bus-Off`).
25. Спровоцировать с ПК ошибки (`cangen` с битом ошибки или отключение трансивера) и зафиксировать переход в `Error Passive` по росту `REC`.
26. Реализовать автоматический выход из `Bus-Off` через `HAL_CAN_Stop` / `HAL_CAN_Start`.
27. В отчёт включить код опроса `CAN_ESR`, лог `UART` за сеанс с ошибками, таблицу «момент времени – `REC` – состояние».

Вариант 7.

23. Битрейт 250 кбит/с. Реализовать на `STM32` одновременную загрузку двух почтовых ящиков `bxCAN`: `TX0` – высокоприоритетный ID `0x010`, `TX1` – ID `0x500`.

24. По нажатию USER-кнопки последовательно вызвать HAL_CAN_AddTxMessage для обоих ID; зафиксировать порядок появления кадров на шине через ПК-приёмник.
25. На ПК через candump -L и сортировку по меткам времени показать, что bxCAN сам выводит ID 0x010 раньше ID 0x500, независимо от порядка добавления.
26. Повторить опыт, добавляя сначала ID 0x500, затем ID 0x010, и подтвердить тот же результат.
27. В отчёт включить код опыта, два варианта дампа и обоснование через механизм приоритетной очереди bxCAN.

Вариант 8.

23. Битрейт 500 кбит/с. Включить внутренний температурный датчик STM32 через ADC1 (канал TEMPSENSOR) с калибровкой по TS_CAL1 / TS_CAL2 из системной памяти.
24. Реализовать на STM32 узел-термометр: при получении запроса с ID 0x080 (DLC любой) STM32 в ответ отправляет кадр ID 0x180 (4 байта – температура в десятых долях градуса, int32_t little-endian).
25. На ПК через python-can раз в секунду слать запрос ID 0x080 и выводить пришедшее значение температуры с переводом в °C.
26. Подтвердить корректность калибровки нагревом / охлаждением корпуса контроллера (палец, фен холодным воздухом).
27. В отчёт включить формулу калибровки, код обоих узлов, лог за 60 секунд с графиком температуры.

Вариант 9.

23. Битрейт 125 кбит/с. Симулировать второй узел полностью программно: STM32 + SLCAN-адаптер CANable 2.0 (драйвер slcand) против python-can на этом же ПК через vcan0 не получится – CANable работает с can0, а vcan0 – виртуальный; поэтому использовать связку CANable как can0 и python-can на can0.

24. На STM32 отправлять последовательно 1000 кадров с ID 0x100 и нагрузкой – счётчиком (uint32_t little-endian).
25. На ПК принимать поток в python-can, проверять монотонный рост счётчика и отсутствие пропусков.
26. Зафиксировать число успешно принятых кадров и при наличии пропусков – их позиции в последовательности.
27. В отчёт включить настройку slcand (slcand -o -c -s4 /dev/ttyACM0 can0), код передатчика, лог проверки, статистику потерь.

Вариант 10.

23. Битрейт 500 кбит/с. Настроить два банка фильтров bxCAN: банк 0 – FILTERMODE_IDMASK 32-bit на ID < 0x400 (маска и значение подбираются по старшим битам), назначение FIFO0; банк 1 – по остальным, FIFO1.
24. Реализовать отдельные callback HAL_CAN_RxFifo0MsgPendingCallback и HAL_CAN_RxFifo1MsgPendingCallback; в каждом увеличивать счётчик и выводить «FIFO0: ID=...» / «FIFO1: ID=...».
25. Сгенерировать с ПК поток кадров с ID 0x100, 0x200, 0x300, 0x400, 0x500; подтвердить через UART, какие ID попали в какой FIFO.
26. Подсчитать счётчики приёма по обоим FIFO и сопоставить с генератором.
27. В отчёт включить расчёт масок банков 0 и 1, лог UART, таблицу «ID – FIFO – число принятых».

Вариант 11.

23. Битрейт 500 кбит/с. Включить счётчик циклов DWT_CYCNT (SystemCoreClock = 84 МГц у F446) для точного измерения интервалов.
24. При каждом нажатии USER-кнопки на STM32 отправлять кадр ID 0x111 с полезной нагрузкой – текущим значением DWT_CYCNT (4 байта little-endian).
25. На ПК через python-can принимать кадры; по разнице значений DWT соседних кадров рассчитывать интервал между нажатиями (с учётом частоты ядра).

26. Сделать 20 нажатий с произвольными интервалами; подтвердить соответствие расчёта секундомеру.

27. В отчёт включить формулу перевода тиков в секунды, код обработчика EXTI, лог 20 нажатий и таблицу «интервал по DWT vs секундомер».

Вариант 12.

23. Битрейт 250 кбит/с. По нажатию USER-кнопки в EXTI-callback подряд заполнить все три mailbox bxCAN кадрами ID 0x201, 0x202, 0x203 (DLC = 8, разные данные).

24. После загрузки трёх ящиков подождать через цикл HAL_CAN_IsTxMessagePending освобождения всех трёх и измерить общее время передачи серии через DWT.

25. На ПК через candump зафиксировать, что все три ID пришли подряд (по меткам времени интервал $\approx$ длительность одного кадра $\approx$ 525 мкс).

26. Сравнить замер с теоретическим расчётом $3 \cdot L / R_bit$.

27. В отчёт включить код загрузки трёх mailbox, фрагмент дампа, теорию и эксперимент в одной таблице.

Вариант 13.

23. Битрейт 500 кбит/с. На STM32 реализовать «heartbeat»: каждую секунду по таймеру TIM2 отправлять кадр ID 0x700 без полезной нагрузки (DLC = 0).

24. На ПК через python-can реализовать наблюдатель: при отсутствии кадра ID 0x700 в течение 3 секунд писать в журнал «node A down [timestamp]»; при возобновлении – «node A up».

25. Имитировать отказ узла: на 5 секунд отключить трансивер от шины (вытащить провод CAN_H) и затем подключить обратно.

26. Зафиксировать в логе сообщения «down» и «up» с метками времени, оценить точность срабатывания ($\approx$ 3 с с момента отключения).

27. В отчёт включить код таймера, код наблюдателя, лог эксперимента и обоснование выбора порога 3 секунды.

Вариант 14.

23. Создать на STM32 две прошивки (или одну с переключением по USER-кнопке) с битрейтами 500 кбит/с и 1 Мбит/с.
24. Для каждого битрейта реализовать обмен «запрос – ответ»: STM32 шлёт ID 0x100 (8 байт), ПК отвечает ID 0x101.
25. Замерить через DWT_CYCCNT на STM32 полное время «запрос – ответ»; усреднить по 1000 циклам.
26. Сравнить замеры на двух битрейтах, рассчитать теоретическую разницу (вклад собственно передачи кадра).
27. В отчёт включить код измерения, таблицу avg / min / max задержки для двух битрейтов, вывод о доле сетевой задержки в общем времени отклика.

Вариант 15.

5. Битрейт 500 кбит/с. На STM32 реализовать ретранслятор: фильтр приёма настроен только на ID 0x100, в callback HAL_CAN_RxFifo0MsgPendingCallback взять полученные 8 байт, увеличить первый байт на 1 и сразу отправить кадр с ID 0x101 (остальные байты – без изменений).
6. Записать в callback метку DWT_CYCCNT входа и DWT_CYCCNT после HAL_CAN_AddTxMessage; разница и есть задержка ретрансляции.
7. На ПК через python-can сгенерировать поток ID 0x100 с частотой 100 Гц и принимать ответный поток ID 0x101.
8. Подсчитать avg / max задержку ретрансляции по 1000 пакетам.
9. В отчёт включить код ретранслятора, гистограмму задержки, вывод о применимости такой схемы в качестве шлюза.

Лабораторная работа 7. Программная эмуляция EtherCAT (SofEtherCAT или простейший симулятор)

Цель работы: освоить принцип работы протокола EtherCAT на примере открытой библиотеки SOEM (Simple Open EtherCAT Master). Развернуть мастер на ПК с Linux, поднять виртуальную линейку slave-устройств (программный эмулятор), запустить циклический обмен процесс-образом, измерить достижимое время цикла, продемонстрировать принцип «processing on the fly» – модификацию данных в кадре, проходящем через slave, без полной буферизации, и сравнить результат с теоретической оценкой.

Теоретическое обоснование

1. Место EtherCAT среди промышленных Real-Time Ethernet решений. EtherCAT (Ethernet for Control Automation Technology) – стандарт промышленной сети, разработанный Beckhoff и закреплённый в IEC 61158/61784. Использует физический и канальный уровень обычного Ethernet 100Base-TX, но строит логику передачи принципиально иначе: вместо набора независимых пакетов между master и каждым slave используется один большой кадр, циркулирующий по кольцу (фактически – по линии, замкнутой обратно через закрытые порты). Это даёт детерминированное время цикла на уровне десятков микросекунд при сотнях узлов.

2. Топология «линия / кольцо» и принцип «processing on the fly». Slave-устройства подключаются гирляндой: каждый узел имеет два порта (in / out). Master отправляет кадр в первый slave; тот аппаратно (через EtherCAT Slave Controller, ESC) считывает свою порцию данных и одновременно вписывает свои входные данные в проходящий кадр, не задерживая передачу более чем на ~1 мкс. Кадр идёт дальше по линии, последний slave разворачивает его, и кадр возвращается к master через закрытые обратные порты. Графически архитектура показана на рисунке 1.

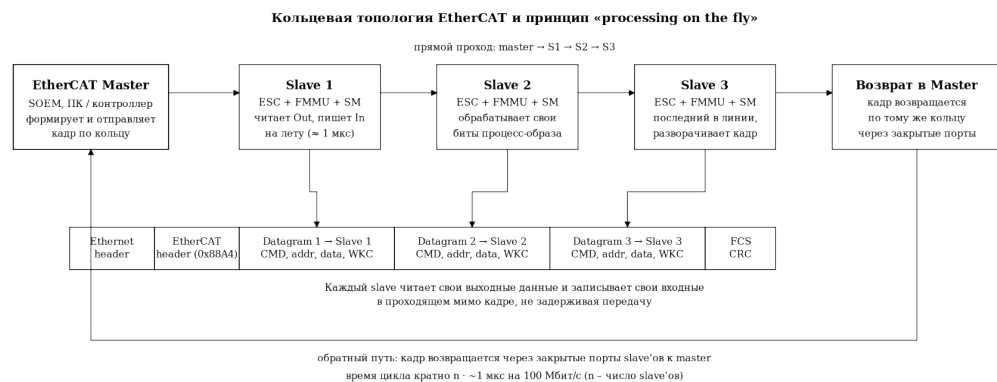


Рисунок 1 – Топология EtherCAT и принцип «processing on the fly»

3. Формат кадра. Кадр EtherCAT инкапсулируется в Ethernet с EtherType = 0x88A4 (либо в UDP-датаграмму). После заголовка EtherCAT идёт один или несколько Datagram'ов; каждый содержит команду доступа (например, LRW – Logical Read/Write, BRD – Broadcast Read, APRW – Auto-increment Physical Read/Write), адрес, длину, поле данных и счётчик Working Counter (WKC). Каждый slave, обрабатывая Datagram, инкрементирует WKC; сравнение прочитанного master'ом WKC с ожидаемым позволяет обнаружить, что все назначенные slave'ы участвовали в обмене.

4. Память slave-устройства: FMMU и SyncManager. ESC содержит 8 КБ DPRAM. Прикладные данные располагаются по фиксированным адресам, а доступ к ним со стороны мастера организуется через:

32. FMMU (Fieldbus Memory Management Unit) – блок отображения логического адреса (общего процесс-образа) в физический адрес DPRAM конкретного slave. Каждый FMMU описывает диапазон, направление (чтение / запись) и битовое смещение.

33. SyncManager – блок управления синхронизацией доступа: режим 3-буфера (для процесс-данных, без блокировок) или режим mailbox (для асинхронных сообщений CoE/EoE).

34. Distributed Clocks (DC) – аппаратная синхронизация часов всех slave'ов от первого опорного с точностью $< 1 \text{ мкс}$.

5. State Machine EtherCAT (ESM). Каждый slave проходит состояния INIT → PRE-OP → SAFE-OP → OP. Переходы инициирует master, записывая

значение в регистр AL Control (0x0120). В состоянии OP разрешены циклические PDO-передачи; в PRE-OP – только mailbox-сообщения (конфигурация CoE).

6. Библиотека SOEM. SOEM (Simple Open EtherCAT Master) – переносимая C-библиотека с открытым исходным кодом (лицензия GPL v2). Актуальная стабильная версия – 1.4.0 (релиз 2023 г., поддерживается репозиторием OpenEtherCATsociety / SOEM на GitHub). Реализует функции мастера: сканирование сети (ec_init / ec_config_init), построение карты процесс-образа (ec_config_map), переход в OP (ec_writestate), обмен PDO (ec_send_processdata + ec_receive_processdata). Утилиты simple_test, slaveinfo идут в составе примеров. Для запуска требуется raw-сокет (CAP_NET_RAW в Linux) и сетевой интерфейс, не управляемый штатным стеком (рекомендуется выделенный порт).

Таблица 1 – Сравнение EtherCAT и обычного TCP/IP

Критерий	EtherCAT	TCP/IP
Топология	Линия / кольцо, гирлянда slave'ов	Произвольная, через коммутаторы
Время цикла, типичное	30...500 мкс на десятки узлов	1...10 мс и выше (зависит от стека)
Детерминированность задержки	Жёсткая, < 1 мкс jitter с DC	Мягкая, зависит от очередей коммутатора и ОС
Накладные расходы на узел	Десятки байт в одном общем кадре	Полный Ethernet-кадр на каждое сообщение
Обработка в slave	На лету, аппаратно через ESC	Полная буферизация в стеке TCP/IP
Типовое применение	Сервоприводы, ввод/вывод реального времени	Конфигурация, диагностика, верхний уровень

7. Виртуальный slave. Аппаратные ESC недоступны в учебной лаборатории, поэтому в работе используется программный эмулятор slave,

который слушает raw-сокеты, разбирает приходящие EtherCAT-кадры, в проходящем датаграмме инкрементирует WКС и при заданном смещении подменяет байты данных, после чего пересылает кадр следующему узлу (или возвращает master'у). Это позволяет на одном ПК запустить мастер и один – три slave'а и пронаблюдать модификацию данных «на лету».

Пример выполнения задания

Рассмотрим разбор «нулевого» варианта: на одном ПК с Linux поднимаются два виртуальных сетевых интерфейса (veth-пара), запускается SOEM-мастер на одном конце и программный slave – на другом. Slave при каждом проходе кадра инкрементирует младший байт процесс-образа, мастер циклически читает результат. Требуется измерить достижимое время цикла и убедиться, что значение растёт детерминированно.

Шаг 1. Подготовка сетевой среды. Команды листинга 1 создают пару veth-интерфейсов veth_m (мастер) и veth_s (slave), поднимают их и выводят список.

Листинг 1 – Создание veth-пары для эмуляции одиночного канала EtherCAT

```
sudo ip link add veth_m type veth peer name veth_s
sudo ip link set veth_m up
sudo ip link set veth_s up
ip -brief link show | grep veth
```

Шаг 2. Сборка SOEM и примера simple_test. Исходники клонируются с GitHub, собираются через CMake, бинарь simple_test появляется в build/test/linux/simple_test/.

Листинг 2 – Сборка SOEM 1.4.0 с примером simple_test

```
git clone --branch v1.4.0 \
  https://github.com/OpenEtherCATsociety/SOEM.git
cd SOEM
mkdir build && cd build
cmake -DBUILD_TESTS=ON ..
make -j
ls test/linux/simple_test/simple_test
```

Шаг 3. Программный slave на python (scapy). Скрипт листинга 3 слушает veth_s, по каждому EtherCAT-кадру инкрементирует счётчик процесс-образа, увеличивает WКС и отправляет кадр обратно. Запуск – с правами CAP_NET_RAW (через sudo).

Листинг 3 – Минимальный эмулятор EtherCAT slave на scapy

```
from scapy.all import Ether, sniff, sendp, Raw
import struct

IFACE = 'veth_s'
ECAT = 0x88A4
counter = 0

def handle(pkt):
    global counter
    if pkt.type != ECAT:
        return
    payload = bytes(pkt.payload)
    # EtherCAT header: 2 байта (длина + reserved + type)
    if len(payload) < 12:
        return
    # datagram: cmd(1) idx(1) addr(4) len(2) irq(2) data(len) wkc(2)
    body = bytearray(payload)
    counter = (counter + 1) & 0xFF
    body[10] = counter      # подмена первого байта данных
    wkc_off = len(body) - 2
    wkc = struct.unpack('<H', bytes(body[wkc_off:wkc_off+2]))[0]
    struct.pack_into('<H', body, wkc_off, wkc + 1)
    new = Ether(src=pkt.dst, dst=pkt.src, type=ECAT) / Raw(bytes(body))
    sendp(new, iface=IFACE, verbose=False)

sniff(iface=IFACE, prn=handle, store=0, filter='ether proto 0x88a4')
```

Шаг 4. Запуск мастера SOEM. simple_test принимает имя сетевого интерфейса аргументом командной строки. Если в сети есть хотя бы один slave (в нашем случае – эмулятор), мастер переведёт его в состояние ОР и начнёт циклический обмен.

Листинг 4 – Запуск SOEM-мастера на veth_m

```
# в первом терминале - эмулятор slave
sudo python3 vslave.py

# во втором терминале - SOEM master
```

```
cd SOEM/build/test/linux/simple_test
sudo ./simple_test veth_m
```

Шаг 5. Измерение времени цикла. В файл `simple_test.c` добавляется (или используется уже встроенный) подсчёт интервала между двумя вызовами `ec_send_processdata` через `clock_gettime(CLOCK_MONOTONIC)`. Усреднение по 1000 циклов и выводится статистика. Фрагмент модификации – листинг 5.

Листинг 5 – Замер времени цикла обмена процесс-данными

```
struct timespec t0, t1;
long min = LONG_MAX, max = 0, sum = 0;
const int N = 1000;

for (int i = 0; i < N; ++i) {
    clock_gettime(CLOCK_MONOTONIC, &t0);
    ec_send_processdata();
    wkc = ec_receive_processdata(EC_TIMEOUTRET);
    clock_gettime(CLOCK_MONOTONIC, &t1);
    long ns = (t1.tv_sec - t0.tv_sec) * 1000000000L
        + (t1.tv_nsec - t0.tv_nsec);
    if (ns < min) min = ns;
    if (ns > max) max = ns;
    sum += ns;
    usleep(1000); /* нагрузочный период 1 мс */
}
printf("cycle: avg=%ld us min=%ld us max=%ld us\n",
    sum / N / 1000, min / 1000, max / 1000);
```

Шаг 6. Ожидаемый вывод. На обычном Linux-ядре (без PREEMPT_RT) и одном эмулируемом slave типичный результат – листинг 6.

Листинг 6 – Пример вывода SOEM-мастера с подсчётом времени цикла

```
SOEM (Simple Open EtherCAT Master)
Starting simple test
ec_init on veth_m succeeded.
1 slaves found and configured.
Slaves mapped, state to SAFE_OP.
Calculated workcounter 1
Request operational state for all slaves
Operational state reached for all slaves.
cycle: avg=180 us min=120 us max=950 us
first byte of input image: 0x01, 0x02, 0x03, 0x04, ...
```

Первый байт входного образа меняется на каждом цикле, подтверждая, что slave модифицирует данные в проходящем кадре («processing on the fly»).

Большой `max` объясняется тем, что ядро не реального времени; на ядре `PREEMPT_RT` `max` сокращается до десятков мкс. В вариантах требуется повторить опыт и сопоставить результат с теоретической оценкой времени цикла.

Задания по вариантам

Номер варианта определяется как $(N - 1) \bmod 15 + 1$, где N – порядковый номер студента в журнале группы. Конкретные числовые параметры задачи берутся из приведённой ниже таблицы. Составить отчет со скриншотами, показывающие выполнение заданий.

Вариант 1.

27. Развернуть SOEM 1.4.0 и эмулятор `slave` из примера; запустить обмен с одним `slave`, период цикла 1 мс, 1000 итераций.
28. Снять `avg` / `min` / `max` времени цикла через `clock_gettime (CLOCK_MONOTONIC)` и сохранить массив RTT в файл.
29. В `python` построить гистограмму распределения; подсчитать процентиля `p50`, `p95`, `p99`.
30. В отчёт включить команды запуска, фрагмент изменения `simple_test.c` с замером, гистограмму, таблицу `avg` / `min` / `max` / `p99` и вывод о джиттере на обычном ядре.

Вариант 2.

28. Развернуть два эмулируемых `slave` цепочкой: между ними поднять две `veth`-пары и `Linux`-бридж (или один процесс, пересылающий кадр от первого ко второму).
29. Прогнать SOEM с двумя `slave` при периоде цикла 1 мс, 1000 итераций.
30. Сравнить `avg` / `max` времени цикла с результатом одиночного `slave` из варианта 1 (или с собственным эталонным замером).
31. В отчёт включить схему цепочки, два набора замеров «1 `slave` / 2 `slave`», график зависимости `avg` от числа `slave` и обоснование линейного роста.

Вариант 3.

28. Один эмулируемый slave; период цикла 500 мкс, 10 000 итераций.
29. В цикле опроса фиксировать факт, что измеренный интервал превышает заданный период (cycle overrun), и подсчитывать число таких событий.
30. Построить распределение задержки в логарифмических осях (matplotlib hist log-log), подсветить «хвост».
31. В отчёт включить число overrun, p99, p99.9 и вывод о том, пригодно ли обычное ядро для цикла 500 мкс.

Вариант 4.

28. Один эмулируемый slave, период цикла 100 мкс, 100 000 итераций.
29. Определить процент итераций, уложившихся в дедлайн 100 мкс, и средний overshoot для тех, что вышли за рамки.
30. Сравнить результат с теоретической оценкой – задержка сетевого гав-сокета плюс время обработки эмулятором.
31. В отчёт включить процент успешных циклов, средний и максимальный overshoot, обоснованный вывод о непригодности ядра non-RT для такого периода (рекомендация PREEMPT_RT).

Вариант 5.

27. Модифицировать эмулятор slave: в проходящем датаграмме читать первые 4 байта данных, инкрементировать их и записывать обратно в эти же 4 байта (а не подменять только один байт, как в примере).
28. В SOEM-мастере по адресу процесс-образа считывать возвращаемое значение и проверять, что разность с переданным равна 1 на каждом цикле.
29. Подсчитать число расхождений за 10 000 циклов.
30. В отчёт включить diff к примеру vslave.py, код проверки в мастере, статистику расхождений и вывод о корректности механизма «processing on the fly».

Вариант 6.

28. Прогнать обмен SOEM + один slave, период 1 мс, 1000 итераций при обычном приоритете процесса (SCHED_OTHER).
29. Повторить с приоритетом SCHED_FIFO 80 (chrt -f 80) и закреплением за ядром (taskset -c 1).
30. Сравнить avg / max / σ для двух случаев; объяснить разницу через политику планировщика.
31. Дополнительно (если доступно ядро PREEMPT_RT) повторить замер на нём и сопоставить с обычным.
32. В отчёт включить три набора замеров, таблицу сравнения и вывод о вкладе приоритета и реального времени.

Вариант 7.

28. Один slave, период 1 мс, 60 секунд работы.
29. Параллельно запустить фоновую нагрузку stress-ng --cpu N (N – число ядер машины) и зафиксировать avg / max времени цикла под нагрузкой.
30. Повторить замер при поднятии приоритета SOEM до SCHED_FIFO 80 (chrt -f 80) с той же нагрузкой.
31. Подсчитать процент превышений периода в двух режимах.
32. В отчёт включить графики «время цикла vs итерация» (без приоритета и с приоритетом), таблицу avg / max, вывод об эффективности SCHED_FIFO.

Вариант 8.

28. Развернуть три эмулируемых slave (через цепочку bridge или три процесса) с разным размером процесс-образа: 8, 32, 128 байт.
29. Прогнать SOEM с тремя slave при периоде 1 мс, 1000 итераций.
30. Раздельно замерить avg времени цикла для каждой комбинации и построить зависимость avg(размер процесс-образа).
31. Сравнить с теоретической оценкой – линейный рост от размера Ethernet-кадра (1 байт $\approx$ 80 нс на 100 Мбит/с).

32. В отчёт включить график $\text{avg}(\text{size})$, линейную регрессию, коэффициент наклона и сравнение с теорией.

Вариант 9.

28. Реализовать в эмуляторе slave логику «эхо-счётчика»: первое слово (4 байта) кадра LRW – счётчик мастера; slave читает его и записывает во второе слово свой ответ – счётчик + 1.

29. На стороне SOEM мастера на каждом цикле инкрементировать счётчик в первом слове и сверять второе слово с ожидаемым (прочитанное на предыдущем цикле + 1).

30. Прогнать обмен в течение 10 000 циклов; зафиксировать число расхождений и моменты их возникновения.

31. В отчёт включить код slave, код проверки в мастере, статистику расхождений и вывод о детерминированности.

Вариант 10.

28. Модифицировать эмулятор slave так, чтобы при каждом 100-м проходящем кадре не инкрементировать WKC.

29. В SOEM-мастере отслеживать значение WKC после `es_receive_processdata`; при отклонении от ожидаемого (`expectedWKC`) записывать в лог номер цикла.

30. Прогнать 1000 циклов и подтвердить, что в логе ровно 10 событий «WKC mismatch», на циклах 100, 200, 300, ...

31. Дополнительно проверить, переводит ли SOEM такой slave в состояние ошибки (`es_state`) при длительном расхождении.

32. В отчёт включить diff к эмулятору, код проверки WKC в мастере, лог рассогласований и обоснование штатной реакции EtherCAT на «отсутствующий» slave.

Вариант 11.

28. Один slave, период 1 мс, 1000 кадров.

29. Параллельно запустить `tcpdump -i veth_m -w cap.pcap 'ether proto 0x88a4'` для захвата кадров.
30. Открыть `cap.pcap` в Wireshark 4.x с включённым EtherCAT-диссектором; раскрыть структуру: Ethernet header, EtherCAT header, Datagram (cmd, idx, addr, len, data, WKC).
31. Сравнить разобранный кадр с описанием в теории и зафиксировать значения cmd (LRW / BRD / APRW), длину Datagram, наблюдаемый WKC.
32. В отчёт включить скриншот разбора кадра в Wireshark, соответствие полей теории и сводную таблицу команд за дампы.

Вариант 12.

28. Прогнать SOEM с одним slave, период 250 мкс, 10 000 итераций, в режиме обычного потока (SCHED_OTHER).
29. Повторить запуск с использованием `pthread_setschedparam` и политикой SCHED_FIFO 80 для основного цикла обмена.
30. Сравнить avg / max / процент превышений периода в двух режимах.
31. Сделать вывод о минимально достижимом периоде для каждого режима.
32. В отчёт включить фрагменты двух запусков, таблицу замеров, график «время цикла vs итерация» для обоих режимов и вывод о пороге, где SCHED_FIFO даёт качественный выигрыш.

Вариант 13.

28. Развернуть два эмулируемых slave: первый – штатный (читает / пишет данные, инкрементирует WKC); второй – «байпас», просто пересылает кадр дальше без модификаций и без инкремента WKC.
29. Прогнать обмен SOEM в течение 1000 циклов.
30. В Wireshark 4.x с EtherCAT-диссектором подтвердить, что WKC в Datagram, адресованном первому slave, растёт, а в Datagram второго slave – остаётся неизменным.
31. Подсчитать ожидаемый expectedWKC SOEM при такой конфигурации.

32. В отчёт включить код двух эмуляторов, скриншот Wireshark с подсветкой WКС, расчёт expectedWКС и вывод о роли WКС в детектировании сбоев.

Вариант 14.

28. Модифицировать эмулятор slave: ввести искусственную задержку `usleep(200)` в обработчике перед отправкой кадра обратно.

29. Прогнать SOEM с одним slave, период 1 мс, 1000 итераций.

30. Замерить `avg` / `max` времени цикла; сравнить с эталоном без задержки и объяснить, почему вырастает именно `max`, а не `avg` (на самом деле растёт и среднее – обосновать).

31. Построить распределение задержки и показать смещение всего распределения вправо на ≈ 200 мкс.

32. В отчёт включить `diff` к эмулятору, две гистограммы (с задержкой и без), таблицу `avg` / `max`, обоснование смещения.

Вариант 15.

10. Один slave, период 1 мс, 60 секунд работы (60 000 циклов).

11. Логировать время каждого цикла; разбить логи на окна по 1 секунде и в каждом окне посчитать `avg` / `max`.

12. Построить два графика «время – `avg`» и «время – `max`» с разрешением 1 секунда.

13. Найти на графиках всплески и попытаться сопоставить с событиями (`htop` в параллели, активность системы).

14. В отчёт включить оба графика, таблицу глобальной статистики за 60 секунд и вывод о долговременной стабильности обмена.

Лабораторная работа 8. Производительность EtherCAT vs стандартный TCP/IP

Цель работы: сравнить характеристики задержки и джиттера для двух способов передачи данных: классического TCP/IP-сокета и программной эмуляции EtherCAT-обмена через raw-сокета. Отправить $N = 1000$ пакетов по каждому способу, измерить минимальную, среднюю, максимальную задержку «туда – обратно» (RTT) и дисперсию, построить распределения, сделать вывод о применимости каждого механизма в системах жёсткого реального времени.

Теоретическое обоснование

1. Задержка и джиттер как метрики реального времени. В системах управления реального времени важна не только средняя задержка, но и её разброс (джиттер). Большой максимум задержки означает, что хотя бы один цикл управления может не уложиться в дедлайн, что приведёт к потере детерминированности. Принятые обозначения: t_{avg} – среднее, t_{min} – минимум, t_{max} – максимум, σ^2 – дисперсия, σ – стандартное отклонение (джиттер). Для выборки из N измерений $\{t_i\}$:

$$t_{avg} = (1/N) \cdot \sum t_i \quad (1)$$

$$\sigma^2 = (1/N) \cdot \sum (t_i - t_{avg})^2 \quad (2)$$

В системах жёсткого реального времени критерием годности обычно служит $t_{max} \leq T_{cycle}$, а не $t_{avg} \leq T_{cycle}$.

2. Путь пакета в стеке TCP/IP. Отправка одного TCP-сегмента в Linux проходит через прикладной буфер, очередь сокета, расчёт контрольных сумм, очередь qdisc сетевой карты, физическую среду, аск-логику на приёмнике и обратный путь. На каждом уровне возможны задержки: переключения контекста, обработка прерываний softirq, повторные передачи, алгоритм Nagle, буферизация в TCP-стеке. Алгоритм Nagle объединяет короткие сегменты и может задержать отправку на десятки миллисекунд; отключается через TCP_NODELAY. Даже после отключения TCP/IP-стек не даёт жёсткой

гарантии max-задержки на обычном ядре Linux – типичный jitter лежит в районе сотен микросекунд и выше.

3. Путь кадра в EtherCAT. В EtherCAT-сети (или её эмуляции на raw-сокете) мастер формирует кадр в пользовательском пространстве, отдаёт его драйверу через PF_PACKET / AF_PACKET, кадр сразу попадает в сетевую карту без обработки стеком IP / TCP. На приёмной стороне slave (или его эмулятор) обрабатывает кадр на лету и отдаёт обратно. Отсутствие стека TCP/IP, отсутствие маршрутизации и однопроходный механизм доставки дают существенно меньший и более стабильный max.

4. Корректная методика измерения. Чтобы измерение отражало именно сетевую часть, а не накладные расходы на разогрев, необходимо:

35. Выполнить «прогрев» – 100 – 200 первых пакетов отбросить из статистики (page faults, кэширование инструкций, ARP / коммутатор).

36. Использовать монотонные часы (clock_gettime CLOCK_MONOTONIC) и измерять RTT, а не одностороннюю задержку (исключает проблему синхронизации часов).

37. Запускать обе стороны на одной машине через loopback или veth-пару – исключает влияние коммутатора и кабеля; разница между TCP и raw тогда отражает чистый вклад стека.

38. Закрепить процесс за конкретным ядром (taskset -c 1) и повысить приоритет (chrt -f 80 или nice -n -20), чтобы снизить влияние планировщика ОС.

39. Отключить TCP_NODELAY для адекватного сравнения короткоживущих сообщений (иначе задержка раздувается алгоритмом Nagle).

5. Ожидаемые порядки величин. Типовые значения для одного и того же ПК с обычным ядром Linux и сообщения 64 байта на loopback приведены в таблице 1. Конкретные цифры зависят от оборудования и нагрузки и должны быть получены студентом.

Таблица 1 – Типовые порядки задержек на loopback (Linux, обычное ядро)

Метрика	TCP/IP	raw-сокет (EtherCAT-like)
t_avg	30 – 80 мкс	10 – 30 мкс
t_min	20 – 50 мкс	8 – 15 мкс
t_max	0.5 – 5 мс	100 – 300 мкс
σ (джиттер)	Высокий, десятки – сотни мкс	Низкий, единицы – десятки мкс
Доля «выпадающих» пакетов (> 1 мс)	1 – 5 %	Практически 0

6. Особенности обработки результата. Распределение задержки редко нормальное – у TCP оно «тяжёлый хвост» (long-tail). Поэтому помимо σ удобно приводить проценти́ли (например, p99 – значение, ниже которого лежит 99 % измерений). Гистограмма с логарифмической осью Y хорошо показывает выбросы. В отчёте обязательно указывают условия измерения: ядро, нагрузка, приоритет процесса, размер сообщения.

Пример выполнения задания

Рассмотрим разбор «нулевого» варианта: два процесса на одной машине, обмен сообщениями по 64 байта, N = 1000 пакетов, loopback. Для TCP используется обычная пара socket/accept; для «EtherCAT-like» – PF_PACKET (raw Ethernet) на veth-паре с минимальным заголовком EtherCAT (EtherType 0x88A4). Цель – получить avg / min / max / σ и распределение задержек.

Шаг 1. Подготовка окружения. Создаётся veth-пара, как в работе 7, плюс ставится python3-scapy.

Листинг 1 – Подготовка veth-пары и зависимостей

```
sudo ip link add veth_a type veth peer name veth_b
sudo ip link set veth_a up
sudo ip link set veth_b up
```

```
sudo apt install -y python3-scapy
```

Шаг 2. TCP-эхо: сервер и клиент. Сервер слушает порт 5000, принимает 64 байта и возвращает их обратно. Клиент N раз отправляет сообщение, замеряет RTT и сохраняет в numpy-массив. Опция TCP_NODELAY обязательна для устранения Nagle.

Листинг 2 – TCP-эхо-сервер на localhost

```
import socket

srv = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
srv.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
srv.bind(('127.0.0.1', 5000))
srv.listen(1)
conn, _ = srv.accept()
conn.setsockopt(socket.IPPROTO_TCP, socket.TCP_NODELAY, 1)
while True:
    data = conn.recv(64)
    if not data:
        break
    conn.sendall(data)
```

Листинг 3 – TCP-клиент с замером RTT

```
import socket, time
import numpy as np

N = 1000
WARMUP = 100
payload = bytes(64)

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.setsockopt(socket.IPPROTO_TCP, socket.TCP_NODELAY, 1)
s.connect(('127.0.0.1', 5000))

rtt = np.zeros(N)
for i in range(N + WARMUP):
    t0 = time.perf_counter_ns()
    s.sendall(payload)
    s.recv(64)
    t1 = time.perf_counter_ns()
    if i >= WARMUP:
        rtt[i - WARMUP] = (t1 - t0) / 1000.0 # мкс

np.save('rtt_tcp.npy', rtt)
print('TCP', 'avg=%.1f min=%.1f max=%.1f std=%.1f' %
      (rtt.mean(), rtt.min(), rtt.max(), rtt.std()))
```

Шаг 3. EtherCAT-эхо через raw-сокет. Сервер (slave-эмулятор) слушает veth_b, разворачивает src/dst и сразу отправляет кадр обратно. Клиент (мастер) шлёт N кадров EtherType = 0x88A4 на veth_a и замеряет RTT. Оба скрипта запускаются с sudo.

Листинг 4 – EtherCAT-like эхо-slave на raw-сокете (scapy)

```
from scapy.all import Ether, sniff, sendp, Raw

IFACE = 'veth_b'
ECAT = 0x88A4

def echo(pkt):
    if pkt.type != ECAT:
        return
    reply = Ether(src=pkt.dst, dst=pkt.src, type=ECAT) / Raw(bytes(pkt.payload))
    sendp(reply, iface=IFACE, verbose=False)

sniff(iface=IFACE, prn=echo, store=0, filter='ether proto 0x88a4')
```

Листинг 5 – EtherCAT-like клиент-мастер с замером RTT

```
from scapy.all import Ether, sendp, sniff, Raw, conf
import socket, time
import numpy as np

IFACE = 'veth_a'
ECAT = 0x88A4
N = 1000
WARMUP = 100

src = '02:00:00:00:00:01'
dst = '02:00:00:00:00:02'
payload = bytes(64)

s = conf.L2socket(iface=IFACE)
rtt = np.zeros(N)
for i in range(N + WARMUP):
    frame = Ether(src=src, dst=dst, type=ECAT) / Raw(payload)
    t0 = time.perf_counter_ns()
    s.send(frame)
    s.recv()
    t1 = time.perf_counter_ns()
    if i >= WARMUP:
        rtt[i - WARMUP] = (t1 - t0) / 1000.0
```

```

np.save('rtt_raw.npy', rtt)
print('RAW', 'avg=%.1f min=%.1f max=%.1f std=%.1f' %
      (rtt.mean(), rtt.min(), rtt.max(), rtt.std()))

```

Шаг 4. Обработка результатов. Скрипт листинга 6 строит гистограмму и считает процентиля p50, p95, p99 для обоих способов.

Листинг 6 – Расчёт процентилей и сравнительная гистограмма

```

import numpy as np
import matplotlib.pyplot as plt

tcp = np.load('rtt_tcp.npy')
raw = np.load('rtt_raw.npy')

for name, x in (('TCP', tcp), ('RAW', raw)):
    print(name, 'avg=%.1f std=%.1f p50=%.1f p95=%.1f p99=%.1f max=%.1f' %
          (x.mean(), x.std(),
           np.percentile(x, 50), np.percentile(x, 95),
           np.percentile(x, 99), x.max()))

bins = np.logspace(0, 4, 60)
plt.hist(tcp, bins=bins, alpha=0.5, label='TCP')
plt.hist(raw, bins=bins, alpha=0.5, label='RAW (EtherCAT-like)')
plt.xscale('log'); plt.yscale('log')
plt.xlabel('RTT, us'); plt.ylabel('count')
plt.legend(); plt.grid(True)
plt.savefig('latency.png', dpi=150)

```

Шаг 5. Ожидаемый вывод. На обычном ноутбуке с ядром Linux 6.x результат будет близок к листингу 7. Конкретные числа на других машинах отличаются, но качественная картина сохраняется – у TCP больше и среднее, и max, и дисперсия.

Листинг 7 – Пример сводной статистики измерений

```

TCP avg=72.4 std=145.3 p50=58.0 p95=120.0 p99=480.0 max=2100.0
RAW avg=18.6 std=12.4 p50=16.0 p95=32.0 p99=58.0 max=180.0

```

Шаг 6. Анализ. Видно, что raw-сокеты уступают TCP по абсолютному минимуму на единицы микросекунд (вклад нашего sсaру-обработчика в python), но многократно выигрывает по max и σ . Это и есть ключевое преимущество EtherCAT-подобного механизма: детерминированность – цена ниже среднего, но «хвост» распределения существенно короче. В вариантах

требуется повторить опыт при разной нагрузке, размере сообщения, приоритете процесса и сделать вывод о возможности применения каждого способа для заданного времени цикла.

Задания по вариантам

Номер варианта определяется как $(N - 1) \bmod 15 + 1$, где N – порядковый номер студента в журнале группы. Конкретные числовые параметры задачи берутся из приведённой ниже таблицы. Составить отчет со скриншотами, показывающие выполнение заданий.

Вариант 1.

- 31. Поднять veth-пару, запустить TCP-эхо-сервер и raw-эхо-slave; прогнать TCP- и raw-клиент по 1000 пакетов 64 байта без фоновой нагрузки.
- 32. Снять $\text{avg} / \text{min} / \text{max} / \sigma$ для обоих способов; сохранить массивы RTT в .pru.
- 33. Построить общую гистограмму (логарифмические оси, два цвета наложены) и подсчитать процентиля p_{50} , p_{95} , p_{99} .
- 34. В отчёт включить команды запуска, скрипты, сводную таблицу статистик и вывод о выигрыше raw по «хвосту» распределения.

Вариант 2.

- 32. Повторить эксперимент при размере сообщения 256 байт, $N = 1000$.
- 33. Сравнить avg и max с базовым результатом из варианта 1 (64 байта).
- 34. Подсчитать относительный выигрыш raw над TCP в обоих случаях; обосновать, как влияет рост размера.
- 35. В отчёт включить две таблицы статистики, отношение $\text{max_TCP} / \text{max_RAW}$ для двух размеров и вывод о масштабировании.

Вариант 3.

- 32. Размер сообщения 1024 байта, $N = 1000$.
- 33. Снять $\text{avg} / \text{max} / \sigma$ для TCP и raw.

34. Сравнить с вариантом 1; оценить, сохраняется ли преимущество gaw, и через какие компоненты задержки оно меняется (вклад копирования, MTU, TCP-сегментации).

35. В отчёт включить таблицу для трёх размеров (64, 256, 1024), график $\text{avg}(\text{size})$ для TCP и gaw, вывод об устойчивости детерминизма.

Вариант 4.

32. Запустить фоновую нагрузку `stress-ng --cpu N --vm 2` (N – число ядер); параллельно прогнать замер для TCP и gaw, размер 64 байта, N = 1000.

33. Зафиксировать avg / max / $p99$ для обоих способов.

34. Сравнить с базовыми (вариант 1) значениями; вычислить коэффициент роста max для TCP и gaw.

35. В отчёт включить две гистограммы (без нагрузки и под нагрузкой), таблицу сравнения и объяснение, почему TCP страдает сильнее.

Вариант 5.

31. Прогнать TCP-замер (64 байта, N = 1000) в двух режимах: клиент и сервер на одном ядре (без `taskset`) и на разных ядрах (`taskset -c 0` для сервера, `taskset -c 1` для клиента).

32. Сравнить avg / max / σ обоих режимов.

33. Объяснить полученную разницу через стоимость переключения контекста и межъядерный обмен данными через кэши.

34. В отчёт включить команды запуска, две таблицы статистики и вывод о пользе закрепления процессов в RT-системах.

Вариант 6.

33. Прогнать TCP-замер (64 байта, N = 1000) с включённым `TCP_NODELAY` и без него (`TCP_NODELAY = 0`; алгоритм Nagle активен).

34. Снять avg / max / σ для обоих случаев и построить гистограмму.

35. Особое внимание уделить max и появлению «бугра» в районе 40 мс (типичная задержка Nagle + delayed ACK).

36. В отчёт включить две гистограммы, таблицу avg / max, обоснование разницы через алгоритмы Nagle и delayed ACK.

Вариант 7.

33. Прогнать TCP- и raw-замер (64 байта, N = 1000) при обычном приоритете и под chrt -f 80 для обоих процессов.

34. Зафиксировать avg / max / p99 в четырёх комбинациях.

35. Подсчитать, на сколько процентов уменьшается max в SCHED_FIFO для каждого способа.

36. В отчёт включить сводную таблицу 2×2 (TCP / raw $\times$ обычный / RT), графики avg / max и вывод об эффективности RT-приоритета.

Вариант 8.

33. Реализовать UDP-эхо-сервер и клиент (то же loopback / veth).

34. Прогнать три способа: TCP, UDP, raw-сокеты по 1000 пакетов 64 байта при идентичных условиях.

35. Снять avg / min / max / σ / p99 по каждому способу.

36. Сравнить три распределения на одной гистограмме (лог-лог).

37. В отчёт включить код UDP-эхо, таблицу замеров для трёх протоколов, общий график и вывод о пригодности каждого для цикла управления заданной частоты.

Вариант 9.

32. Прогнать раз в 1 мс TCP-замер в течение 60 секунд (60 000 пакетов 64 байта).

33. Разбить замеры на окна по 1 секунде; в каждом окне посчитать avg и max.

34. Построить два графика «время – avg» и «время – max» с разрешением 1 секунда; выделить всплески.

35. Сопоставить всплески с системными событиями (cron, активность ОС в htop / atop).

36. В отчёт включить оба графика, таблицу глобальной статистики и вывод о долговременной стабильности.

Вариант 10.

33. Если доступно ядро с PREEMPT_RT – загрузить его и повторить базовый замер (TCP и raw, 64 байта, 1000 пакетов).

34. Если PREEMPT_RT недоступно – использовать связку `chrt -f 99 + taskset` + изоляция ядра через `isolcpus=` в `cmdline`.

35. Сравнить $avg / max / \sigma$ с замерами на обычном ядре (вариант 1).

36. Подсчитать улучшение max и σ .

37. В отчёт включить `uname -a` для обоих ядер, две таблицы статистики, гистограммы и вывод о цене реального времени.

Вариант 11.

33. Через `tc qdisc add dev veth_a root netem delay 100us 50us` ввести искусственную задержку с разбросом.

34. Повторить raw-замер (64 байта, 1000 пакетов) с включённым `netem`; снять $avg / max / \sigma$.

35. Сравнить с вариантом 1 (без `netem`); подсчитать вклад `netem` в avg и σ .

36. Снять `netem` (`tc qdisc del dev veth_a root`) и подтвердить, что статистика вернулась к базовой.

37. В отчёт включить команды `tc`, три набора замеров (без / с `netem` / снова без), две гистограммы и вывод о применимости `netem` для тестирования.

Вариант 12.

33. Прогнать TCP-замер (64 байта, 1000 пакетов) в двух топологиях: классический `loopback` (127.0.0.1) и через `veth`-пару (TCP-сервер слушает 10.0.0.2 на `veth_b`, клиент коннектится с 10.0.0.1 на `veth_a`).

34. Снять $avg / max / \sigma$ для каждого случая.

35. Сравнить и объяснить разницу через путь пакета: loopback обходит сетевой стек на уровне ядра, veth – проходит через полный сетевой стек дважды.

36. В отчёт включить ip-конфигурацию, две таблицы замеров, обоснование разницы.

Вариант 13.

33. Реализовать неблокирующий TCP-клиент (fcntl O_NONBLOCK + poll по событиям POLLIN / POLLOUT) с тем же RTT-замером.

34. Прогнать замер (64 байта, 1000 пакетов) для блокирующей и неблокирующей версий.

35. Снять avg / max / p99 для обоих случаев; обосновать разницу.

36. Объяснить, в каких задачах (один сокет / много сокетов) каждая версия выгоднее.

37. В отчёт включить код обоих клиентов, таблицу замеров, сценарии применимости.

Вариант 14.

33. Прогнать TCP- и raw-замер (64 байта) при $N = 100$, $N = 1000$ и $N = 10\,000$.

34. Для каждого N посчитать avg / σ / p99.

35. Оценить, как объём выборки влияет на стабильность оценки σ и p99 (доверительные интервалы).

36. Построить график $\sigma(N)$ и p99(N) для TCP и raw.

37. В отчёт включить таблицу замеров по трём N , графики сходимости статистик, вывод о минимально достаточном N для надёжного измерения «хвоста».

Вариант 15.

15. Подготовить две машины, соединённые прямым Ethernet-кабелем; настроить статические IP на обоих концах.

16. Запустить TSP- и raw-эхо-сервер на одной машине, клиент – на другой; прогнать по 1000 пакетов 64 байта.
17. Снять avg / max / σ для обоих способов.
18. Сравнить с loopback-результатами из варианта 1; оценить вклад физического канала (PHY + MAC + кабель).
19. В отчёт включить схему стенда, ip-конфигурацию, две таблицы (loopback и физический канал), вывод о минимальной задержке при выходе за пределы одной машины.

Лабораторная работа 9. Установка Fast DDS и создание Publisher/Subscriber

Цель работы: освоить базовые приёмы работы с middleware eProsima Fast DDS 3.x – установку библиотеки и генератора кода Fast DDS-Gen, описание типа сообщения на языке OMG IDL, генерацию сериализаторов, разработку процессов Publisher и Subscriber на C++17, запуск двух процессов на одном ПК в общем домене и проверку обмена сообщениями через автоматическое обнаружение участников по протоколу Simple Discovery Protocol.

Теоретическое обоснование

1. Назначение DDS и место Fast DDS в экосистеме. Data Distribution Service (DDS) – стандарт OMG для распределённого обмена данными по модели «издатель – подписчик» с типизированными топиками и управляемым качеством обслуживания (QoS). eProsima Fast DDS – открытая реализация спецификации DDS v1.4 и протокола RTPS v2.3, используемая по умолчанию в ROS 2 (Foxy, Humble, Iron, Jazzy). С 2024 года актуальной является ветка Fast DDS 3.x, в которой окончательно завершён переход на пространство имён `eprosima::fastdds::dds`, унифицированы заголовочные файлы в каталоге `fastdds/dds/` и удалены устаревшие синонимы из старого RTPS-API. Библиотека написана на C++17, распространяется по лицензии Apache 2.0 и поставляется вместе с генератором кода Fast DDS-Gen (исполняемый файл `fastddsgen`), который преобразует IDL-описание в сериализаторы и адаптер типа.

2. Глобальное пространство данных и сущности DDS. Концептуально DDS образует виртуальную «общую шину», в которой каждое сообщение привязано к именованному топику с фиксированным типом. Основные сущности перечислены в таблице 1.

Таблица 1 – Основные сущности модели DDS в Fast DDS 3.x

Сущность	Назначение
DomainParticipant	Корневой объект процесса в домене; владеет всеми Publisher, Subscriber и Topic. Создаётся через DomainParticipantFactory::get_instance()
Topic	Именованный канал данных фиксированного типа; связывает издателей и подписчиков
Publisher	Контейнер-фабрика для DataWriter в рамках одного DomainParticipant
Subscriber	Контейнер-фабрика для DataReader
DataWriter	Объект, через который приложение записывает экземпляры топика в общее пространство методом write()
DataReader	Объект, через который приложение получает экземпляры топика и уведомления об изменениях через listener
TypeSupport	Адаптер сериализации; регистрируется в DomainParticipant под именем типа из IDL

3. Идентификация участников и обнаружение. Каждый DomainParticipant принадлежит домену с целочисленным идентификатором domain_id (от 0 до 232). Участники одного домена автоматически находят друг друга через Simple Discovery Protocol (SDP): по умолчанию посылается multicast-объявление в группу 239.255.0.1 на порт 7400 + 250·domain_id. SDP состоит из двух фаз – Participant Discovery Phase (PDP, обмен GUID участников) и Endpoint Discovery Phase (EDP, обмен описаниями DataWriter и DataReader). После взаимного обнаружения DataWriter и DataReader, подписанные на один и тот же топик и обладающие совместимыми QoS, переходят в состояние matched и начинают обмен по протоколу RTPS поверх UDP. Общая схема взаимодействия двух процессов в одном домене приведена на рисунке 1.

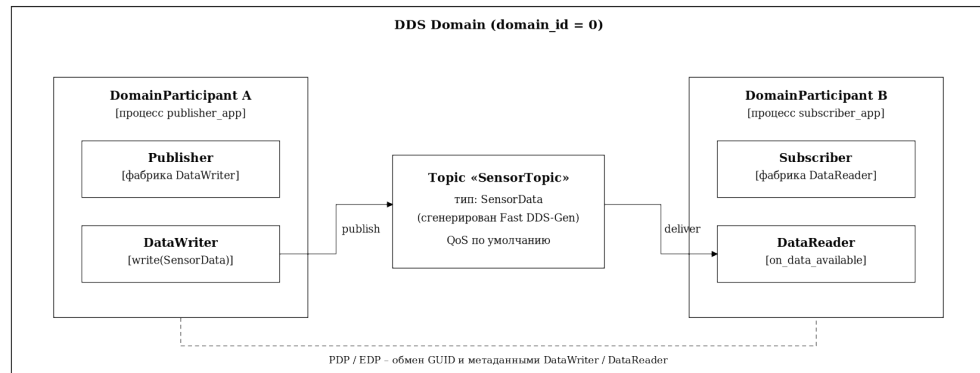


Рисунок 1 – Структура обмена «издатель – подписчик» в одном домене Fast DDS 3.x

4. Описание типа сообщения на IDL. Тип данных, передаваемых через топик, описывается в файле с расширением .idl на подмножестве языка OMG IDL. Поддерживаются примитивы (boolean, octet, short, long, float, double, string), массивы и последовательности переменной длины (sequence<T>), вложенные структуры. Утилита fastddsgen генерирует из IDL пять файлов: <Type>.hpp с объявлением C++-структуры, <Type>PubSubTypes.cxx/.hpp с реализацией TypeSupport, а также <Type>CdrAux.hpp/.ipp со вспомогательными функциями CDR-сериализации.

5. Жизненный цикл сущностей. Создание выполняется только через фабрики, освобождение – строго в обратном порядке: сначала DataWriter / DataReader, затем Publisher / Subscriber и Topic, последним удаляется DomainParticipant. Несоблюдение порядка приводит к утечкам внутренних потоков и блокировке порта. Для корректного освобождения принято хранить указатели в полях класса и удалять их в деструкторе через DomainParticipantFactory::get_instance()->delete_participant().

6. Сравнение DDS с очередями сообщений. DDS отличается от классических брокеров (MQTT, Kafka, RabbitMQ) типизацией топиков на уровне компилятора, отсутствием центрального брокера и наличием детального QoS. Краткое сопоставление приведено в таблице 2.

Таблица 2 – Fast DDS и брокерные системы обмена сообщениями

Свойство	Fast DDS 3.x	MQTT / Kafka
Архитектура	Peer-to-peer, без брокера	Централизованный брокер
Типизация	Строгая, из IDL на этапе компиляции	Слабая, payload – массив байт
Обнаружение	Автоматическое (SDP / multicast)	Через адрес брокера
QoS	22 политики (DEADLINE, LATENCY_BUDGET и др.)	QoS 0/1/2 (только надёжность)
Целевая задержка	Десятки – сотни микросекунд	Единицы – десятки миллисекунд

Пример выполнения задания

В качестве «нулевого» варианта рассматривается обмен сообщениями типа SensorData с двумя полями: целочисленным идентификатором датчика и значением температуры. Publisher отправляет сообщение раз в секунду, Subscriber выводит принятые значения в стандартный поток. Оба процесса работают на одном ПК с Ubuntu 24.04 в домене domain_id = 0.

Шаг 1. Установка Fast DDS 3.x. Сборка выполняется из домашнего каталога пользователя; зависимости устанавливаются из пакетов дистрибутива, после чего последовательно собираются foonthan_memory_vendor, Fast CDR и сама Fast DDS. Команды приведены в листинге 1. Для Ubuntu 24.04 eProsima также распространяет готовые debian-пакеты fastdds 3.x (репозиторий <https://www.eprosima.com/debian>), но для учебных целей предпочтительна сборка из исходных кодов – она даёт доступ к примерам и тестам.

Листинг 1 – Установка Fast DDS 3.x и Fast DDS-Gen в Ubuntu 24.04

```
sudo apt update
sudo apt install -y cmake g++ git python3-pip openjdk-17-jre \
    libasio-dev libtinyxml2-dev libssl-dev
```

```

mkdir -p ~/fastdds_ws && cd ~/fastdds_ws

git clone https://github.com/eProsima/foonathan_memory_vendor.git
cmake -S foonathan_memory_vendor -B build_fm \
  -DCMAKE_INSTALL_PREFIX=$HOME/fastdds_install
cmake --build build_fm --target install -j$(nproc)

git clone https://github.com/eProsima/Fast-CDR.git -b v2.2.5
cmake -S Fast-CDR -B build_cdr \
  -DCMAKE_INSTALL_PREFIX=$HOME/fastdds_install
cmake --build build_cdr --target install -j$(nproc)

git clone https://github.com/eProsima/Fast-DDS.git -b v3.1.0
cmake -S Fast-DDS -B build_dds \
  -DCMAKE_INSTALL_PREFIX=$HOME/fastdds_install \
  -DCMAKE_PREFIX_PATH=$HOME/fastdds_install
cmake --build build_dds --target install -j$(nproc)

# Генератор кода Fast DDS-Gen (Java-приложение)
git clone --recursive \
  https://github.com/eProsima/Fast-DDS-Gen.git
cd Fast-DDS-Gen && ./gradlew assemble
sudo ln -s $PWD/scripts/fastddsgen /usr/local/bin/fastddsgen

```

Шаг 2. Настройка окружения. Перед каждым сеансом работы необходимо добавить пути к установленным библиотекам и заголовкам в переменные окружения, иначе компоновщик не найдёт fastdds и fastcdr.

Листинг 2 – Подключение каталога установки Fast DDS к окружению

```

export FASTDDS=$HOME/fastdds_install
export CMAKE_PREFIX_PATH=$FASTDDS:$CMAKE_PREFIX_PATH
export LD_LIBRARY_PATH=$FASTDDS/lib:$LD_LIBRARY_PATH

```

Шаг 3. Описание типа сообщения. В рабочем каталоге проекта создаётся файл SensorData.idl. Структура содержит идентификатор датчика (32-битное целое) и текущее значение температуры (число с плавающей точкой одинарной точности).

Листинг 3 – Файл SensorData.idl с описанием типа сообщения

```

// SensorData.idl
struct SensorData
{
    long sensor_id;

```

```
float temperature;
};
```

Шаг 4. Генерация сериализаторов. Запуск `fastddsgen` с ключом `-replace` перезаписывает сгенерированные файлы; ключ `-cs (plain CDR)` опускается – по умолчанию генератор использует оптимизированную сериализацию XCDR v2 из стандарта DDS-XTypes.

Листинг 4 – Генерация TypeSupport и заголовков для SensorData

```
fastddsgen -replace SensorData.idl
# Появятся файлы:
# SensorData.hpp
# SensorDataPubSubTypes.cxx / .hpp
# SensorDataCdrAux.hpp / SensorDataCdrAux.ipp
# SensorDataTypeObjectSupport.cxx / .hpp
```

Шаг 5. Реализация Publisher. Класс инкапсулирует все сущности DDS, регистрирует тип `SensorData` в участнике и периодически вызывает `write()` в основном потоке. `Listener` используется для логирования момента обнаружения подписчика. Все заголовки – из пространства имён `eprosima::fastdds::dds` (Fast DDS 3.x).

Листинг 5 – Файл Publisher.cpp – издатель сообщений SensorData

```
#include <chrono>
#include <thread>
#include <iostream>
#include <fastdds/dds/domain/DomainParticipantFactory.hpp>
#include <fastdds/dds/domain/DomainParticipant.hpp>
#include <fastdds/dds/publisher/Publisher.hpp>
#include <fastdds/dds/publisher/Writer.hpp>
#include <fastdds/dds/publisher/WriterListener.hpp>
#include <fastdds/dds/topic/TypeSupport.hpp>
#include "SensorDataPubSubTypes.hpp"

using namespace eprosima::fastdds::dds;

class PubListener : public DataWriterListener
{
public:
    void on_publication_matched(
        DataWriter*, const PublicationMatchStatus& s) override
    {
        if (s.current_count_change == 1)
```

```

        std::cout << "Subscriber matched\n";
    else if (s.current_count_change == -1)
        std::cout << "Subscriber unmatched\n";
    }
};

int main()
{
    DomainParticipantQos pqos;
    pqos.name("publisher_app");
    auto* dp = DomainParticipantFactory::get_instance()
        ->create_participant(0, pqos);

    TypeSupport type(new SensorDataPubSubType());
    type.register_type(dp);

    auto* topic = dp->create_topic(
        "SensorTopic", type.get_type_name(), TOPIC_QOS_DEFAULT);
    auto* pub = dp->create_publisher(PUBLISHER_QOS_DEFAULT);

    PubListener listener;
    auto* writer = pub->create_datawriter(
        topic, DATAWRITER_QOS_DEFAULT, &listener);

    SensorData msg;
    msg.sensor_id(1);
    for (int i = 0; i < 20; ++i)
    {
        msg.temperature(20.0f + 0.1f * i);
        writer->write(&msg);
        std::cout << "sent T=" << msg.temperature() << "\n";
        std::this_thread::sleep_for(std::chrono::seconds(1));
    }

    pub->delete_datawriter(writer);
    dp->delete_publisher(pub);
    dp->delete_topic(topic);
    DomainParticipantFactory::get_instance()
        ->delete_participant(dp);
    return 0;
}

```

Шаг 6. Реализация Subscriber. Структура симметрична Publisher; вместо периодического write используется callback on_data_available, в котором DataReader забирает накопившиеся сэмплы методом take_next_sample. В Fast DDS 3.x возвращаемое значение методов API имеет тип ReturnCode_t; константы успеха – RETCODE_OK.

Листинг 6 – Файл Subscriber.cpp – подписчик сообщений SensorData

```
#include <iostream>
#include <thread>
#include <chrono>
#include <fastdds/dds/domain/DomainParticipantFactory.hpp>
#include <fastdds/dds/domain/DomainParticipant.hpp>
#include <fastdds/dds/subscriber/Subscriber.hpp>
#include <fastdds/dds/subscriber/DataReader.hpp>
#include <fastdds/dds/subscriber/DataReaderListener.hpp>
#include <fastdds/dds/subscriber/SampleInfo.hpp>
#include <fastdds/dds/topic/TypeSupport.hpp>
#include "SensorDataPubSubTypes.hpp"

using namespace eprosima::fastdds::dds;

class SubListener : public DataReaderListener
{
public:
    void on_subscription_matched(
        DataReader*, const SubscriptionMatchedException& s) override
    {
        if (s.current_count_change == 1)
            std::cout << "Publisher matched\n";
    }
    void on_data_available(DataReader* reader) override
    {
        SensorData msg;
        SampleInfo info;
        while (reader->take_next_sample(&msg, &info) == RETCODE_OK)
            if (info.valid_data)
                std::cout << "recv id=" << msg.sensor_id()
                    << " T=" << msg.temperature() << "\n";
    }
};

int main()
{
    DomainParticipantQos pqos;
    pqos.name("subscriber_app");
    auto* dp = DomainParticipantFactory::get_instance()
        ->create_participant(0, pqos);

    TypeSupport type(new SensorDataPubSubType());
    type.register_type(dp);

    auto* topic = dp->create_topic(
        "SensorTopic", type.get_type_name(), TOPIC_QOS_DEFAULT);
    auto* sub = dp->create_subscriber(SUBSCRIBER_QOS_DEFAULT);

    SubListener listener;
```

```

auto* reader = sub->create_datareader(
    topic, DATAREADER_QOS_DEFAULT, &listener);

std::this_thread::sleep_for(std::chrono::seconds(25));

sub->delete_datareader(reader);
dp->delete_subscriber(sub);
dp->delete_topic(topic);
DomainParticipantFactory::get_instance()
    ->delete_participant(dp);
return 0;
}

```

Шаг 7. Сборка проекта через CMake. Минимальный CMakeLists подключает пакеты fastcdr и fastdds, собирает два исполняемых файла из Publisher.cpp / Subscriber.cpp и сгенерированного SensorDataPubSubTypes.cxx. CMake 3.16+ требуется по двум причинам: поддержка target_link_libraries для INTERFACE-целей пакета fastdds и совместимость с C++17.

Листинг 7 – Файл CMakeLists.txt для сборки Publisher и Subscriber

```

cmake_minimum_required(VERSION 3.16)
project(lab9_dds CXX)
set(CMAKE_CXX_STANDARD 17)

find_package(fastcdr REQUIRED)
find_package(fastdds 3 REQUIRED)

set(TYPE_SRC SensorDataPubSubTypes.cxx
    SensorDataTypeObjectSupport.cxx)

add_executable(publisher_app Publisher.cpp ${TYPE_SRC})
target_link_libraries(publisher_app fastdds fastcdr)

add_executable(subscriber_app Subscriber.cpp ${TYPE_SRC})
target_link_libraries(subscriber_app fastdds fastcdr)

```

Шаг 8. Запуск и проверка. После сборки в build/ оба исполняемых файла запускаются в двух разных терминалах; порядок запуска несущественен – discovery срабатывает в любом случае. Ожидаемый вывод процессов приведён в листинге 8.

Листинг 8 – Совмещённый вывод publisher_app и subscriber_app

терминал 1

```

$ ./publisher_app
Subscriber matched
sent T=20
sent T=20.1
sent T=20.2
...

# терминал 2
$ ./subscriber_app
Publisher matched
recv id=1 T=20
recv id=1 T=20.1
recv id=1 T=20.2
...

```

Сообщения «matched» в обоих процессах подтверждают успешное взаимное обнаружение; совпадение значений температуры на стороне подписчика – корректность сериализации CDR. На этом базовый сценарий обмена считается выполненным.

Задания по вариантам

Номер варианта определяется как $(N - 1) \bmod 15 + 1$, где N – порядковый номер студента в журнале группы. Конкретные числовые параметры задачи берутся из приведённой ниже таблицы. Составить отчет со скриншотами, показывающие выполнение заданий.

Вариант 1.

40. Описать в IDL тип Temperature с полями long id, float value, string<8> unit. Сгенерировать TypeSupport через fastdds-gen.
41. Реализовать Publisher для топика «TempTopic», период 500 мс, 30 сообщений; значение value линейно растёт от 20 до 35.
42. Реализовать Subscriber с выводом id, value и unit в стандартный поток.
43. Сравнить размер сериализованного пакета (по логу netstat -s udp до и после) с вариантом, в котором поле unit удалено из IDL.

Вариант 2.

- 35. Описать тип GpsFix с полями double lat, double lon, double alt, unsigned long long timestamp_ns; сгенерировать TypeSupport.
- 36. Реализовать Publisher для топика «GpsTopic», период 200 мс. Координаты – произвольная траектория длиной 100 точек.
- 37. Реализовать Subscriber, который замеряет wall-clock задержку «write → on_data_available» по разнице между timestamp_ns в сообщении и текущим временем приёма по 100 сэмплам.
- 38. Подсчитать среднее, максимум и 95-перцентильную задержку по 100 измерениям.

Вариант 3.

- 36. Описать тип Imu с полями float ax, ay, az, gx, gy, gz.
- 37. Реализовать Publisher для топика «ImuTopic», период 10 мс, 1000 сообщений; значения – синус по времени с разными фазами.
- 38. Реализовать Subscriber, подсчитывающий число принятых сэмплов и длину сеанса.
- 39. Оценить пропускную способность в кб/с (объём CDR-сериализации × число сэмплов / время) и сравнить с теоретическим минимумом (размер payload × частоту).

Вариант 4.

- 36. Описать тип Command с полями long motor_id, long speed_rpm, boolean enable.
- 37. Реализовать Publisher для топика «MotorCmd», отправляющий сообщение по нажатию клавиши Enter в основном потоке (std::getline).
- 38. Реализовать Subscriber, логирующий все поступившие команды с номером сэмпла.
- 39. Подсчитать число сэмплов от Publisher за сеанс из 10 нажатий и сопоставить с логом Subscriber – подтвердить отсутствие потерь при QoS по умолчанию.

Вариант 5.

- 36. Описать тип Image с полями long seq, sequence<octet, 1024> data.
- 37. Реализовать Publisher для топика «ImgTopic», период 1 с, 20 сообщений; data заполнять случайными байтами через std::random_device.
- 38. Реализовать Subscriber, который при приёме сэмпла подсчитывает контрольную сумму data (XOR всех байт) и выводит её.
- 39. Подтвердить, что суммы на стороне Publisher и Subscriber совпадают по всем 20 сэмплам.

Вариант 6.

- 35. Описать тип Status с полями long node_id, string<32> state, long uptime_s.
- 36. Реализовать Publisher для топика «StatusTopic», период 2 с; state циклически принимает значения {«init», «run», «idle»}.
- 37. Реализовать Subscriber; запустить два экземпляра в разных терминалах.
- 38. Подтвердить, что оба Subscriber независимо получают одну и ту же последовательность сэмплов – проверить совпадение seq и state в их логах.

Вариант 7.

- 37. Описать тип Pose с полями float x, y, yaw.
- 38. Реализовать Publisher для топика «PoseTopic», период 50 мс; движение имитировать по окружности радиуса 1 м с угловой скоростью 0.1 рад/с.
- 39. Реализовать Subscriber, который выводит координаты с шагом 1 с (делает downsampling по счётчику).
- 40. Построить таблицу из 20 строк (один раз в секунду): t, x, y, yaw – и убедиться, что точки лежат на окружности.

Вариант 8.

- 37. Описать тип Lidar с полями float ranges[16], long stamp.
- 38. Реализовать Publisher для топика «LidarTopic», период 100 мс; значения ranges – 0.5–10.0 м с равномерным шумом ± 0.05 м.

39. Реализовать Subscriber, выводящий среднее значение по 16 лучам и значение stamp.
40. Подтвердить за сеанс длительностью 10 с, что число принятых сэмплов ≈ 100 (допустимое отклонение ± 2).

Вариант 9.

38. Описать тип Battery с полями float voltage, current, octet level_percent.
39. Реализовать Publisher для топика «BatteryTopic», период 1 с; level_percent монотонно убывает от 100 до 0, voltage линейно связан с уровнем (12.6 $\rightarrow$ 10.5 В).
40. Реализовать Subscriber, печатающий все поля с двумя знаками после запятой.
41. Завершить Publisher автоматически при достижении нуля и подтвердить, что Subscriber принял ровно 101 сэмпл.

Вариант 10.

37. Описать тип Event с полями long code, string<64> message, long long timestamp.
38. Реализовать Publisher для топика «EventTopic», публикующий случайно с интервалом 0.5–3 с (uniform); code – случайное число от 0 до 200.
39. Реализовать Subscriber, который в обработчике on_data_available фильтрует сэмплы по code > 100 и печатает только их.
40. За сеанс длительностью 60 с подсчитать долю отфильтрованных сэмплов и сравнить с теоретическим значением 0.5.

Вариант 11.

38. Описать тип Temperature2 с полями long id, float value.
39. Реализовать два Publisher для одного топика «TempTopic», id = 1 и id = 2, период 500 мс.
40. Реализовать Subscriber, ведущий два счётчика – число сэмплов от id = 1 и от id = 2.

41. После 30 с подтвердить, что счётчики примерно равны (отношение 0.9–1.1).

Вариант 12.

38. Описать тип Joint с полями long joint_id, double position, double velocity.

39. Реализовать Publisher для топика «JointState», период 20 мс; в одной итерации цикла отправлять три сообщения подряд для joint_id = 1, 2, 3.

40. position – синус по времени с фазами 0, $2\pi/3$, $4\pi/3$; velocity – численная производная по предыдущему значению.

41. Реализовать Subscriber, который выводит position только для joint_id = 2 (фильтрация в обработчике).

Вариант 13.

37. Описать тип PingMsg с полями long seq, long long send_ns.

38. Реализовать Publisher для топика «PingTopic», период 100 мс, 100 сообщений; send_ns – steady_clock на момент write().

39. Реализовать Subscriber, который при приёме вычитает send_ns из текущего steady_clock и сохраняет разницу в std::vector.

40. После сеанса напечатать среднюю и максимальную задержку – оценка односторонней задержки на одном ПК.

Вариант 14.

38. Описать тип ConfigUpdate с полями string<32> key, string<64> value, long version.

39. Реализовать Publisher для топика «ConfigTopic», публикующий сэмпл по вводу строки key=value в std::getline; version автоинкрементируется.

40. Реализовать Subscriber, поддерживающий std::map<string, pair<string, long>> – последнюю версию каждого ключа.

41. После 10 публикаций по 3 ключам вывести содержимое словаря и подтвердить, что версии монотонно растут.

Вариант 15.

38. Описать тип Telemetry с полями long device_id, float values[8].
39. Реализовать Publisher для топика «TelemetryTopic», период 250 мс; $values[i] = i \cdot 0.1$.
40. Реализовать Subscriber, проверяющий что values[0..7] на приёме ровно равны 0, 0.1, 0.2, ..., 0.7 (с допуском $1e-6$).
41. За 20 сэмплов подтвердить отсутствие искажений порядка байт – вывести сводку «ok / fail» и количество ошибок.

Лабораторная работа 10. Настройка QoS для гарантированной доставки (RELIABLE)

Цель работы: освоить настройку политики качества обслуживания RELIABILITY в Fast DDS 3.x, экспериментально сравнить поведение режимов BEST_EFFORT и RELIABLE при искусственно вносимых потерях пакетов на петлевом интерфейсе, измерить процент доставленных сообщений и понять механизм повторных передач Heartbeat / ACKNACK, на котором основан режим RELIABLE.

Теоретическое обоснование

1. Понятие качества обслуживания в DDS. Политики QoS определяют контракт между DataWriter и DataReader, регулирующий доставку данных: надёжность, упорядоченность, время жизни выборки, максимальный интервал между сообщениями и так далее. Все политики проверяются на совместимость на этапе обнаружения: если QoS сторон несовместим, соединение не устанавливается и в соответствующих listener-ах срабатывают коллбеки `on_offered_incompatible_qos` / `on_requested_incompatible_qos`. В Fast DDS 3.x все политики находятся в пространстве имён `eprosima::fastdds::dds`, а их объекты доступны как поля QoS-структур `DataWriterQos` и `DataReaderQos`.

2. Политика RELIABILITY. Политика управляет тем, что произойдёт с сэмплом, потерянным на сетевом уровне. Допустимы два значения, перечисленных в таблице 1.

Таблица 1 – Значения политики RELIABILITY

Значение	Поведение
BEST_EFFORT_RELIABILITY_QOS	DataWriter отправляет каждый сэмпл один раз и не отслеживает доставку. Потерянные сэмплы не восстанавливаются. Минимальные накладные расходы; пригоден для непрерывных потоков с допустимыми пропусками (телеметрия, видео, периодические позы).
RELIABLE_RELIABILITY_QOS	DataWriter хранит каждый отправленный

Значение	Поведение
OS	сэмпл в кэше до получения подтверждения от всех совместимых DataReader. При потере или повреждении сэмпла включается механизм Heartbeat / ACKNACK протокола RTPS, выполняется повторная передача. Гарантирует доставку, но увеличивает задержку и память.

3. Правило совместимости. Сторона, запрашивающая более слабую гарантию, совместима с более сильной, но не наоборот. Конкретные сочетания DataWriter (offered) и DataReader (requested) приведены в таблице 2.

Таблица 2 – Совместимость значений RELIABILITY

DataWriter (offered)	DataReader (requested)	Соединение
BEST_EFFORT	BEST_EFFORT	Устанавливается
BEST_EFFORT	RELIABLE	Несовместимо (RxO)
RELIABLE	BEST_EFFORT	Устанавливается
RELIABLE	RELIABLE	Устанавливается (с ARQ)

4. Механизм повторной передачи в режиме RELIABLE. В режиме RELIABLE Fast DDS поверх UDP реализует протокол подтверждений ARQ. DataWriter периодически рассылает сообщение Heartbeat со списком sequence-номеров, имеющих в его кэше. DataReader сравнивает список с принятыми сэмплами и в ответ посылает ACKNACK с перечнем пропущенных номеров. DataWriter повторно отправляет запрошенные сэмплы. Цикл повторяется до тех пор, пока кэш DataReader не подтвердит все номера или пока сэмпл не покинет историю DataWriter по политике HISTORY.

5. Взаимодействие с политикой HISTORY. RELIABILITY работает вместе с политикой HISTORY, ограничивающей объём кэша. При

KEEP_LAST с depth = N DataWriter хранит не более N последних сэмплов; при KEEP_ALL – все сэмплы, пока хватает памяти. Если DataReader временно недоступен и кэш RELIABLE-DataWriter переполняется, метод write() блокируется на max_blocking_time (значение задаётся через wqos.reliability().max_blocking_time). По истечении тайм-аута возвращается RETCODE_TIMEOUT, и сэмпл считается потерянным даже в режиме RELIABLE.

6. Способы симуляции потерь на петлевом интерфейсе. Для проверки поведения двух режимов необходимо искусственно ввести потери пакетов. На Linux это удобно делать через netem – модуль планировщика очередей tc, способный отбрасывать заданный процент пакетов на интерфейсе. Основные команды сведены в таблицу 3.

Таблица 3 – Команды netem для эмуляции потерь на интерфейсе lo

Команда	Действие
sudo tc qdisc add dev lo root netem loss 20%	Включить потерю 20 % исходящих пакетов на lo
sudo tc qdisc change dev lo root netem loss 50%	Изменить процент потерь без снятия дисциплины
sudo tc qdisc add dev lo root netem delay 5ms loss 10%	Дополнительно ввести задержку 5 мс
sudo tc qdisc del dev lo root	Снять все искусственные искажения
tc -s qdisc show dev lo	Показать накопленную статистику (sent, dropped)

7. Метрики оценки. Для количественного сравнения двух режимов достаточно фиксировать на стороне Subscriber два счётчика: число принятых сэмплов M и число обнаруженных пропусков в sequence-номерах L. Отношение $R = M / (M + L)$ показывает фактический процент доставки. Для RELIABLE при умеренных потерях ожидается $R = 1.0$ при росте средней задержки; для BEST_EFFORT R приблизительно равно $(1 - p)$, где p – заданный netem-ом процент потерь. Важно: при тестах с SHM-транспортом

netem не действует – перед опытом нужно отключить SHM (см. шаг 4 примера).

Пример выполнения задания

В качестве «нулевого» варианта используется обмен сообщениями типа Counter с одним полем – монотонно возрастающим номером. Publisher отправляет 200 сообщений с периодом 50 мс. Subscriber по полю seq восстанавливает порядок и считает пропуски. Опыт повторяется четыре раза: BEST_EFFORT без потерь, BEST_EFFORT с потерей 20 %, RELIABLE без потерь, RELIABLE с потерей 20 %.

Шаг 1. Описание типа Counter в IDL и генерация TypeSupport.

Листинг 1 – Файл Counter.idl

```
// Counter.idl
struct Counter
{
    long seq;
};
```

Листинг 2 – Генерация сериализаторов

```
fastddsgen -replace Counter.idl
```

Шаг 2. Publisher с выбираемой политикой RELIABILITY. Режим задаётся первым аргументом командной строки – строкой best_effort или reliable. После настройки QoS создаётся DataWriter и выполняется отправка 200 сэмплов с периодом 50 мс. Для воздействия netem на трафик принудительно отключается SHM-транспорт – оставляется только UDPv4.

Листинг 3 – Файл PublisherQos.cpp

```
#include <chrono>
#include <thread>
#include <string>
#include <iostream>
#include <fastdds/dds/domain/DomainParticipantFactory.hpp>
#include <fastdds/dds/domain/DomainParticipant.hpp>
#include <fastdds/dds/publisher/Publisher.hpp>
#include <fastdds/dds/publisher/DataWriter.hpp>
```

```

#include <fastdds/dds/topic/TypeSupport.hpp>
#include <fastdds/rtps/transport/UDPV4TransportDescriptor.hpp>
#include "CounterPubSubTypes.hpp"

using namespace eprosima::fastdds::dds;
using namespace eprosima::fastdds::rtps;

int main(int argc, char* argv[])
{
    std::string mode = (argc > 1) ? argv[1] : "best_effort";

    DomainParticipantQos pqos;
    pqos.transport().use_builtin_transports = false;
    pqos.transport().user_transports.push_back(
        std::make_shared<UDPV4TransportDescriptor>());

    auto* dp = DomainParticipantFactory::get_instance()
        ->create_participant(0, pqos);
    TypeSupport type(new CounterPubSubType());
    type.register_type(dp);

    auto* topic = dp->create_topic(
        "CounterTopic", type.get_type_name(), TOPIC_QOS_DEFAULT);
    auto* pub = dp->create_publisher(PUBLISHER_QOS_DEFAULT);

    DataWriterQos wqos = DATAWRITER_QOS_DEFAULT;
    if (mode == "reliable")
        wqos.reliability().kind = RELIABLE_RELIABILITY_QOS;
    else
        wqos.reliability().kind = BEST_EFFORT_RELIABILITY_QOS;
    wqos.history().kind = KEEP_ALL_HISTORY_QOS;

    auto* writer = pub->create_datawriter(topic, wqos);

    // ждём появления подписчика
    std::this_thread::sleep_for(std::chrono::seconds(2));

    Counter msg;
    for (int i = 1; i <= 200; ++i)
    {
        msg.seq(i);
        writer->write(&msg);
        std::this_thread::sleep_for(std::chrono::milliseconds(50));
    }
    std::cout << "sent 200 samples (" << mode << ")\n";

    std::this_thread::sleep_for(std::chrono::seconds(2));
    pub->delete_datawriter(writer);
    dp->delete_publisher(pub);
    dp->delete_topic(topic);
}

```

```

DomainParticipantFactory::get_instance()
->delete_participant(dp);
return 0;
}

```

Шаг 3. Subscriber с подсчётом пропусков. Listener фиксирует принятые номера; пропуски определяются по разрывам в последовательности. Параметр RELIABILITY должен совпадать с режимом Publisher либо быть «слабее».

Листинг 4 – Файл SubscriberQos.cpp

```

#include <chrono>
#include <thread>
#include <string>
#include <iostream>
#include <fastdds/dds/domain/DomainParticipantFactory.hpp>
#include <fastdds/dds/domain/DomainParticipant.hpp>
#include <fastdds/dds/subscriber/Subscriber.hpp>
#include <fastdds/dds/subscriber/DataReader.hpp>
#include <fastdds/dds/subscriber/DataReaderListener.hpp>
#include <fastdds/dds/subscriber/SampleInfo.hpp>
#include <fastdds/dds/topic/TypeSupport.hpp>
#include <fastdds/rtps/transport/UDPV4TransportDescriptor.hpp>
#include "CounterPubSubTypes.hpp"

using namespace eprosima::fastdds::dds;
using namespace eprosima::fastdds::rtps;

class CntListener : public DataReaderListener
{
public:
    int received = 0;
    int gaps = 0;
    int last = 0;
    void on_data_available(DataReader* r) override
    {
        Counter msg;
        SampleInfo info;
        while (r->take_next_sample(&msg, &info) == RETCODE_OK)
        {
            if (!info.valid_data) continue;
            ++received;
            if (last != 0 && msg.seq() != last + 1)
                gaps += msg.seq() - last - 1;
            last = msg.seq();
        }
    }
};

```

```

int main(int argc, char* argv[])
{
    std::string mode = (argc > 1) ? argv[1] : "best_effort";

    DomainParticipantQos pqos;
    pqos.transport().use_builtin_transports = false;
    pqos.transport().user_transports.push_back(
        std::make_shared<UDPV4TransportDescriptor>());

    auto* dp = DomainParticipantFactory::get_instance()
        ->create_participant(0, pqos);
    TypeSupport type(new CounterPubSubType());
    type.register_type(dp);

    auto* topic = dp->create_topic(
        "CounterTopic", type.get_type_name(), TOPIC_QOS_DEFAULT);
    auto* sub = dp->create_subscriber(SUBSCRIBER_QOS_DEFAULT);

    DataReaderQos rqos = DATAREADER_QOS_DEFAULT;
    if (mode == "reliable")
        rqos.reliability().kind = RELIABLE_RELIABILITY_QOS;
    else
        rqos.reliability().kind = BEST_EFFORT_RELIABILITY_QOS;
    rqos.history().kind = KEEP_ALL_HISTORY_QOS;

    CntListener listener;
    auto* reader = sub->create_datareader(topic, rqos, &listener);

    std::this_thread::sleep_for(std::chrono::seconds(15));

    int total = listener.received + listener.gaps;
    double rate = total ? 100.0 * listener.received / total : 0.0;
    std::cout << mode << ": received=" << listener.received
        << " gaps=" << listener.gaps
        << " rate=" << rate << "%\n";

    sub->delete_datareader(reader);
    dp->delete_subscriber(sub);
    dp->delete_topic(topic);
    DomainParticipantFactory::get_instance()
        ->delete_participant(dp);
    return 0;
}

```

Шаг 4. Сценарий измерений. Опыт повторяется четыре раза по схеме «эталон без потерь – такой же режим с потерями 20 %»; между запусками

очередь netem пересоздаётся, чтобы обнулить статистику. Команды последовательно вводятся в третьем терминале.

Листинг 5 – Сценарий запуска четырёх опытов

```
# (1) BEST_EFFORT без потерь
sudo tc qdisc del dev lo root 2>/dev/null
./subscriber_qos best_effort &
./publisher_qos best_effort
wait

# (2) BEST_EFFORT с потерей 20 %
sudo tc qdisc add dev lo root netem loss 20%
./subscriber_qos best_effort &
./publisher_qos best_effort
wait

# (3) RELIABLE без потерь
sudo tc qdisc del dev lo root
./subscriber_qos reliable &
./publisher_qos reliable
wait

# (4) RELIABLE с потерей 20 %
sudo tc qdisc add dev lo root netem loss 20%
./subscriber_qos reliable &
./publisher_qos reliable
wait

sudo tc qdisc del dev lo root
```

Шаг 5. Ожидаемые результаты. Сводка типичного вывода Subscriber приведена в листинге 5; точные значения зависят от ядра и нагрузки, но качественная картина воспроизводится стабильно.

Листинг 6 – Сводка вывода Subscriber по четырём опытам

```
best_effort: received=200 gaps=0 rate=100%
best_effort: received=160 gaps=40 rate=80%
reliable: received=200 gaps=0 rate=100%
reliable: received=200 gaps=0 rate=100%
```

Шаг 6. Выводы. BEST_EFFORT при потере 20 % исходящих пакетов теряет около 20 % сэмплов – согласуется с биномиальной моделью. RELIABLE при тех же потерях восстанавливает все сэмплы за счёт повторных передач, что подтверждает rate = 100 %. Цена – рост трафика на

интерфейсе (его можно увидеть командой «tc -s qdisc show dev lo» – число backlog-пакетов и retransmit), а также рост средней задержки доставки запоздавших сэмплов.

Задания по вариантам

Номер варианта определяется как $(N - 1) \bmod 15 + 1$, где N – порядковый номер студента в журнале группы. Конкретные числовые параметры задачи берутся из приведённой ниже таблицы. Составить отчет со скриншотами, показывающие выполнение заданий.

Вариант 1.

- 44. Тип Counter {long seq;}. Period = 20 мс, N = 500. Для обоих режимов прогнать опыты с потерями 10 %, 30 %, 50 %.
- 45. Собрать таблицу rate (3 уровня потерь × 2 режима = 6 строк).
- 46. Построить график rate(p) для BEST_EFFORT и RELIABLE на одной координатной плоскости.
- 47. Объяснить, почему график RELIABLE остаётся на 100 % до некоторого критического значения p.

Вариант 2.

- 39. Тип Telemetry {long seq; float value;}. Period 100 мс, N = 100. Потери 25 %.
- 40. Для RELIABLE дополнительно измерить wall-clock задержку приёма по разнице между текущим временем и моментом write() (передаётся в поле value через приведение к секундам steady_clock).
- 41. Получить распределение задержек по 100 сэмплам: среднее, максимум, 95-й перцентиль.
- 42. Сравнить с тем же показателем при потерях 0 % – оценить оверхед повторных передач.

Вариант 3.

- 40. Тип BurstMsg {long seq; octet payload[256];}. Period 10 мс, N = 1000. Потери 15 %.

- 41. Реализовать оба режима; в Subscriber фиксировать момент первого и последнего принятого сэмпла.
- 42. Подсчитать фактическую пропускную способность в Кб/с: размер одного сэмпла $\times$ число принятых / длительность приёма.
- 43. Сравнить пропускную способность в режимах BEST_EFFORT и RELIABLE; объяснить наблюдаемую разницу.

Вариант 4.

- 40. Тип Counter. Period 50 мс, N = 200.
- 41. Создать несовместимую пару: Publisher настроен на BEST_EFFORT, Subscriber – на RELIABLE.
- 42. Реализовать в обоих процессах обработчики on_offered_incompatible_qos и on_requested_incompatible_qos, печатать total_count и поле last_policy_id.
- 43. Подтвердить, что matched-связь не устанавливается и ни одного сэмпла не доставлено.

Вариант 5.

- 40. Тип Counter. Period 50 мс, N = 200, RELIABLE.
- 41. Ввести задержку netem delay 100ms без потерь.
- 42. Замерить общее время сеанса от первого write() до последнего on_data_available.
- 43. Сравнить с теоретическим временем $200 \times 50 \text{ мс} = 10 \text{ с}$ и оценить, насколько netem-овская задержка увеличила сеанс.

Вариант 6.

- 39. Тип Counter. Period 30 мс, N = 300. Потери 40 %, RELIABLE.
- 40. Опыт 1: HISTORY = KEEP_ALL. Опыт 2: HISTORY = KEEP_LAST с depth = 10.
- 41. В обоих опытах подсчитать rate и число пропусков, зафиксированных Subscriber-ом.

42. Объяснить, почему при `depth = 10` даже `RELIABLE` может потерять сэмплы.

Вариант 7.

41. Тип Counter. Period 100 мс, `N = 100`. `RELIABLE`.

42. Запустить Subscriber через 5 с после Publisher.

43. Опыт 1: `DURABILITY = VOLATILE` (по умолчанию). Опыт 2: `DURABILITY = TRANSIENT_LOCAL`.

44. Подтвердить, что в опыте 2 Subscriber получает сэмплы с `seq = 1`, а в опыте 1 – только с момента подключения.

Вариант 8.

41. Тип Counter. Period 50 мс, `N = 400`. Потери 20 %.

42. Запустить два Subscriber одновременно: один с `BEST_EFFORT`, второй с `RELIABLE`.

43. Publisher – `RELIABLE`, чтобы быть совместимым с обоими.

44. Сравнить `rate` и число пропусков у двух Subscriber в одном опыте; объяснить, почему `BEST_EFFORT`-Subscriber всё равно получает большую часть сэмплов.

Вариант 9.

42. Тип Counter. Period 5 мс, `N = 2000`. `RELIABLE`. Потери 50 %.

43. Задать `wqos.reliability().max_blocking_time = Duration_t{0, 50000000}` (50 мс).

44. В Publisher логировать каждый возврат `RETCODE_TIMEOUT` из `write()` и подсчитывать их общее число.

45. Подтвердить, что часть сэмплов теряется даже в `RELIABLE` из-за тайм-аута и переполнения кэша.

Вариант 10.

41. Тип Counter. Period 50 мс, `N = 200`. `RELIABLE`.

42. `netem delay 50ms ± 20ms (jitter)` без потерь.

43. В Subscriber передавать через поле seq метку времени write() и измерять задержку доставки.

44. Построить распределение задержек по 200 сэмплам (10 бинов от 30 до 80 мс).

Вариант 11.

42. Тип PosMsg {long seq; double x; double y;}. Period 25 мс, N = 400. Потери 30 %, BEST_EFFORT.

43. В Publisher генерировать траекторию по окружности радиуса 1 м.

44. В Subscriber сохранять принятые точки в файл траектории.

45. Визуально (по выводу файла) оценить, можно ли восстановить окружность из принятых точек при 30 % потерь.

Вариант 12.

42. Тип Counter. Period 50 мс, N = 200.

43. Запустить два Publisher (RELIABLE), один Subscriber (RELIABLE).

44. Publisher-ы пишут с разными значениями seq (нечётные и чётные).

45. Подтвердить, что rate = 100 % по обоим источникам; обосновать, как RTPS различает потоки от двух DataWriter на одном топики.

Вариант 13.

41. Тип Counter. Period 100 мс, N = 100. RELIABLE + HISTORY = KEEP_LAST depth = 1.

42. Включить netem delay 500ms (без потерь).

43. В Subscriber подсчитать число принятых сэмплов и обнаруженных пропусков.

44. Показать, что при большой задержке и depth = 1 даже RELIABLE теряет промежуточные сэмплы, поскольку кэш переписывается.

Вариант 14.

42. Тип Counter. Period 50 мс, N = 200. Оба режима.

43. Цикл из десяти процентов потерь: 5, 10, 15, ..., 50 % – по опыту на каждый.
44. Собрать таблицу $\text{rate}(p)$ для BEST_EFFORT и RELIABLE ($10 \times 2 = 20$ значений).
45. Сравнить с теоретическим $(1 - p)$ для BEST_EFFORT; оценить, где RELIABLE начинает терять сэмплы.

Вариант 15.

42. Тип Counter. Period 20 мс, $N = 500$. RELIABLE.
43. Сценарий с переменными потерями: 0 % – 5 с, 30 % – 5 с, 0 % – оставшееся время.
44. Менять netem командой `tc qdisc change` по расписанию из третьего терминала.
45. Подтвердить, что $\text{rate} = 100\%$ на всей дистанции; зафиксировать момент, когда кэш Publisher максимален (по логу).

Лабораторная работа 11. DDS с DEADLINE и LATENCY_BUDGET

Цель работы: освоить временные QoS-политики Fast DDS 3.x – DEADLINE (контракт на максимальный интервал между сообщениями) и LATENCY_BUDGET (допустимая задержка доставки), научиться обрабатывать соответствующие listener-коллбеки `on_offered_deadline_missed` и `on_requested_deadline_missed`, измерить фактическую задержку доставки и сделать выводы о применимости политик в системах реального времени.

Теоретическое обоснование

1. Временные QoS-политики DDS. Политики DEADLINE и LATENCY_BUDGET относятся к группе временных контрактов: они не управляют доставкой сэмплов напрямую, как RELIABILITY, а описывают временные ожидания приложения от DataWriter и DataReader. При нарушении ожиданий middleware уведомляет приложение через соответствующие listener-коллбеки, не прерывая обмен. Ответственность за реакцию (переключение источника, тревога, fallback) лежит на приложении. В Fast DDS 3.x обе политики и связанные структуры находятся в пространстве имён `eprosima::fastdds::dds`.

2. Политика DEADLINE. Политика задаёт максимально допустимый интервал между двумя последовательными обновлениями сэмпла одного экземпляра. Значение хранится в поле `period` структуры `DeadlineQosPolicy` типа `eprosima::fastdds::dds::Duration_t` (секунды и наносекунды). На стороне DataWriter таймер запускается каждый раз после успешного `write()`: если до истечения `period` не было следующего `write()`, вызывается коллбек `on_offered_deadline_missed`. На стороне DataReader таймер сбрасывается при поступлении сэмпла: если сэмпл не получен в течение `period`, вызывается `on_requested_deadline_missed`.

3. Совместимость DEADLINE. Контракт считается совместимым, если период DataWriter не превышает периода DataReader: $\text{offered} \leq \text{requested}$. При нарушении (например, DataWriter заявил 100 мс, а DataReader требует 50 мс)

соединение не устанавливается и срабатывают on_offered_incompatible_qos / on_requested_incompatible_qos.

4. Структура отчёта о пропуске. Коллбеки получают объект статуса со счётчиками, аналогичными другим коллбекам DDS. Поля статусов DEADLINE приведены в таблице 1.

Таблица 1 – Поля статусов OfferedDeadlineMissedStatus / RequestedDeadlineMissedStatus

Поле	Смысл
total_count	Накопленное число пропусков с момента создания DataWriter / DataReader
total_count_change	Прирост total_count с момента предыдущего вызова listener
last_instance_handle	Идентификатор экземпляра, по которому зафиксирован последний пропуск

5. Политика LATENCY_BUDGET. Политика задаёт целевую максимальную задержку между моментом write() на стороне DataWriter и моментом доставки сэмпла DataReader. В отличие от DEADLINE, нарушение LATENCY_BUDGET не вызывает специального коллбека: значение является «подсказкой» middleware о том, насколько агрессивно следует группировать сэмплы и насколько коротким должен быть Heartbeat-цикл. Значение по умолчанию – 0 нс, что означает «отправлять немедленно». Совместимость требует offered ≤ requested (DataWriter обещает уложиться в более жёсткий бюджет, чем требует DataReader, либо равный).

6. Измерение фактической задержки. Поскольку Fast DDS не предоставляет встроенного коллбека по LATENCY_BUDGET, для контроля используют поле source_timestamp в SampleInfo либо явное поле send_ns в самом сообщении. Разница между текущим временем приёма и источником даёт одностороннюю задержку. По 100–1000 измерениям строится распределение, из которого извлекаются среднее, максимум и 95-й

перцентиль. Сопоставление перцентиля с LATENCY_BUDGET позволяет оценить, выдерживается ли контракт на практике.

7. Сводка трёх временных QoS приведена в таблице 2.

Таблица 2 – Сравнение временных политик DEADLINE, LATENCY_BUDGET и LIFESPAN

Политика	Что задаёт	Действие при нарушении
DEADLINE	Макс. интервал между write() / приёмами одного экземпляра	Коллбек on*_deadline_missed; обмен продолжается
LATENCY_BUDGET	Целевая макс. задержка доставки сэмпла	Коллбека нет; подсказка планировщику RTPS
LIFESPAN	Срок действия сэмпла после write()	Просроченный сэмпл удаляется из кэша до доставки

8. Применение политик. DEADLINE применяют для мониторинга активности источников (потеря heartbeat-сэмпла от датчика, молчание управляющего узла), LATENCY_BUDGET – для согласования ожиданий контура управления (например, для контура с шагом квантования 10 мс задают LATENCY_BUDGET = 2 мс). Совместное использование с RELIABLE недопустимо, если максимальное время повторной передачи превышает DEADLINE – контракт будет регулярно нарушаться.

Пример выполнения задания

В качестве «нулевого» варианта рассматривается DataWriter, обязующийся выдавать сэмплы типа Beacon не реже одного раза в 200 мс, с целевой задержкой доставки LATENCY_BUDGET = 5 мс. Publisher работает в двух режимах: «нормальный» – с периодом 100 мс (укладывается в дедлайн) и «деградировавший» – с периодом 300 мс (нарушает дедлайн). Subscriber фиксирует обе ситуации и измеряет фактическую задержку доставки.

Шаг 1. Описание типа Beacon в IDL и генерация TypeSupport.

Листинг 1 – Файл Beacon.idl

```
// Beacon.idl
struct Beacon
{
    long        seq;
    unsigned long long send_ns;
};
```

Листинг 2 – Генерация

```
fastddsgen -replace Beacon.idl
```

Шаг 2. Publisher с переменным периодом и заданным DEADLINE. Период задаётся первым аргументом командной строки в миллисекундах; DEADLINE фиксирован равным 200 мс. Все типы Duration_t берутся из eprosima::fastdds::dds (Fast DDS 3.x).

Листинг 3 – Файл PublisherDeadline.cpp

```
#include <chrono>
#include <thread>
#include <iostream>
#include <fastdds/dds/domain/DomainParticipantFactory.hpp>
#include <fastdds/dds/domain/DomainParticipant.hpp>
#include <fastdds/dds/publisher/Publisher.hpp>
#include <fastdds/dds/publisher/DataWriter.hpp>
#include <fastdds/dds/publisher/DataWriterListener.hpp>
#include <fastdds/dds/topic/TypeSupport.hpp>
#include "BeaconPubSubTypes.hpp"

using namespace eprosima::fastdds::dds;

class PubL : public DataWriterListener
{
public:
    void on_offered_deadline_missed(
        DataWriter*,
        const OfferedDeadlineMissedStatus& s) override
    {
        std::cout << "[pub] offered_deadline_missed total="
            << s.total_count
            << " " << s.total_count_change << "\n";
    }
};
```

```

int main(int argc, char* argv[])
{
    int period_ms = (argc > 1) ? std::stoi(argv[1]) : 100;

    auto* dp = DomainParticipantFactory::get_instance()
        ->create_participant(0, PARTICIPANT_QOS_DEFAULT);
    TypeSupport type(new BeaconPubSubType());
    type.register_type(dp);
    auto* topic = dp->create_topic(
        "BeaconTopic", type.get_type_name(), TOPIC_QOS_DEFAULT);
    auto* pub = dp->create_publisher(PUBLISHER_QOS_DEFAULT);

    DataWriterQos wqos = DATAWRITER_QOS_DEFAULT;
    wqos.reliability().kind = RELIABLE_RELIABILITY_QOS;
    wqos.deadline().period = Duration_t(0, 200000000); // 200 мс
    wqos.latency_budget().duration = Duration_t(0, 5000000); // 5 мс

    PubL listener;
    auto* writer = pub->create_datawriter(topic, wqos, &listener);
    std::this_thread::sleep_for(std::chrono::seconds(2));

    Beacon msg;
    using namespace std::chrono;
    for (int i = 1; i <= 60; ++i)
    {
        msg.seq(i);
        auto t = duration_cast<nanoseconds>(
            steady_clock::now().time_since_epoch()).count();
        msg.send_ns(static_cast<unsigned long long>(t));
        writer->write(&msg);
        std::this_thread::sleep_for(milliseconds(period_ms));
    }

    std::this_thread::sleep_for(std::chrono::seconds(1));
    pub->delete_datawriter(writer);
    dp->delete_publisher(pub);
    dp->delete_topic(topic);
    DomainParticipantFactory::get_instance()
        ->delete_participant(dp);
    return 0;
}

```

Шаг 3. Subscriber с теми же контрактами и измерением задержки. Listener реализует `on_requested_deadline_missed` и подсчитывает среднюю, максимальную и 95-перцентильную задержку по полю `send_ns`.

Листинг 4 – Файл `SubscriberDeadline.cpp`

```
#include <algorithm>
```

```

#include <chrono>
#include <thread>
#include <vector>
#include <iostream>
#include <fastdds/dds/domain/DomainParticipantFactory.hpp>
#include <fastdds/dds/domain/DomainParticipant.hpp>
#include <fastdds/dds/subscriber/Subscriber.hpp>
#include <fastdds/dds/subscriber/DataReader.hpp>
#include <fastdds/dds/subscriber/DataReaderListener.hpp>
#include <fastdds/dds/subscriber/SampleInfo.hpp>
#include <fastdds/dds/topic/TypeSupport.hpp>
#include "BeaconPubSubTypes.hpp"

using namespace eprosima::fastdds::dds;

class SubL : public DataReaderListener
{
public:
    std::vector<long long> latencies_ns;
    void on_requested_deadline_missed(
        DataReader*,
        const RequestedDeadlineMissedStatus& s) override
    {
        std::cout << "[sub] requested_deadline_missed total="
            << s.total_count
            << " " << s.total_count_change << "\n";
    }
    void on_data_available(DataReader* r) override
    {
        Beacon msg;
        SampleInfo info;
        using namespace std::chrono;
        while (r->take_next_sample(&msg, &info) == RETCODE_OK)
        {
            if (!info.valid_data) continue;
            auto now_ns = duration_cast<nanoseconds>(
                steady_clock::now().time_since_epoch()).count();
            latencies_ns.push_back(
                now_ns - static_cast<long long>(msg.send_ns()));
        }
    }
};

int main()
{
    auto* dp = DomainParticipantFactory::get_instance()
        ->create_participant(0, PARTICIPANT_QOS_DEFAULT);
    TypeSupport type(new BeaconPubSubType());
    type.register_type(dp);
    auto* topic = dp->create_topic(
        "BeaconTopic", type.get_type_name(), TOPIC_QOS_DEFAULT);
    auto* sub = dp->create_subscriber(SUBSCRIBER_QOS_DEFAULT);

```

```

DataReaderQos rqos = DATAREADER_QOS_DEFAULT;
rqos.reliability().kind      = RELIABLE_RELIABILITY_QOS;
rqos.deadline().period      = Duration_t(0, 200000000);
rqos.latency_budget().duration = Duration_t(0, 5000000);

SubL listener;
auto* reader = sub->create_datareader(topic, rqos, &listener);

std::this_thread::sleep_for(std::chrono::seconds(25));

auto& v = listener.latencies_ns;
if (!v.empty())
{
    std::sort(v.begin(), v.end());
    long long sum = 0;
    for (auto x : v) sum += x;
    long long avg = sum / static_cast<long long>(v.size());
    long long p95 = v[static_cast<size_t>(v.size() * 0.95)];
    long long mx = v.back();
    std::cout << "received=" << v.size()
                << " avg_us=" << avg / 1000
                << " p95_us=" << p95 / 1000
                << " max_us=" << mx / 1000 << "\n";
}

sub->delete_datareader(reader);
dp->delete_subscriber(sub);
dp->delete_topic(topic);
DomainParticipantFactory::get_instance()
    ->delete_participant(dp);
return 0;
}

```

Шаг 4. Опыт 1 – нормальный режим. Publisher запускается с периодом 100 мс; ни один интервал не превышает заявленный DEADLINE = 200 мс. Коллбеки on_*_deadline_missed не вызываются. Ожидаемый вывод приведён в листинге 4.

Листинг 5 – Опыт 1: period_ms = 100 (норма)

```

$ ./publisher_deadline 100
(коллбеков нет)

```

```

$ ./subscriber_deadline
received=60 avg_us=180 p95_us=320 max_us=540

```

Шаг 5. Опыт 2 – нарушение DEADLINE. Publisher запускается с периодом 300 мс. После каждого write() таймер DataWriter истекает раньше следующего write() – вызывается on_offered_deadline_missed; со стороны Subscriber аналогично вызывается on_requested_deadline_missed. Ожидаемый вывод приведён в листинге 5.

Листинг 6 – Опыт 2: period_ms = 300 (нарушение)

```
$ ./publisher_deadline 300
[pub] offered_deadline_missed total=1 +1
[pub] offered_deadline_missed total=2 +1
...
[pub] offered_deadline_missed total=59 +1

$ ./subscriber_deadline
[sub] requested_deadline_missed total=1 +1
[sub] requested_deadline_missed total=2 +1
...
received=60 avg_us=210 p95_us=380 max_us=620
```

Шаг 6. Анализ LATENCY_BUDGET. Заявленный бюджет – 5 мс (5000 мкс). В опыте 1 средняя задержка составила 180 мкс, 95-й перцентиль – 320 мкс, что втрое меньше бюджета – контракт выполнен с запасом. В опыте 2 распределение почти не изменилось (данные передаются по тому же петлевому интерфейсу), но формальный контракт DEADLINE нарушен на каждом периоде. Этот результат иллюстрирует независимость двух политик: маленькая задержка доставки не спасает от пропуска дедлайна, если источник просто молчит.

Шаг 7. Выводы. on_offered_deadline_missed позволяет приложению своевременно обнаружить «умолкшего» издателя, а измерение по send_ns – оценить, насколько фактическая задержка вписывается в LATENCY_BUDGET. В реальных системах две политики комбинируют: DEADLINE контролирует частоту обновлений (heartbeat от датчика), LATENCY_BUDGET – допустимый «возраст» данных в контуре управления.

Задания по вариантам

Номер варианта определяется как $(N - 1) \bmod 15 + 1$, где N – порядковый номер студента в журнале группы. Конкретные числовые параметры задачи берутся из приведённой ниже таблицы. Составить отчет со скриншотами, показывающие выполнение заданий.

Вариант 1.

48. Тип Beacon {long seq;}. DEADLINE = 100 мс, LATENCY_BUDGET = 2 мс.
49. Опыт 1: Publisher с периодом 50 мс (норма). Опыт 2: с периодом 150 мс (нарушение).
50. Зафиксировать число on_offered_deadline_missed и on_requested_deadline_missed в обоих опытах за 30 с.
51. Объяснить, почему счётчики на стороне Publisher и Subscriber совпадают.

Вариант 2.

43. Тип Heartbeat {long node_id;}. DEADLINE = 1 с, LATENCY_BUDGET = 50 мс.
44. Publisher отправляет сэмплы с периодом 200 мс. Со 2-й по 4-ю секунду – пауза без write().
45. Subscriber работает 8 с; зафиксировать число on_requested_deadline_missed.
46. Объяснить, почему число пропусков ровно 2 (1 раз за каждый истекший period в зоне молчания).

Вариант 3.

44. Тип Tick {long seq;}. DEADLINE = 20 мс, LATENCY_BUDGET = 1 мс. Period = 10 мс, $N = 1000$.
45. В Subscriber собирать задержки в `std::vector<long long>`.
46. После сеанса напечатать avg, p50, p95, p99, max задержки в микросекундах.

47. Сравнить p_{95} с заявленным LATENCY_BUDGET; оценить, какой процент сэмплов уложился в бюджет.

Вариант 4.

44. Тип Beacon. DataWriter DEADLINE = 200 мс, DataReader DEADLINE = 100 мс.

45. В обоих процессах реализовать on_offered_incompatible_qos / on_requested_incompatible_qos.

46. Подтвердить, что matched не наступает и ни один сэмпл не доставлен.

47. Изменить DataReader DEADLINE на 200 мс; убедиться, что соединение устанавливается.

Вариант 5.

44. Тип Beacon. DEADLINE = 500 мс, LATENCY_BUDGET = 10 мс.

45. Period задаётся случайно в диапазоне 100–700 мс через std::uniform_int_distribution.

46. За сеанс длительностью 60 с подсчитать число пропусков DEADLINE и средний фактический период.

47. Объяснить, при каком среднем периоде доля пропусков становится значимой ($> 10\%$).

Вариант 6.

43. Тип Pose {double x; double y;}. DEADLINE = 50 мс, LATENCY_BUDGET = 1 мс. Period = 20 мс.

44. Дополнительно RELIABLE + netem loss 30 % на интерфейсе lo.

45. Замерить задержку каждого принятого сэмпла; подсчитать число on_requested_deadline_missed.

46. Сравнить максимальную задержку повторной передачи с DEADLINE; сделать вывод о применимости RELIABLE при таком DEADLINE.

Вариант 7.

45. Тип Beacon. DEADLINE = 300 мс.

- 46. Запустить два Publisher (с разными ключами в IDL через @key long sensor_id) с периодами 100 мс и 400 мс на один топик.
- 47. В Subscriber по last_instance_handle отличать пропуски по разным экземплярам.
- 48. Подтвердить, что пропуски фиксируются только по «медленному» экземпляру.

Вариант 8.

- 45. Тип Beacon. DEADLINE = 100 мс. Period = 50 мс.
- 46. Опыт 1: LATENCY_BUDGET = 0 (по умолчанию). Опыт 2: LATENCY_BUDGET = 10 мс.
- 47. В обоих опытах собрать распределение задержек по 200 сэмплам.
- 48. Сравнить avg и p95 двух распределений; объяснить, насколько LATENCY_BUDGET = 10 мс влияет на фактическую группировку сэмплов в RTPS.

Вариант 9.

- 46. Тип Beacon. DEADLINE = 1 с.
- 47. Publisher: 10 сэмплов с периодом 100 мс, потом 5 с молчит, потом ещё 10 сэмплов.
- 48. Подсчитать число on_requested_deadline_missed за весь сеанс.
- 49. Объяснить связь между длиной паузы (5 с) и числом пропусков (должно быть около 5).

Вариант 10.

- 45. Тип Beacon. DEADLINE = 250 мс. Period = 200 мс.
- 46. Запустить под нагрузкой stress-ng --cpu N на одном ПК.
- 47. Замерить число пропусков и avg/p95 задержки для N = 0, 4, 8.
- 48. Построить таблицу N → пропуски, p95; сделать вывод о необходимости SCHED_FIFO в системах реального времени.

Вариант 11.

- 46. Тип CmdVel {double v; double w;}. DEADLINE = 100 мс, LATENCY_BUDGET = 5 мс. Period = 50 мс. RELIABLE.
- 47. Publisher: $v = 0.5 + 0.1 \cdot \sin(t)$, $w = 0.2 \cdot \cos(t)$.
- 48. В Subscriber замерить avg/p95/max задержки доставки.
- 49. Сравнить максимальную задержку с LATENCY_BUDGET; сделать вывод о пригодности для контура управления.

Вариант 12.

- 46. Тип Beacon. DEADLINE = 200 мс. Один Publisher.
- 47. Запустить два Subscriber: один без netem, второй – через netem delay 250ms (нарушение DEADLINE).
- 48. Сравнить число on_requested_deadline_missed у двух подписчиков за 30 с.
- 49. Объяснить, почему «здоровый» Subscriber не страдает от проблем второго.

Вариант 13.

- 45. Тип Beacon. DEADLINE = 100 мс. Period = 80 мс, N = 200.
- 46. Опыт 1: запустить процесс с приоритетом по умолчанию (nice 0).
- 47. Опыт 2: запустить под chrt -f 50 (SCHED_FIFO приоритет 50).
- 48. Сравнить число пропусков и p95 задержки в двух опытах.

Вариант 14.

- 46. Тип Alarm {long code; string<32> source;}. DEADLINE = 5 с.
- 47. Publisher отправляет сэмпл только по событию: задержка между событиями случайна в диапазоне 1–10 с.
- 48. За сеанс длительностью 30 с подсчитать число пропусков DEADLINE на стороне DataReader.
- 49. Объяснить, как такие пропуски могут служить признаком отсутствия активности в системе мониторинга.

Вариант 15.

46. Тип Beacon. DEADLINE = 50 мс, LATENCY_BUDGET = 0.5 мс. Period = 25 мс, N = 400.
47. Замерить avg/p95/max задержки; подсчитать долю сэмплов с задержкой $\leq$ 500 мкс.
48. Зафиксировать число on*_deadline_missed при штатной работе (должно быть 0).
49. Сравнить с опытом, где Period = 60 мс (нарушение DEADLINE); подтвердить рост числа пропусков.

Лабораторная работа 12. Распределённая система с несколькими участниками (Domain, Partition)

Цель работы: построить распределённую систему из трёх узлов на базе Fast DDS 3.x, разделить трафик с помощью идентификаторов доменов (domain_id) и политики PARTITION, наблюдать работу Simple Discovery Protocol при добавлении и отключении узлов, оценить устойчивость системы к пропаданию одного из участников и закрепить умение настраивать сетевое окружение для DDS на нескольких ПК или виртуальных машинах.

Теоретическое обоснование

1. Домен как граница видимости. DomainParticipant создаётся с обязательным параметром domain_id (целое от 0 до 232). Все участники одного домена автоматически обнаруживают друг друга и обмениваются метаданными о созданных DataWriter и DataReader; участники разных доменов разделены полностью – пакеты одного домена не доставляются в другой даже при совпадении адресов. Разделение опирается на разные multicast-порты Simple Discovery Protocol: каждому domain_id соответствует уникальный порт $7400 + 250 \cdot \text{domain_id}$.

2. Простой протокол обнаружения (SDP). Протокол состоит из двух фаз. Сначала Participant Discovery Phase (PDP): каждый DomainParticipant периодически рассылает в multicast-группу 239.255.0.1 объявление PDP с собственным GUID и списком unicast-адресов. После взаимного обнаружения участники переходят к Endpoint Discovery Phase (EDP): обмену встроенными топиками DCPSPublications и DCPSSubscriptions, в которых публикуются все созданные на участнике DataWriter и DataReader. По итогам DataWriter и DataReader, совместимые по топику, типу и QoS, переходят в состояние matched.

3. Политика PARTITION. PARTITION – строковая «логическая ветка» внутри топика, доступная и Publisher, и Subscriber как список имён партиций. Сэмплы доставляются только если множества партиций Publisher и Subscriber

пересекаются хотя бы по одному имени. По умолчанию список партиций пуст – это эквивалентно одной партиции с пустым именем; такие endpoint-ы видны друг другу. PARTITION задаётся на уровне Publisher / Subscriber (не DataWriter / DataReader), поэтому распространяется на все endpoint-ы, созданные через эти фабрики.

4. Шаблоны имён партиций. Поддерживаются POSIX-подобные wildcard-шаблоны: «*» – произвольная подстрока, «?» – один произвольный символ. Это позволяет Publisher выдавать сэмплы в конкретную партицию (например, «sensors/temp»), а Subscriber – подписываться на «sensors/*» и получать оба потока. В отличие от ROS 1 namespace, PARTITION можно менять в рантайме через set_qos().

5. Сравнение разделения трафика через Domain и Partition. Краткое сопоставление приведено в таблице 1.

Таблица 1 – Domain и Partition как механизмы изоляции трафика

Свойство	Domain	Partition
Уровень изоляции	Сетевой (отдельные multicast-порты)	Логический (фильтр по строкам)
Discovery между группами	Полностью изолировано	Происходит, но endpoint-ы не matched
Шаблоны (wildcards)	Не поддерживаются	Поддерживаются (* и ?)
Смена в рантайме	Невозможна (фиксируется при создании DomainParticipant)	Возможна через set_qos()
Расход ресурсов	Каждый домен – отдельные сокеты и порты	Только проверка строк при matching

6. Конфигурация сети для нескольких узлов. На одном ПК участники общаются через петлевой интерфейс lo и SHM-транспорт. Для работы на нескольких физических ПК нужно: (а) обеспечить взаимную доступность узлов по IP в одной подсети; (б) убедиться, что маршрутизатор передаёт

multicast (или явно включить интерфейс с белым адресом); (в) открыть на каждом узле UDP-порты $7400 + 250 \cdot \text{domain_id}$ (PDP) и user-data-порты $7411 + 250 \cdot \text{domain_id}$; (г) при наличии нескольких сетевых интерфейсов – задать транспорт через XML-профиль и атрибут interfaceWhiteList.

7. XML-профиль Fast DDS 3.x. Конфигурация задаётся через переменную FASTDDS_DEFAULT_PROFILES_FILE (в Fast DDS 2.x переменная называлась FASTRTPS_DEFAULT_PROFILES_FILE, в 3.x – переименована, старое имя ещё поддерживается для совместимости). Структура корневого XML-документа описана в таблице 2.

Таблица 2 – Структура XML-профиля Fast DDS 3.x для двух узлов

Элемент	Назначение
<participant profile_name="node">	Корневой профиль участника
<rtps><builtinTransports>UD Pv4</builtinTransports></rtps>	Отключить SHM и использовать только UD Pv4
<userTransports>	Список явных транспортов (для interfaceWhiteList)
<interfaceWhiteList><address>192.168.1.0</address></interfaceWhiteList>	Привязка discovery к конкретной подсети

8. Поведение при отказе узла. Когда узел исчезает, остальные обнаруживают это по тайм-ауту lease_duration (по умолчанию 20 с). До истечения тайм-аута DataWriter в режиме RELIABLE будет копить сэмплы;

по истечении – DomainParticipant удаляется из локальной таблицы, matched-связь снимается. Возвращение узла снова запускает SDP, и связь восстанавливается без участия приложения. Целевая топология эксперимента приведена на рисунке 1.

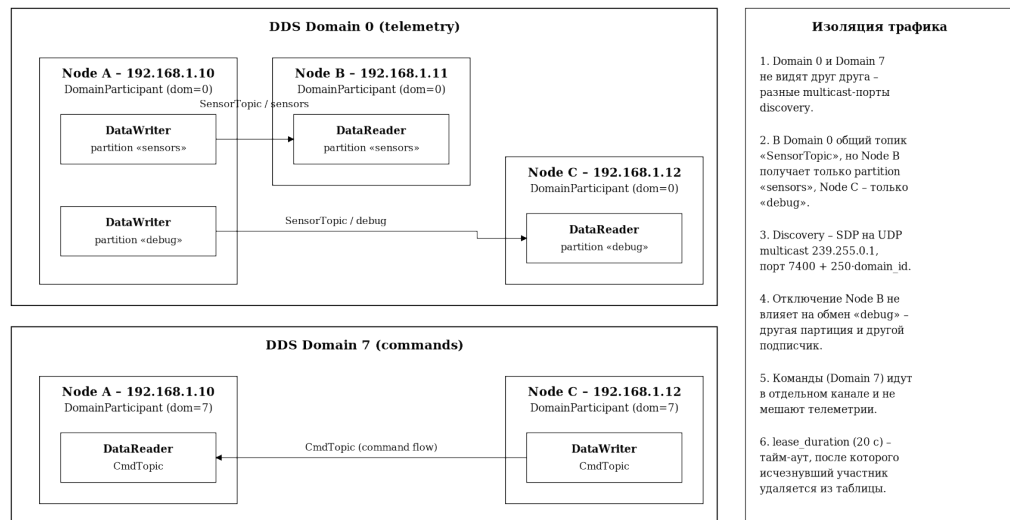


Рисунок 1 – Распределённая система из трёх узлов: два домена и две партиции

Пример выполнения задания

В качестве «нулевого» варианта рассматривается система из трёх узлов А, В, С. На каждом узле запускаются один-два процесса Fast DDS. Узлы могут быть физическими ПК в подсети 192.168.1.0/24 либо тремя терминалами на одном ПК (тогда «адрес» эмулируется уникальными значениями имени участника и domain_id). Тип сообщения единый – Telemetry с целочисленным sequence-номером и идентификатором источника.

Шаг 1. Описание типа и генерация TypeSupport.

Листинг 1 – Файл Telemetry.idl

```
// Telemetry.idl
struct Telemetry
{
    long seq;
    long node_id;
};
```

Листинг 2 – Генерация

```
fastddsgen -replace Telemetry.idl
```

Шаг 2. Узел А – Publisher в Domain 0, партиция «sensors». Узел А создаёт DomainParticipant в домене 0, Publisher с PARTITION = {«sensors»} и единственный DataWriter. Каждые 200 мс отправляется очередной сэмпл.

Листинг 3 – Файл NodeA.cpp

```
#include <chrono>
#include <thread>
#include <iostream>
#include <fastdds/dds/domain/DomainParticipantFactory.hpp>
#include <fastdds/dds/domain/DomainParticipant.hpp>
#include <fastdds/dds/publisher/Publisher.hpp>
#include <fastdds/dds/publisher/DataWriter.hpp>
#include <fastdds/dds/topic/TypeSupport.hpp>
#include "TelemetryPubSubTypes.hpp"

using namespace eprosima::fastdds::dds;

int main()
{
    DomainParticipantQos pqos;
    pqos.name("NodeA");
    auto* dp = DomainParticipantFactory::get_instance()
        ->create_participant(0, pqos);
    TypeSupport type(new TelemetryPubSubType());
    type.register_type(dp);
    auto* topic = dp->create_topic(
        "DataTopic", type.get_type_name(), TOPIC_QOS_DEFAULT);

    PublisherQos pubqos = PUBLISHER_QOS_DEFAULT;
    pubqos.partition().push_back("sensors");
    auto* pub = dp->create_publisher(pubqos);

    auto* writer = pub->create_datawriter(
        topic, DATAWRITER_QOS_DEFAULT);

    Telemetry msg;
    msg.node_id(1);
    for (int i = 1; i <= 100; ++i)
    {
        msg.seq(i);
        writer->write(&msg);
        std::cout << "A sent seq=" << i << "\n";
        std::this_thread::sleep_for(std::chrono::milliseconds(200));
    }
}
```

```

pub->delete_datawriter(writer);
dp->delete_publisher(pub);
dp->delete_topic(topic);
DomainParticipantFactory::get_instance()
    ->delete_participant(dp);
return 0;
}

```

Шаг 3. Узел В – Subscriber в Domain 0, партиция «sensors». Этот узел получает сэмпле от А; партиции совпадают – связь matched.

Листинг 4 – Файл NodeB.cpp

```

#include <chrono>
#include <thread>
#include <iostream>
#include <fastdds/dds/domain/DomainParticipantFactory.hpp>
#include <fastdds/dds/domain/DomainParticipant.hpp>
#include <fastdds/dds/subscriber/Subscriber.hpp>
#include <fastdds/dds/subscriber/DataReader.hpp>
#include <fastdds/dds/subscriber/DataReaderListener.hpp>
#include <fastdds/dds/subscriber/SampleInfo.hpp>
#include <fastdds/dds/topic/TypeSupport.hpp>
#include "TelemetryPubSubTypes.hpp"

using namespace eprosima::fastdds::dds;

class L : public DataReaderListener
{
public:
    void on_subscription_matched(
        DataReader*, const SubscriptionMatchedException& s) override
    {
        std::cout << "B matched change="
            << s.current_count_change << "\n";
    }
    void on_data_available(DataReader* r) override
    {
        Telemetry msg; SampleInfo info;
        while (r->take_next_sample(&msg, &info) == RETCODE_OK)
            if (info.valid_data)
                std::cout << "B recv node=" << msg.node_id()
                    << " seq=" << msg.seq() << "\n";
    }
};

int main()
{
    DomainParticipantQos pqos;

```

```

pqos.name("NodeB");
auto* dp = DomainParticipantFactory::get_instance()
    ->create_participant(0, pqos);
TypeSupport type(new TelemetryPubSubType());
type.register_type(dp);
auto* topic = dp->create_topic(
    "DataTopic", type.get_type_name(), TOPIC_QOS_DEFAULT);

SubscriberQos subqos = SUBSCRIBER_QOS_DEFAULT;
subqos.partition().push_back("sensors");
auto* sub = dp->create_subscriber(subqos);

L listener;
auto* reader = sub->create_datareader(
    topic, DATAREADER_QOS_DEFAULT, &listener);

std::this_thread::sleep_for(std::chrono::seconds(40));

sub->delete_datareader(reader);
dp->delete_subscriber(sub);
dp->delete_topic(topic);
DomainParticipantFactory::get_instance()
    ->delete_participant(dp);
return 0;
}

```

Шаг 4. Узел С – Subscriber в Domain 0, партиция «debug». Узел С подписывается на тот же топик, но с другой партицией. По умолчанию сообщения от А (партиция «sensors») не должны доходить до С.

Листинг 5 – Файл NodeC.cpp – ключевое отличие от NodeB

```

DomainParticipantQos pqos;
pqos.name("NodeC");
auto* dp = DomainParticipantFactory::get_instance()
    ->create_participant(0, pqos);
// ...
SubscriberQos subqos = SUBSCRIBER_QOS_DEFAULT;
subqos.partition().push_back("debug"); // другая партиция
auto* sub = dp->create_subscriber(subqos);

```

Шаг 5. Запуск базового сценария. Три процесса запускаются на одном или разных ПК в любом порядке; discovery срабатывает автоматически. Ожидаемое поведение приведено в листинге 5.

Листинг 6 – Совмещённый вывод трёх узлов

```
# Node A
$ ./node_a
A sent seq=1
A sent seq=2
...

# Node B (партиция «sensors») – принимает все сэмплы
$ ./node_b
B matched change=1
B recv node=1 seq=1
B recv node=1 seq=2
...

# Node C (партиция «debug») – ничего не принимает
$ ./node_c
(matched не срабатывает, recv нет)
```

Шаг 6. Проверка фильтрации по `domain_id`. Если на любом из узлов поменять `domain_id` с 0 на 7 (передать как аргумент в `create_participant`), процесс перестаёт видеть остальных – `discovery` идёт уже в другом порту, и `matched` не срабатывает ни при каких партициях. Это подтверждает сетевую изоляцию доменов.

Листинг 7 – Запуск NodeA в Domain 7 (на узле А добавлен параметр)

```
$ ./node_a 7
A sent seq=1
...

# Node B и Node C по-прежнему работают в Domain 0
$ ./node_b
(matched не срабатывает)
```

Шаг 7. Проверка устойчивости к отключению узла. После того как связь установлена, узел В принудительно завершается комбинацией Ctrl-C. Поведение узла А зависит от QoS RELIABILITY: при BEST_EFFORT отправка продолжается без изменений; при RELIABLE DataWriter копит сэмплы в кэше до истечения `lease_duration` (20 с по умолчанию), после чего связь автоматически снимается. Перезапуск узла В без перезапуска А приводит к новому циклу `discovery` и продолжению приёма с того момента, на котором А находится в этот момент.

Шаг 8. Выводы. Эксперимент демонстрирует три механизма изоляции Fast DDS: домен (сетевая изоляция), партиция (логическая изоляция в пределах домена) и `lease_duration` (временная изоляция при отказе узла). Сочетание этих механизмов позволяет строить распределённые системы, в которых разные подсистемы (телеметрия, команды, отладка) разделены на уровне `middleware` и не требуют отдельных брокеров или контроллеров.

Задания по вариантам

Номер варианта определяется как $(N - 1) \bmod 15 + 1$, где N – порядковый номер студента в журнале группы. Конкретные числовые параметры задачи берутся из приведённой ниже таблицы. Составить отчет со скриншотами, показывающие выполнение заданий.

Вариант 1.

- 52. Три узла, Domain 0, тип Telemetry.
- 53. Партиции: A – «sensors», B – «sensors», C – «*» (wildcard).
- 54. Запустить все три процесса и подтвердить, что Node C принимает сэмплы благодаря wildcard – matched срабатывает, `recv` появляется в логе.
- 55. Поменять у C партицию на «sens?rs» (один любой символ); подтвердить, что приём сохраняется.

Вариант 2.

- 47. Три узла, тип Telemetry.
- 48. A – Publisher в Domain 0, B – Subscriber в Domain 0, C – Subscriber в Domain 1.
- 49. Запустить все три; подтвердить, что `matched` срабатывает только у B, у C – нет.
- 50. Подсчитать число принятых сэмплов у B и C за 30 с; объяснить результат через разницу multicast-портов.

Вариант 3.

- 48. Три узла, тип Telemetry.

49. A – Publisher с PARTITION = {«telem/cmd», «telem/log»} (одновременно в двух партициях).

50. B – Subscriber на «telem/cmd», C – Subscriber на «telem/log».

51. За 30 с подсчитать число принятых сэмплов у каждого; подтвердить, что суммарно у B + C $\approx 2N$ (каждый сэмпл доставляется в обе партиции).

Вариант 4.

48. Три узла, тип Telemetry. A – Publisher (RELIABLE) в Domain 0, B – Subscriber (RELIABLE) в Domain 0.

49. Через 5 с после старта принудительно завершить B (Ctrl-C).

50. В A реализовать on_publication_matched и логировать момент снятия matched.

51. Замерить время от завершения B до снятия matched в логе A; сравнить с lease_duration (20 с).

Вариант 5.

48. Четыре узла, тип Telemetry. Domain 0.

49. A, B, C – Publisher с разными node_id (1, 2, 3), период 100 мс каждый.

50. D – Subscriber, который ведёт три счётчика по node_id.

51. За 30 с подтвердить, что счётчики ≈ 300 для каждого источника; вывести таблицу node_id $\rightarrow$ count.

Вариант 6.

47. Три узла, тип Telemetry. Domain 0.

48. A – Publisher с PARTITION = {«default»}.

49. B – Subscriber без указания PARTITION (пустой список).

50. Запустить и объяснить (с проверкой по логу), доходят ли сэмплы; обосновать правилом «пустой список = одна партиция с пустым именем».

Вариант 7.

49. Три узла, тип Telemetry. Domain 5.

50. A – Publisher, B и C – Subscriber.

- 51. Запустить все три; через 10 с принудительно перезапустить С.
- 52. Подтвердить по логу А (на `on_publication_matched`), что `matched` сначала срабатывает на двух Subscriber, потом уменьшается на 1 при остановке С, потом снова увеличивается при перезапуске.

Вариант 8.

- 49. Три узла. Тип Telemetry. Domain 0. А и В на одном ПК, С – на другом в той же подсети 192.168.1.0/24.
- 50. Проверить ping и работу UDP multicast (`iperf -u -B 239.255.0.1`).
- 51. Запустить процессы; в С реализовать `on_subscription_matched`.
- 52. Замерить время от старта С до первого `matched`; объяснить вкладом PDP-цикла Fast DDS (примерно 2–3 с по умолчанию).

Вариант 9.

- 50. Три узла, тип Telemetry. Domain 0. А – Publisher, В – Subscriber.
- 51. Подготовить XML-профиль `fastdds.xml` с `interfaceWhiteList` только для одного сетевого интерфейса.
- 52. Экспортировать `FASTDDS_DEFAULT_PROFILES_FILE=$PWD/fastdds.xml` и запустить оба процесса.
- 53. Через `tcpdump` на «разрешённом» и «запрещённом» интерфейсах подтвердить, что трафик идёт только по выбранному.

Вариант 10.

- 49. Один процесс с двумя DomainParticipant: первый – Publisher в Domain 0, второй – Subscriber в Domain 1.
- 50. Создать второй процесс-зеркало: Subscriber в Domain 0, Publisher в Domain 1.
- 51. Запустить оба процесса; подтвердить, что данные между доменами не пересекаются (Domain 0-Publisher слышит только Domain 0-Subscriber и наоборот).

52. Объяснить, как один процесс может одновременно жить в двух доменах через создание двух DomainParticipant.

Вариант 11.

50. Три узла, тип Telemetry. Domain 0.

51. A – Publisher, стартует с PARTITION = {«sensors»}. Через 10 с вызывает pub->set_qos() с новой PARTITION = {«debug»}.

52. B – Subscriber на «sensors», C – Subscriber на «debug».

53. Зафиксировать в логе обоих момент смены matched (исчезает у B, появляется у C); подтвердить корректность рантайм-смены.

Вариант 12.

50. Три узла. Domain 0. A – Publisher, B и C – Subscriber, оба в партиции «common».

51. Запустить и подсчитать число сэмплов на стороне B и C за 30 с.

52. Подтвердить, что оба получают одинаковую последовательность (B.count = C.count, и сэмплы идентичны по seq).

53. Объяснить, как один write() доходит до двух DataReader через multicast-доставку RTPS.

Вариант 13.

49. Три узла. Тип Telemetry. Domain 0.

50. B – Subscriber на «sub_*», C – Subscriber на «*_priority».

51. Подобрать имя партиции Publisher A так, чтобы сэмплы получали оба (то есть имя удовлетворяет обоим шаблонам).

52. Подтвердить экспериментально; объяснить, какое имя подходит (например, «sub_priority»).

Вариант 14.

50. Три узла. Тип Telemetry. Все в Domain 0, без партиций.

51. На одном ПК отключить multicast: sudo ip link set lo multicast off.

52. Запустить процессы; зафиксировать поведение discovery (matched может срабатывать через unicast initial peers, если они заданы, иначе – нет).
53. Включить multicast обратно и подтвердить восстановление discovery без перезапуска процессов.

Вариант 15.

50. Три узла. Domain 0. A – Publisher.
51. B и C – Subscriber; реализовать цикл: каждые 15 с C перезапускается (kill + новый запуск).
52. B A считать число событий matched/unmatched через on_publication_matched.
53. За 60 с подсчитать суммарное число изменений matched; подтвердить, что оно близко к 8 (4 unmatched + 4 matched при 4 перезапусках).

Лабораторная работа 13. Синхронизация времени с PTP (Linux PTP)

Цель работы: освоить практическую настройку протокола точного времени PTP (IEEE 1588-2019) в Linux с помощью пакета linuxptp 4.x. Развернуть на двух виртуальных машинах Ubuntu 24.04 LTS схему master-slave: на одной запустить ptp4l в режиме мастера, на второй – в режиме ведомого; добиться синхронизации аппаратных часов (PHC) сетевой карты ведомого узла с часами мастера, измерить смещение offset и скорость дрейфа drift по логам, а затем синхронизировать системные часы CLOCK_REALTIME с PHC с помощью утилиты phc2sys и оценить итоговую точность синхронизации в микросекундном диапазоне.

Теоретическое обоснование

1. Назначение и модель PTP. Протокол точного времени PTP (Precision Time Protocol), стандартизированный как IEEE 1588 (редакции 2008 и 2019 годов), предназначен для синхронизации часов узлов локальной сети с точностью до микросекунд (программная реализация) и до десятков наносекунд (аппаратная реализация на сетевой карте, поддерживающей временные метки на уровне MAC). В отличие от NTP, который обеспечивает точность миллисекунд при работе через WAN, PTP рассчитан на детерминированную локальную среду и применяется в промышленной автоматизации, телекоммуникациях, синхронизации фаз в энергетике, измерительных стендах и распределённых системах управления в реальном времени. Профиль gPTP (IEEE 802.1AS-2020) – родственная разновидность PTP для Time-Sensitive Networking, отличается обязательным L2-транспортом и фиксированным набором QoS-параметров.

В сети PTP выделяются несколько ролей часов: Grandmaster Clock – источник эталонного времени для всего домена; Ordinary Clock – обычные часы узла, работающие в режиме master или slave; Boundary Clock – граничные часы коммутатора, у которых один порт slave относительно вышестоящего мастера и несколько портов master относительно

нижестоящих сегментов; Transparent Clock – коммутатор, вносящий поправку на собственную внутреннюю задержку в проходящие сообщения. Выбор Grandmaster Clock в домене выполняется алгоритмом Best Master Clock Algorithm (BMCA) на основе значений `priority1`, `clockClass`, `clockAccuracy`, `offsetScaledLogVariance`, `priority2` и `clockIdentity`, сравниваемых лексикографически.

2. Обмен сообщениями и расчёт смещения. Базовая процедура синхронизации в режиме `end-to-end delay` использует четыре сообщения: `Sync`, `Follow_Up`, `Delay_Req` и `Delay_Resp`. Мастер периодически отправляет `Sync`; в момент его отправки фиксируется временная метка t_1 (если карта поддерживает `one-step` – она уезжает прямо в `Sync`, иначе передаётся следующим сообщением `Follow_Up`). Slave принимает `Sync` в момент t_2 . Затем slave отправляет `Delay_Req` в момент t_3 , мастер принимает его в момент t_4 и возвращает t_4 в ответном `Delay_Resp`. По четырём меткам вычисляется задержка сети и смещение часов:

$$delay = ((t_2 - t_1) + (t_4 - t_3)) / 2 \quad (1)$$

$$offset = ((t_2 - t_1) - (t_4 - t_3)) / 2 \quad (2)$$

Формулы (1) и (2) корректны при условии симметричности задержки сети в обоих направлениях; асимметрия маршрута непосредственно приводит к систематической ошибке `offset`. По вычисленному `offset` программный сервоконтур (PI-регулятор по умолчанию) подстраивает частоту локальных часов через интерфейс `adjtimex / clock_adjtime`, пока не сведёт `offset` к нулю.

3. Понятия `offset`, `drift` и `servo`. `Offset` – мгновенная разность между локальными часами и часами мастера в наносекундах. `Drift` (или `frequency adjustment`) – величина частотной поправки, выраженная в `parts per billion` (ppb): значение -12500 означает, что без поправки локальный осциллятор отстаёт на 12.5 микросекунды в секунду. На стационарном режиме `offset` колеблется вокруг нуля с амплитудой, определяемой джиттером сети и точностью отметок времени; `drift` стабилизируется на значении,

компенсирующем уход локального кварца, и медленно меняется при изменении температуры. Параметры сервопетли `pi_proportional_const` и `pi_integral_const` в конфигурации `ptp4l` определяют скорость и устойчивость захвата.

4. Сводка ключевых сообщений PTP приведена в таблице 1.

Таблица 1 – Сообщения протокола PTP и их назначение

Сообщение	Тип	Отправитель	Назначение
Sync	Event	Master	Несёт временную метку t_1 ; на slave фиксируется t_2
Follow_Up	General	Master	Передаёт точную t_1 в режиме two-step
Delay_Req	Event	Slave	Запрос измерения задержки; на slave фиксируется t_3
Delay_Resp	General	Master	Возврат метки t_4 приёма Delay_Req на master
Announce	General	Master	Параметры часов для BMCA; период 1-2 с
Management	General	Любой	Чтение/запись параметров через <code>pmc</code>

5. Linux PTP. Эталонная open-source реализация для Linux – пакет `linuxptp` (актуальная ветка 4.x, поддерживает PTP-профили 2.x и gPTP IEEE 802.1AS-2020). В него входят демоны `ptp4l` (PTP boundary/ordinary clock поверх UDP или Layer 2), `phc2sys` (синхронизация любых пар часов: обычно РНС сетевой карты и системного `CLOCK_REALTIME`), `pmc` (консоль управления PTP), `timemaster` (надстройка, объединяющая `ptp4l` и `chronyd`). Каждый демон ведёт лог с записями вида «master offset N s2 freq +N path delay N», где s_0 – свободный ход, s_1 – захват частоты, s_2 – синхронизированное состояние. В версии 4.x в конфигурации используются

термины `timeReceiver` / `timeTransmitter` вместо устаревших `slaveOnly` / `masterOnly`, хотя старые имена пока приняты для обратной совместимости.

Аппаратные временные метки (HWTSTAMP) ставит сетевая карта в момент пересечения пакетом MAC-уровня и доступны через `ioctl SIOCSHWTSTAMP`. Программные метки (SOFTWARE) ставит ядро при копировании пакета в сокет – они подвержены джиттеру очередей и обеспечивают точность на уровне 10-100 микросекунд. Режим работы выбирается параметром `time_stamping` в конфигурации `ptp4l`. На ядрах 6.12 и новее `PREEMPT_RT` входит в `mainline` (отдельный патч более не требуется): включение `CONFIG_PREEMPT_RT` в конфигурации ядра уменьшает джиттер программных меток в 3-5 раз за счёт вытесняемых критических секций.

Пример выполнения задания

В качестве разбираемого «нулевого» варианта рассмотрим стенд из двух виртуальных машин под управлением Ubuntu 24.04 LTS, соединённых через выделенный интерфейс `enp0s8` в режиме `internal network` гипервизора. Узел `node-m` получает адрес `10.0.10.1/24` и играет роль мастера, узел `node-s` получает адрес `10.0.10.2/24` и работает как ведомый. Домен PTP – 0, режим – `ordinary clock`, механизм задержки – E2E, временные метки – программные.

Шаг 1. Установка пакета `linuxptp` и проверка возможностей карты. На обеих ВМ выполняются команды установки и проверки списка поддерживаемых временных меток сетевой карты.

Листинг 1 – Установка `linuxptp` и проверка возможностей сетевого адаптера

```
sudo apt update
sudo apt install -y linuxptp ethtool
ethtool -T enp0s8
```

Если в выводе `ethtool` строка `Hardware Transmit Timestamp Modes` пуста, аппаратные метки недоступны – используется параметр `time_stamping software`. Для виртуальных адаптеров (`virtio-net`, `e1000`) это типичная

ситуация, поэтому в данной работе сразу ориентируемся на программные метки.

Шаг 2. Конфигурация мастера. На node-m создаётся файл /etc/linuxptp/ptp4l-master.conf со следующим содержимым.

Листинг 2 – Конфигурация ptp4l на ведущем узле (/etc/linuxptp/ptp4l-master.conf)

```
[global]
domainNumber      0
priority1          128
priority2          128
clockClass         248
clockAccuracy      0xFE
offsetScaledLogVariance 0xFFFF
logAnnounceInterval 1
logSyncInterval    0
logMinDelayReqInterval 0
tx_timestamp_timeout 10
time_stamping      software
network_transport   UDPv4
delay_mechanism     E2E
summary_interval    0
[enp0s8]
masterOnly         1
```

Параметр masterOnly = 1 фиксирует роль порта и отключает участие в ВМСА в качестве кандидата на потерю мастерства. priority1 = 128 – значение по умолчанию для рабочих часов; clockClass = 248 соответствует обычным часам, не синхронизированным с эталоном.

Шаг 3. Конфигурация ведомого. На node-s создаётся файл /etc/linuxptp/ptp4l-slave.conf, в котором отличаются роль порта и значения приоритетов.

Листинг 3 – Конфигурация ptp4l на ведомом узле (/etc/linuxptp/ptp4l-slave.conf)

```
[global]
domainNumber      0
priority1          255
priority2          255
clockClass         255
logAnnounceInterval 1
```

```
logSyncInterval    0
logMinDelayReqInterval 0
tx_timestamp_timeout 10
time_stamping      software
network_transport  UDPv4
delay_mechanism     E2E
summary_interval   0
[enp0s8]
slaveOnly          1
```

Шаг 4. Запуск `ptp4l` на обоих узлах. Демон запускается с ключами `-f` для пути к конфигу, `-i` для имени интерфейса (если не указан в секции конфига) и `-m` для вывода логов в `stdout`.

Листинг 4 – Запуск `ptp4l` в роли мастера и слейва (две разные ВМ)

```
# node-m (master)
sudo ptp4l -f /etc/linuxptp/ptp4l-master.conf -m

# node-s (slave)
sudo ptp4l -f /etc/linuxptp/ptp4l-slave.conf -m
```

Шаг 5. Чтение логов ведомого. После 5-10 секунд работы `slave` проходит фазы `s0` (свободный ход), `s1` (захват частоты) и выходит в `s2` (синхронизировано). Ожидаемый формат строк лога приведён в листинге 5.

Листинг 5 – Фрагмент журнала `ptp4l` на ведомом узле в момент захвата

```
ptp4l[12.345]: selected /dev/ptp0 as PTP clock
ptp4l[12.346]: port 1: INITIALIZING to LISTENING on INIT_COMPLETE
ptp4l[14.812]: port 1: new foreign master 001b21.ffe.abcd01-1
ptp4l[18.815]: selected best master clock 001b21.ffe.abcd01
ptp4l[18.815]: port 1: LISTENING to UNCALIBRATED on RS_SLAVE
ptp4l[19.815]: master offset -47213 s0 freq +0 path delay 12450
ptp4l[20.815]: master offset -12044 s1 freq -47213 path delay 12450
ptp4l[21.815]: master offset -1832 s2 freq -48901 path delay 12377
ptp4l[22.815]: master offset 412 s2 freq -48637 path delay 12390
ptp4l[23.815]: master offset -185 s2 freq -48705 path delay 12385
```

Поле `master offset` – текущее смещение в наносекундах; `s2` – статус «синхронизировано»; `freq` – накопленная частотная поправка в `ppb` (на примере локальный кварц отстаёт примерно на 48.7 `ppb`, или 48.7 наносекунды в секунду); `path delay` – оценка задержки сети в одну сторону. В

установившемся режиме offset держится в пределах ± 2 микросекунды для программных меток на стенде с минимальной нагрузкой.

Шаг 6. Синхронизация системных часов с РНС. Демон ptp4l подстраивает только РНС сетевой карты (/dev/ptp0). Чтобы перенести точное время на CLOCK_REALTIME ядра, запускается phc2sys.

Листинг 6 – Запуск phc2sys для переноса времени с РНС на системные часы

```
sudo phc2sys -s /dev/ptp0 -c CLOCK_REALTIME -w -m
```

Ключ -s задаёт источник, -c – назначение, -w (wait) откладывает старт до выхода ptp4l в состояние s2, -m включает вывод в stdout. Ожидаемые строки приведены в листинге 7.

Листинг 7 – Журнал phc2sys в установившемся режиме

```
phc2sys[31.815]: CLOCK_REALTIME phc offset   -3210 s0 freq   +0 delay  2450
phc2sys[32.815]: CLOCK_REALTIME phc offset   -1812 s1 freq  -3210 delay  2430
phc2sys[33.815]: CLOCK_REALTIME phc offset   -421 s2 freq  -3631 delay  2440
phc2sys[34.815]: CLOCK_REALTIME phc offset    118 s2 freq  -3513 delay  2435
```

Шаг 7. Контроль через pmc и анализ. Параметры сессии и текущее состояние портов можно прочитать утилитой pmc, обращаясь к ptp4l через UDS-сокеты.

Листинг 8 – Чтение состояния порта и идентификации мастера через pmc

```
sudo pmc -u -b 0 'GET PORT_DATA_SET'
sudo pmc -u -b 0 'GET CURRENT_DATA_SET'
sudo pmc -u -b 0 'GET TIME_PROPERTIES_DATA_SET'
```

Для отчёта по работе из логов ptp4l на интервале не менее 5 минут извлекаются столбцы offset, freq, path delay; считаются среднее, СКО и максимум абсолютного значения offset; строится график offset(t). Аналогично анализируется журнал phc2sys.

Задания по вариантам

Номер варианта определяется как $(N - 1) \bmod 15 + 1$, где N – порядковый номер студента в журнале группы. Конкретные числовые параметры задачи берутся из приведённой ниже таблицы. Составить отчет со скриншотами, показывающие выполнение заданий.

Вариант 1.

- 56. Развернуть базовый стенд: ptp4l 4.x на двух ВМ Ubuntu 24.04, домен 0, E2E, программные метки, logSyncInterval = 0 (1 Sync в секунду).
- 57. Снять журнал ptp4l на ведомом узле длительностью 10 минут, извлечь столбец master offset.
- 58. Построить гистограмму offset с шагом 200 нс, рассчитать среднее, СКО и максимум модуля offset.
- 59. Сделать вывод о типовом джиттере программных меток в виртуальной среде.

Вариант 2.

- 51. Поднять стенд по образцу из примера; на втором проходе уменьшить logSyncInterval до -3 (8 Sync/c) и logMinDelayReqInterval до -3 на обоих узлах.
- 52. Параллельно с ptp4l запустить tcpdump -i enp0s8 udp port 319 or udp port 320 и оценить полосу в Кбит/с.
- 53. Сравнить СКО offset с базовым прогоном; объяснить, почему более частые Sync уменьшают СКО только до определённого предела.

Вариант 3.

- 52. Перевести оба узла в режим P2P (delay_mechanism = P2P), сообщения Pdelay_Req / Pdelay_Resp.
- 53. Снять offset на 10 минутах при «чистом» канале и сравнить среднее СКО с режимом E2E.
- 54. Внести искусственную асимметрию задержки (netem delay 5 ms только на исходящем направлении мастера) и зафиксировать систематическую ошибку

offset; объяснить, почему P2P в симметричном случае точнее E2E, но не спасает от асимметрии.

Вариант 4.

52. Перевести транспорт на Layer 2 (network_transport = L2, Ethernet-тип 0x88F7).

53. Через tcpdump -i enp0s8 ether proto 0x88f7 подтвердить отсутствие UDP-инкапсуляции.

54. Замерить offset на 10 минутах и сравнить со средним по UDPv4 из варианта 1; объяснить, за счёт чего L2-транспорт даёт выигрыш в точности.

Вариант 5.

52. Изменить domainNumber мастера на 0, ведомого – на 1.

53. Через rmc подтвердить, что у ведомого отсутствуют foreign masters, состояние порта остаётся LISTENING.

54. Привести оба узла в один домен (1) и зафиксировать время от перезапуска ведомого до выхода в s2.

Вариант 6.

51. На интерфейсе ведомого выполнить tc qdisc add dev enp0s8 root netem delay 5ms (постоянная задержка только в одном направлении).

52. Снять журнал ptr4l на 5 минутах; зафиксировать установившееся значение offset и path delay.

53. Подтвердить расчётом по формулам (1)-(2), что систематическое смещение offset равно половине дополнительной асимметричной задержки.

Вариант 7.

53. Внести джиттер сети через tc qdisc netem delay 1ms 500us distribution normal на интерфейсе ведомого.

54. Снять offset за 10 минут, построить распределение и сравнить его СКО с базовым прогоном.

55. Подобрать `pi_proportional_const`, при котором СКО `offset` минимально, и зафиксировать значение.

Вариант 8.

53. Внести потери пакетов `tc qdisc netem loss 5 %` на ведомом.

54. Зафиксировать, сколько подряд пропущенных Sync приводит к выходу из `s2` в `s1`, и сколько – к `s0`.

55. Увеличить `logSyncInterval` до 1 (1 Sync на 2 с) при тех же потерях; объяснить, почему более редкие Sync делают захват менее устойчивым к потерям.

Вариант 9.

54. Поднять третий узел `node-m2` как второго мастера с `priority1 = 100` (более приоритетного), `domain 0`.

55. Через `ptmc` на ведомом подтвердить, что ВМСА переключил slave на `node-m2`.

56. Остановить `node-m2` и измерить, за сколько секунд slave выберет обратно `node-m` по тайм-ауту `Announce`.

Вариант 10.

53. Запустить установившуюся синхронизацию (`s2`, `drift` стабилизирован).

54. Внезапно перезапустить `ptp4l` мастера; в журнале ведомого зафиксировать момент потери Sync и момент возврата в `s2`.

55. Построить график `offset(t)` на интервале ± 30 с от момента потери; оценить перерегулирование и время установления переходного процесса.

Вариант 11.

54. Запустить `phc2sys` в обратном направлении: `-s CLOCK_REALTIME -c /dev/ptp0` (имитация выгрузки времени из NTP в PHS).

55. Снять журнал `phc2sys` на 5 минутах, сравнить СКО `offset` с прямым направлением из примера.

56. Объяснить, почему обратное направление обычно менее точно (шум CLOCK_REALTIME от NTP).

Вариант 12.

54. Снять переходный процесс `offset(t)` при значениях по умолчанию `pi_proportional_const` и `pi_integral_const`.

55. Подобрать пару значений, уменьшающих время установления в два раза; зафиксировать значения и перерегулирование.

56. Объяснить, почему слишком большие коэффициенты приводят к колебаниям `offset` вокруг нуля с растущей амплитудой.

Вариант 13.

53. Включить `summary_interval = 4` (одна сводная строка каждые 16 с) на обоих узлах.

54. Снять только `summary`-строки за 30 минут; подтвердить, что `gms` и `max` в `summary` считаются именно по тем `offset`, что обычно печатаются каждую секунду.

55. Сравнить размер журнала с базовым прогоном; оценить выигрыш по дисковому объёму при сохранении полноты статистики.

Вариант 14.

54. Запустить `ptp4l` и `phc2sys` при `logSyncInterval = 0`; через `top -p $(pgrep -d, ptp4l,phc2sys)` снять % CPU за 5 минут.

55. Повторить эксперимент при `logSyncInterval = -3` (8 Sync/c) и `logMinDelayReqInterval = -3`.

56. Построить столбчатую диаграмму % CPU vs частота сообщений; объяснить, какой компонент даёт основную нагрузку.

Вариант 15.

54. На том же сегменте поднять `chronyd` на ведомом, забирающий время с того же узла-мастера (NTP).

55. Параллельно с PTP-синхронизацией снимать chronyc tracking и offset из ptp4l.

56. Свести offset обоих демонов в одну таблицу за 15 минут; построить графики и сделать вывод о соотношении точностей PTP и NTP в локальной сети.

Лабораторная работа 14. Симуляция отказоустойчивой системы (Leader-Follower)

Цель работы: освоить принципы построения отказоустойчивой распределённой системы с одним активным лидером (Leader-Follower) и автоматическим переизбранием при сбое. Реализовать на трёх узлах симулятор, в котором лидер периодически рассылает heartbeat двум фолловерам; при пропуске нескольких heartbeat подряд фолловеры по таймауту запускают процедуру выборов и переводят одного из них в роль лидера, согласуя решение с кворумом. Измерить время восстановления t_{recovery} от момента сбоя до выхода нового лидера в активный режим и оценить зависимость этого времени от периода heartbeat и порога election timeout.

Теоретическое обоснование

1. Модель отказа и роли узлов. Простейшая отказоустойчивая распределённая система состоит из набора одинаковых процессов, один из которых в каждый момент времени играет роль лидера (Leader): принимает входные команды от клиента, обрабатывает их и реплицирует результат остальным узлам – фолловерам (Follower). При отказе лидера (crash-stop fault: процесс падает и больше не отвечает; либо network partition: становится недоступен по сети) система должна автоматически выбрать нового лидера среди живых узлов и продолжить работу. Свойства, которые требуется обеспечить, формулируются в терминах safety («в каждый момент времени не более одного лидера в одной эпохе») и liveness («если большинство узлов живо, то рано или поздно лидер избирается»).

2. Heartbeat и обнаружение сбоя. Базовый механизм обнаружения отказа – периодическая рассылка лидером служебных сообщений heartbeat с периодом T_{hb} . Фолловер ведёт таймер election_timeout длительностью T_{to} ; при получении любого heartbeat от текущего лидера таймер сбрасывается. Если за T_{to} от лидера не пришло ни одного сообщения, фолловер считает

лидера упавшим и инициирует выборы. Чтобы исключить ложные срабатывания из-за сетевых задержек и пропусков пакетов, должно выполняться:

$$T_{to} \geq k \cdot T_{hb}, \quad k \geq 3 \quad (1)$$

Коэффициент k определяет компромисс между чувствительностью и устойчивостью: при $k = 3$ для срабатывания нужно три пропущенных подряд heartbeat. Чтобы при одновременном таймауте у нескольких фолловеров они не запускали выборы синхронно (split vote), у каждого узла T_{to} выбирается случайно в диапазоне $[T_{to_min}, T_{to_max}]$, обычно $[150, 300]$ мс при $T_{hb} = 50$ мс.

3. Алгоритм выборов и кворум. Минимальный набор шагов выборов в стиле RAFT: фолловер с истёкшим таймером инкрементирует номер эпохи term, переходит в состояние Candidate, голосует за себя и рассылает request_vote(term, candidate_id) всем остальным узлам. Каждый узел в одной эпохе term отдаёт голос только одному кандидату – тому, чьё request_vote пришёл первым; ответ vote_granted = true. Кандидат становится лидером, как только соберёт большинство голосов $majority = \lfloor N/2 \rfloor + 1$, и сразу рассылает heartbeat с новым term, чтобы остановить остальные таймеры и завершить выборы. Для $N = 3$ $majority = 2$, то есть достаточно голоса самого кандидата и одного фолловера – кворума.

4. Время восстановления. Полное время восстановления $t_{recovery}$ складывается из четырёх компонент:

$$t_{recovery} = T_{to} + t_{rv} + t_{vote} + t_{hb_new} \quad (2)$$

Здесь T_{to} – интервал ожидания таймера выборов; t_{rv} – время доставки request_vote до фолловеров; t_{vote} – время сбора кворума голосов; t_{hb_new} – интервал до первого heartbeat нового лидера. На локальной сети t_{rv} , t_{vote} , t_{hb_new} пренебрежимо малы (1-5 мс), поэтому в реальности $t_{recovery} \approx T_{to}$. Уменьшение T_{to} ускоряет восстановление, но повышает риск ложных переизбраний при сетевых задержках.

5. Сводка состояний узла приведена в таблице 1.

Таблица 1 – Состояния узла в Leader-Follower с failover

Состояние	Описание	Переход
Follower	Принимает heartbeat и команды; сбрасывает election_timeout	election_timeout истёк -> Candidate
Candidate	Инициировал выборы; собирает голоса в новой эпохе term + 1	majority голосов -> Leader; принят heartbeat с term $\geq$ моего -> Follower; новый timeout -> Candidate с term + 1
Leader	Рассылает heartbeat; обрабатывает команды клиента; реплицирует данные	Принят heartbeat с term > моего -> Follower

6. Сценарий взаимодействия трёх узлов при сбое лидера показан на рисунке 1. До момента сбоя лидер node-1 рассылает heartbeat обоим фолловерам с периодом T_{hb} . После аварии node-1 перестаёт отвечать; на node-2 первым истекает election_timeout, узел переходит в Candidate, инкрементирует term, отправляет request_vote, получает голос node-3 (кворум) и переходит в Leader, начиная рассылку heartbeat с новой эпохой.

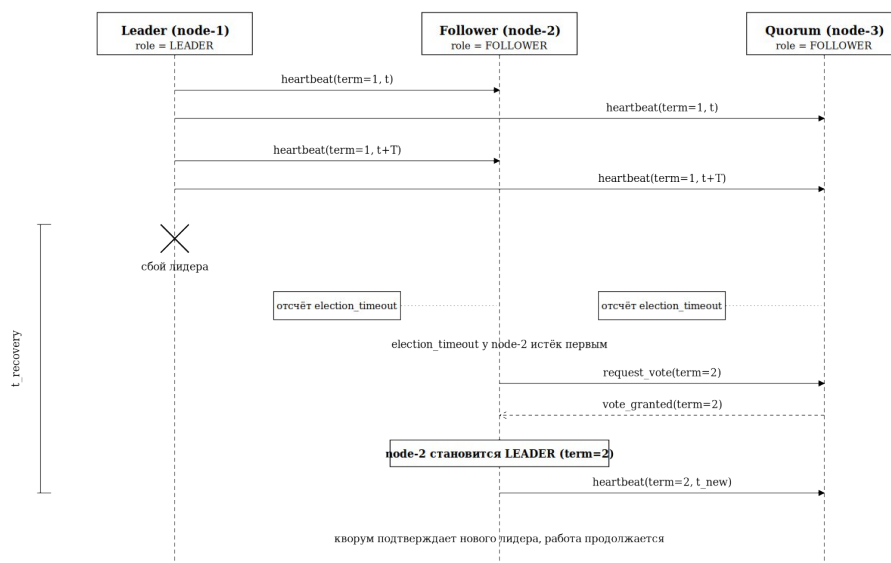


Рисунок 1 – Диаграмма последовательности: переизбрание лидера при отказе

7. Реализационные нюансы. Транспорт для heartbeat и request_vote удобно взять ZeroMQ (паттерн PUB/SUB для heartbeat и REQ/REP для голосования) либо raw UDP-сокеты – в обоих случаях нет накладных расходов на установление соединения, и потеря одного сообщения автоматически отрабатывается следующим. Каждый узел запускается с уникальным node_id и списком адресов (host:port) остальных участников. Логирование событий ведётся с временной меткой монотонных часов time.monotonic_ns(), что позволяет потом вычислить t_recovery как разность меток «leader down» (последний пропущенный heartbeat) и «I am leader now».

Пример выполнения задания

В качестве разбираемого «нулевого» варианта рассмотрим стенд из трёх процессов, запускаемых на одной машине на портах 5001, 5002 и 5003. Транспорт – UDP. Параметры: период heartbeat $T_{hb} = 100$ мс, диапазон election_timeout [400, 700] мс (то есть k от 4 до 7). Сбой эмулируется отправкой SIGKILL процессу-лидеру.

Шаг 1. Структура сообщений и состояний. Описываем enum состояний и dataclass для сообщений heartbeat и request_vote с полями term, sender_id и monotonic-меткой времени.

Листинг 1 – Базовые типы данных узла и формат сообщений

(node_types.py)

```
from dataclasses import dataclass
from enum import Enum
import json

class Role(str, Enum):
    FOLLOWER = 'follower'
    CANDIDATE = 'candidate'
    LEADER = 'leader'

@dataclass
class Message:
    kind: str      # 'hb' | 'vote_req' | 'vote_ok'
    term: int
    sender: int
    ts_ns: int

    def to_bytes(self) -> bytes:
        return json.dumps(self.__dict__).encode('utf-8')

    @classmethod
    def from_bytes(cls, raw: bytes) -> 'Message':
        return cls(**json.loads(raw.decode('utf-8')))
```

Шаг 2. Основной цикл узла. Реализация показана в листинге 2: сокет non-blocking, тики каждые 20 мс, в каждом тике обновляется роль и при необходимости рассылаются служебные сообщения.

Листинг 2 – Основной цикл узла Leader-Follower (node.py)

```
import socket, random, time, sys
from node_types import Role, Message

TICK = 0.02          # 20 мс
T_HB = 0.10          # heartbeat 100 мс
T_TO_MIN = 0.40
T_TO_MAX = 0.70

class Node:
    def __init__(self, node_id, port, peers):
        self.id = node_id
        self.peers = peers # [(host, port), ...]
        self.role = Role.FOLLOWER
        self.term = 0
        self.leader_id = None
        self.voted_for = {} # term -> sender_id
        self.votes = set()
```

```

self.sock = socket.socket(socket.AF_INET,
                           socket.SOCK_DGRAM)
self.sock.bind(('127.0.0.1', port))
self.sock.setblocking(False)
self._reset_timeout()
self.last_hb_sent = 0.0

def _reset_timeout(self):
    self.deadline = time.monotonic() + random.uniform(
        T_TO_MIN, T_TO_MAX)

def _send(self, msg: Message):
    data = msg.to_bytes()
    for host, port in self.peers:
        try:
            self.sock.sendto(data, (host, port))
        except OSError:
            pass

def _now_ns(self):
    return time.monotonic_ns()

def step_rcv(self):
    while True:
        try:
            raw, _ = self.sock.recvfrom(4096)
        except BlockingIOError:
            return
        m = Message.from_bytes(raw)
        if m.term > self.term:
            self.term = m.term
            self.role = Role.FOLLOWER
            self.leader_id = None
        if m.kind == 'hb' and m.term >= self.term:
            self.term = m.term
            self.role = Role.FOLLOWER
            self.leader_id = m.sender
            self._reset_timeout()
        elif m.kind == 'vote_req':
            if self.voted_for.get(m.term) is None:
                self.voted_for[m.term] = m.sender
                self._send(Message('vote_ok', m.term,
                                   self.id, self._now_ns()))
            elif m.kind == 'vote_ok' and self.role == Role.CANDIDATE:
                if m.term == self.term:
                    self.votes.add(m.sender)
                    if len(self.votes) + 1 >= 2: # majority
                        self._become_leader()

def _become_leader(self):
    self.role = Role.LEADER

```

```

self.leader_id = self.id
print(f'{self._now_ns()} node{self.id} LEADER '
      f'term={self.term}')
self.last_hb_sent = 0.0 # форс-отсылка

def step_time(self):
    now = time.monotonic()
    if self.role == Role.LEADER:
        if now - self.last_hb_sent >= T_HB:
            self._send(Message('hb', self.term, self.id,
                               self._now_ns()))
            self.last_hb_sent = now
        return
    if now >= self.deadline:
        self.term += 1
        self.role = Role.CANDIDATE
        self.voted_for[self.term] = self.id
        self.votes = set()
        self._reset_timeout()
        print(f'{self._now_ns()} node{self.id} CANDIDATE '
              f'term={self.term}')
        self._send(Message('vote_req', self.term, self.id,
                           self._now_ns()))

def run(self):
    while True:
        self.step_rcv()
        self.step_time()
        time.sleep(TICK)

if __name__ == '__main__':
    me = int(sys.argv[1])
    base = 5000
    ports = {1: 5001, 2: 5002, 3: 5003}
    peers = [(i, p) for i, p in ports.items() if i != me]
    Node(me, ports[me], peers).run()

```

Шаг 3. Запуск стенда и эмуляция сбоя. В трёх терминалах запускаются процессы; через несколько секунд один из них становится лидером (по таймеру). Сбой лидера эмулируется SIGKILL.

Листинг 3 – Запуск трёх узлов и эмуляция сбоя лидера

```

# терминалы 1-3
python3 node.py 1
python3 node.py 2
python3 node.py 3

```

```

# терминал 4: убить текущего лидера (узнать PID через ps)

```

```
kill -9 $(pgrep -f 'node.py 1')
```

Шаг 4. Ожидаемый журнал событий. После запуска один из узлов выигрывает выборы ($\text{term} = 1$). После kill оставшиеся два обнаруживают пропуск heartbeat и за ~ 500 мс выбирают нового лидера с $\text{term} = 2$.

Листинг 4 – Журнал событий трёх узлов (выдержка, ts_ns в МОНОТОННЫХ нс)

```
12000123456 node1 CANDIDATE term=1
12000231047 node1 LEADER term=1
12100245001 node2 hb_received from=1 term=1
12100247811 node3 hb_received from=1 term=1
# ... lider node1 работает ...
30000000000 node1 KILLED (внешний SIGKILL)
30528104221 node2 CANDIDATE term=2
30531217540 node2 LEADER term=2
30531298110 node3 hb_received from=2 term=2
```

Шаг 5. Расчёт времени восстановления. По меткам ts_ns $t_{\text{recovery}} = 30531217540 - 30000000000 = 531217540$ нс ≈ 531 мс. Это согласуется с формулой (2): election_timeout случайно выпал ~ 520 мс, выборы заняли около 11 мс. Для отчёта по варианту запускается $N = 30$ повторов «убийства» лидера и строится гистограмма распределения t_{recovery} .

Задания по вариантам

Номер варианта определяется как $(N - 1) \bmod 15 + 1$, где N – порядковый номер студента в журнале группы. Конкретные числовые параметры задачи берутся из приведённой ниже таблицы. Составить отчет со скриншотами, показывающие выполнение заданий.

Вариант 1.

60. Развернуть базовый стенд из трёх узлов на одной машине (порты 5001-5003): $T_{\text{hb}} = 100$ мс, $T_{\text{to}} \in [400, 700]$ мс.

61. Реализовать скрипт-эксперимент, автоматически запускающий все три процесса, ждущий 2 с и убивающий текущего лидера; t_{recovery} считается по логам.

62. Прогнать 30 повторов; построить гистограмму распределения t_{recovery} с шагом 25 мс.

63. Сделать вывод о среднем и максимальном времени восстановления при базовых параметрах.

Вариант 2.

54. Уменьшить T_{hb} до 20 мс, $T_{\text{to}} \in [80, 150]$ мс.

55. Параллельно с тестом запустить нагрузку `stress-ng --cpu 4 --cpu-load 80`.

56. Снять 30 повторов и сравнить среднее t_{recovery} , а также долю ложных переизбраний (переход в CANDIDATE без реального сбоя) с вариантом 1.

57. Объяснить, при какой нагрузке короткие T_{to} становятся неустойчивыми.

Вариант 3.

55. Увеличить T_{hb} до 500 мс, $T_{\text{to}} \in [2000, 3500]$ мс.

56. Прогнать 30 повторов теста с убийством лидера; зафиксировать среднее t_{recovery} .

57. Построить график «среднее T_{to} vs среднее t_{recovery} » по трём точкам (варианты 1, 2, 3); подтвердить, что $t_{\text{recovery}} \approx T_{\text{to}}$.

Вариант 4.

55. Зафиксировать одинаковый $T_{\text{to}} = 500$ мс для всех трёх узлов (без рандомизации).

56. Прогнать 50 экспериментов с убийством лидера; подсчитать долю запусков со `split vote` (никто не набрал `majority` в первом раунде).

57. Включить рандомизацию обратно и сравнить долю `split vote`; обосновать необходимость случайного T_{to} .

Вариант 5.

55. Расширить стенд до пяти узлов на портах 5001-5005; `majority` = 3.

56. Прогнать 30 повторов с убийством лидера; сравнить среднее t_{recovery} со стендом $N = 3$.

57. Подсчитать число обменов `request_vote / vote_ok` в одном раунде выборов и сопоставить с теоретическим $N \cdot (N - 1)$.

Вариант 6.

54. Реализовать вариант транспорта на ZeroMQ: `heartbeat` – PUB/SUB, `request_vote / vote_ok` – REQ/REP.

55. Замерить число системных вызовов в секунду через `strace -s` для UDP- и ZeroMQ-вариантов на одинаковой нагрузке.

56. Сравнить среднее `t_recovery` по 30 повторам для обоих транспортов; объяснить, что даёт основной вклад в разницу.

Вариант 7.

56. Включить эмуляцию потерь: `sudo tc qdisc add dev lo root netem loss 20%`.

57. Подобрать минимальное `k` в формуле (1), при котором за 10 минут установившейся работы нет ложных переизбраний.

58. Снять зависимость `t_recovery` от $k \in \{3, 5, 10\}$ при тех же потерях; объяснить компромисс между чувствительностью и устойчивостью.

Вариант 8.

56. Внести сетевую задержку: `tc qdisc netem delay 80ms 20ms distribution normal`.

57. Подобрать минимальное `T_to_min`, при котором за 5 минут нет ложных переизбраний.

58. Прогнать тест с убийством лидера при найденном `T_to` и сравнить `t_recovery` со стендом без задержки.

Вариант 9.

57. Сценарий `network partition`: после выхода в установившийся режим командой `iptables -A INPUT -p udp --sport 5001 -j DROP` изолировать лидера от node-3 (но не от node-2).

58. Подтвердить по логам, что лидер сохраняется (его слышит node-2), а node-3 уходит в `Candidate` и не набирает `majority`.

59. Снять правило iptables и зафиксировать, как node-3 догоняет term и возвращается в Follower.

Вариант 10.

56. Реализовать сценарий двойного отказа: убить лидера; через $T_{to} + 200$ мс убить и нового лидера.

57. Прогнать 30 повторов; измерить суммарное время от первого kill до стабильного третьего лидера.

58. Объяснить, почему оставшийся одиночный узел не может избраться при $majority = 2$ из 3, и зафиксировать поведение по логам.

Вариант 11.

57. Реализовать сценарий восстановления: убить лидера, дождаться избрания нового, поднять прежний процесс с тем же node_id.

58. По логам нового процесса зафиксировать, через сколько hb он догоняет актуальный term и переходит в Follower.

59. Повторить эксперимент 10 раз; убедиться, что восстановленный узел никогда не остаётся в роли Candidate.

Вариант 12.

57. Добавить нагрузку клиентских команд: внешний скрипт шлёт лидеру 10 запросов/с, лидер реплицирует их в виде hb с payload.

58. Снять $t_{recovery}$ по 30 повторам убийства лидера при наличии клиентской нагрузки и без неё.

59. Подсчитать долю команд, потерянных при failover, и оценить, сколько подряд команд теряется при крупном T_{to} .

Вариант 13.

56. Заменить majority на полное согласие $quorum = N$ в коде Candidate.

57. Прогнать тест с убийством одного узла из трёх; зафиксировать, что выборы не проходят (никто не становится Leader).

58. Сделать вывод о нарушении liveness при таком правиле согласия.

Вариант 14.

- 57. Добавить персистентность `voted_for`: при каждом голосовании записывать json-файл с `os.fsync()`; при старте узла читать файл.
- 58. Снять рост `t_recovery` по 30 повторам по сравнению с in-memory вариантом.
- 59. Обосновать, почему именно `persistence voted_for` требуется для `safety`: продемонстрировать сценарий с двойным голосом при потере памяти.

Вариант 15.

- 57. Реализовать `pre-vote`: перед инкрементом `term` `Candidate` шлёт `pre_vote` и переходит в полные выборы только при получении `majority pre_vote_ok`.
- 58. Включить `netem loss 20 %` и прогнать тест на 10 минут.
- 59. Сравнить число ложных переизбраний с базовым вариантом (без `pre-vote`) при тех же потерях.

Лабораторная работа 15. Двухканальное резервирование (1oo2) на STM32 или симуляции

Цель работы: освоить принципы двухканального резервирования по схеме 1oo2 (one out of two) в распределённых системах управления. Реализовать симулятор, в котором два независимых канала (узла) опрашивают по сенсору, вычисляют управляющее решение и независимо отправляют его общему исполнительному устройству; голосование выполняется по правилу «среднее арифметическое при согласии», а при расхождении результатов сверх порога фиксируется сигнал диагностики и выполняется переход в безопасное состояние. Снять характеристики: вероятность ложного срабатывания диагностики при разных порогах, поведение при отказе одного канала, время обнаружения расхождения.

Теоретическое обоснование

1. Понятие и назначение схемы 1oo2. Резервирование 1oo2 относится к классу мажоритарных архитектур систем безопасности (Safety Instrumented Systems, SIS) по МЭК 61508 / 61511. Цифры в обозначении читаются как «один из двух»: для выработки команды достаточно сигнала ОДНОГО из двух каналов. Поэтому схема 1oo2 обладает высокой чувствительностью к опасному событию (низкая вероятность отказа на запрос PFD по опасной стороне), но более склонна к ложным срабатываниям (Spurious Trip Rate, STR), чем одноканальная схема, поскольку для ложного срабатывания достаточно отказа любого из двух каналов. В литературе наряду с 1oo2 рассматривают 2oo2 (требуется согласие обоих, минимальные ложные срабатывания, но низкая безопасность) и 2oo3 (компромисс: переживает один отказ без потери ни безопасности, ни доступности).

2. Канал и его компоненты. Канал – независимая цепочка «сенсор -> вычислитель -> выходной интерфейс», физически разделённая с соседним каналом: отдельное питание, отдельный процессор, отдельный кабель сенсора. Цель разделения – исключить общую причину отказа (common cause

failure, CCF), вероятность которой характеризуется коэффициентом β (доля коррелированных отказов). Программная диверсификация (разные алгоритмы фильтрации, разные компиляторы, иногда разные команды-разработчики) и аппаратная (сенсоры разных производителей) дополнительно снижают β . Для упрощённой оценки PFD двухканальной схемы используется приближённая формула:

$$PFD_{1oo2} \approx (1 - \beta) \cdot (\lambda_{DU} \cdot T_{PT})^2 / 3 + \beta \cdot \lambda_{DU} \cdot T_{PT} / 2 \quad (1)$$

Здесь λ_{DU} – интенсивность опасных необнаруженных отказов канала; T_{PT} – интервал между диагностическими проверками (proof test). Видно, что при отсутствии CCF ($\beta = 0$) PFD падает квадратично с уменьшением $\lambda_{DU} \cdot T_{PT}$, что и даёт основной выигрыш дублирования; коэффициент β задаёт «дно» – неустранимую часть, общую для обоих каналов.

3. Правило голосования и сравнение значений. Когда оба канала вырабатывают не дискретную команду (вкл/выкл), а аналоговую уставку u_i , выполняется не логическое «1 из 2», а проверка согласованности значений с последующим усреднением. Правило, реализуемое в данной работе:

$$|u_1 - u_2| \leq \Delta_{tol} \rightarrow u_{out} = (u_1 + u_2) / 2 \quad (2)$$

$$|u_1 - u_2| > \Delta_{tol} \rightarrow DIAG = 1, u_{out} = SAFE \quad (3)$$

Порог расхождения Δ_{tol} выбирается заведомо больше типичного шума сенсора, но заметно меньше величины, при которой различие выходов уже опасно. На практике Δ_{tol} задают как 3-5 $\sigma_{шум}$ либо как фиксированный процент от диапазона. Безопасное значение SAFE определяется природой объекта: для нагревателя – нулевая уставка мощности, для клапана аварийного сброса – полное открытие, для исполнительного механизма позиции – удержание последнего «согласованного» значения.

4. Таблица голосования сведена в таблицу 1. В ней рассматривается, какое решение принимает арбитр в зависимости от состояния каналов: оба

живы и согласны, оба живы и расходятся, один отказал по таймауту (нет сообщения за интервал T_{alive}).

Таблица 1 – Правила голосования арбитра 1oo2 (среднее при согласии)

Состояние канала 1	Состояние канала 2	Условие на u	u_{out}	DIAG
Активе н	Активе н	$ u_1 - u_2 \leq \Delta_{\text{tol}}$	$(u_1 + u_2) / 2$	0
Активе н	Активе н	$ u_1 - u_2 > \Delta_{\text{tol}}$	SAFE	1 (mismatch)
Активе н	Таймаут т	u_1 задано	u_1 (degraded)	2 (one-channel)
Таймаут т	Активе н	u_2 задано	u_2 (degraded)	2 (one-channel)
Таймаут т	Таймаут т	нет данных	SAFE	3 (loss of both)

5. Распределённая реализация. В этой работе каналы выполняются как два независимых процесса, общающихся с арбитром по UDP. Каждый канал передаёт пакет (channel_id , seq , ts_ns , u_i , health). Арбитр поддерживает скользящее окно последних значений от каждого канала и выполняет голосование по правилам (2)-(3) на каждом тике с фиксированным периодом T_{ctrl} . Если за интервал $T_{\text{alive}} = k \cdot T_{\text{ctrl}}$ от канала не пришло ни одного пакета, канал помечается как «таймаут». Все события (валидное голосование, mismatch, потеря канала, восстановление) логируются с монотонной меткой времени для последующего анализа.

Пример выполнения задания

В качестве разбираемого «нулевого» варианта рассмотрим стенд «регулятор температуры»: два канала измеряют один и тот же физический сигнал (синтетическая температура 70°C с независимым гауссовым шумом $\sigma = 0.3^\circ\text{C}$), вычисляют уставку мощности пропорционально отклонению от

75 °C и отправляют арбитру с периодом $T_{ctrl} = 50$ мс. Порог расхождения $\Delta_{tol} = 2$ (в процентах мощности), таймаут $T_{alive} = 200$ мс. $SAFE = 0$ %.

Шаг 1. Канал. Каждый процесс-канал генерирует свой шум, вычисляет $u_i = K_p \cdot (T_{set} - T_i)$, ограничивает уставку диапазоном $[0, 100]$ % и шлёт пакет арбитру.

Листинг 1 – Процесс канала 1oo2 (channel.py)

```
import socket, json, time, random, sys

CH_ID  = int(sys.argv[1])    # 1 | 2
ARB_ADDR = ('127.0.0.1', 6000)
T_CTRL  = 0.050              # 50 мс
T_SET   = 75.0
T_TRUE  = 70.0
SIGMA   = 0.3
KP      = 8.0

def clamp(x, lo=0.0, hi=100.0):
    return max(lo, min(hi, x))

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
seq = 0
while True:
    t_meas = T_TRUE + random.gauss(0.0, SIGMA)
    u = clamp(KP * (T_SET - t_meas))
    pkt = {'ch': CH_ID, 'seq': seq, 'ts_ns': time.monotonic_ns(),
           'u': u, 'health': 'ok'}
    sock.sendto(json.dumps(pkt).encode(), ARB_ADDR)
    seq += 1
    time.sleep(T_CTRL)
```

Шаг 2. Арбитр. Слушает порт 6000, держит последнее пришедшее значение от каждого канала и метку времени; каждые T_{ctrl} выполняет голосование.

Листинг 2 – Арбитр голосования 1oo2 (arbiter.py)

```
import socket, json, time

PORT = 6000
T_CTRL = 0.050
T_ALIVE_NS = 200_000_000    # 200 мс
DELTA_TOL = 2.0             # %
SAFE      = 0.0             # %
```

```

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind(('127.0.0.1', PORT))
sock.setblocking(False)

```

```

last = {1: None, 2: None} # ch -> (ts_ns, u)
next_tick = time.monotonic()
while True:
    # 1. приём
    while True:
        try:
            raw, _ = sock.recvfrom(4096)
        except BlockingIOError:
            break
        m = json.loads(raw.decode())
        last[m['ch']] = (time.monotonic_ns(), m['u'])
    # 2. голосование на тике
    if time.monotonic() >= next_tick:
        now = time.monotonic_ns()
        alive = {}
        for ch, item in last.items():
            if item is None:
                continue
            ts, u = item
            if now - ts <= T_ALIVE_NS:
                alive[ch] = u
        if len(alive) == 2:
            u1, u2 = alive[1], alive[2]
            if abs(u1 - u2) <= DELTA_TOL:
                u_out, diag = (u1 + u2) / 2.0, 0
            else:
                u_out, diag = SAFE, 1
        elif len(alive) == 1:
            u_out, diag = next(iter(alive.values())), 2
        else:
            u_out, diag = SAFE, 3
        print(f'{now} u_out={u_out:6.2f} diag={diag} '
              f'have={sorted(alive)}')
        next_tick += T_CTRL
    time.sleep(0.005)

```

Шаг 3. Запуск стенда.

Листинг 3 – Запуск арбитра и двух каналов

```

python3 arbiter.py &
python3 channel.py 1 &
python3 channel.py 2 &

```

Шаг 4. Ожидаемый вывод в установившемся режиме. Оба канала согласованы в пределах шума, $diag = 0$, u_out близок к $(75 - 70) \cdot 8 = 40 \%$.

Листинг 4 – Журнал арбитра в установившемся режиме (выдержка)

```
41234567890 u_out= 39.84 diag=0 have=[1, 2]
41284567950 u_out= 40.21 diag=0 have=[1, 2]
41334568010 u_out= 39.97 diag=0 have=[1, 2]
41384568070 u_out= 40.08 diag=0 have=[1, 2]
```

Шаг 5. Эмуляция расхождения. В одном из каналов программно добавляется смещение +10 %; ожидаем $\text{diag} = 1$ и $u_{\text{out}} = \text{SAFE}$.

Листинг 5 – Журнал арбитра при искусственном расхождении канала 2

```
42384568430 u_out= 40.05 diag=0 have=[1, 2]
42434568490 u_out= 0.00 diag=1 have=[1, 2]    # mismatch
42484568550 u_out= 0.00 diag=1 have=[1, 2]
# время от события (момент скачка) до первой строки diag=1
# составило T_ctrl = 50 мс
```

Шаг 6. Эмуляция отказа канала. Канал 2 убивается; через 200 мс (T_{alive}) арбитра переключается в degraded-режим с одним каналом и $\text{diag} = 2$.

Листинг 6 – Журнал арбитра после kill -9 channel.py 2

```
43384568910 u_out= 39.92 diag=0 have=[1, 2]
43434568970 u_out= 40.11 diag=0 have=[1, 2]
43484569030 u_out= 40.07 diag=2 have=[1]
43534569090 u_out= 39.98 diag=2 have=[1]
```

Шаг 7. Метрики для отчёта. По логу длительностью не менее 10 минут считаются: средний и максимальный модуль $|u_1 - u_2|$; число тиков с $\text{diag} = 1$ при отсутствии искусственного расхождения (false-mismatch rate); среднее время от момента инъекции расхождения до первого $\text{diag} = 1$ (должно быть $\leq T_{\text{ctrl}}$); среднее время от kill канала до перехода в $\text{diag} = 2$ (должно быть $\leq T_{\text{alive}}$).

Задания по вариантам

Номер варианта определяется как $(N - 1) \bmod 15 + 1$, где N – порядковый номер студента в журнале группы. Конкретные числовые параметры задачи берутся из приведённой ниже таблицы. Составить отчет со скриншотами, показывающие выполнение заданий.

Вариант 1.

- 64. Развернуть базовый стенд: arbiter.py и два канала с $\sigma = 0.3$ °C, $\Delta_{tol} = 2$ %, $T_{ctrl} = 50$ мс на Ubuntu 24.04.
- 65. Снять журнал арбитра длительностью 10 минут.
- 66. Подсчитать false-mismatch rate (доля тиков с $diag = 1$ в отсутствие реального расхождения) и среднее $|u_1 - u_2|$.
- 67. Построить гистограмму $|u_1 - u_2|$ с шагом 0.1 %.

Вариант 2.

- 58. Увеличить σ канала 1 до 1.0 при $\Delta_{tol} = 2$ %; снять 10 мин журнала.
- 59. Объяснить рост false-mismatch rate и сравнить со значением из варианта 1.
- 60. Подобрать новый Δ_{tol} , возвращающий долю ложных диагнозов ниже 0.1 %; зафиксировать значение.

Вариант 3.

- 58. Уменьшить T_{ctrl} до 10 мс на каналах и арбитра.
- 59. Через top -p оценить рост загрузки CPU арбитра и каналов.
- 60. Сравнить СКО u_{out} и false-mismatch rate с базовым вариантом; сделать вывод, окупается ли учащение T_{ctrl} .

Вариант 4.

- 58. Изменить логику SAFE на «удержание последнего согласованного u_{out} » (last good).
- 59. Прогнать сценарий с инъекцией mismatch на 2 с и сравнить профиль $u_{out}(t)$ с базовым (SAFE = 0).

60. Сделать вывод, в каких приложениях last good безопаснее, а в каких – опаснее.

Вариант 5.

58. Внести систематический bias канала 2: $u_2 := u_2 + \text{bias}$, где bias линейно растёт от 0 до 5 % за 100 с.

59. Зафиксировать момент времени, когда впервые срабатывает diag = 1.

60. Сопоставить найденный bias с порогом Δ_{tol} с учётом шума 3σ .

Вариант 6.

57. Реализовать программную диверсификацию: канал 1 – чистый P-регулятор, канал 2 – PI-регулятор с $T_i = 5$ с.

58. Подать на оба канала ступеньку уставки T_{set} : 75 $\rightarrow$ 80 °C.

59. Подобрать минимальное Δ_{tol} , при котором переходный процесс не вызывает срабатывания diag = 1.

Вариант 7.

59. Реализовать аппаратную диверсификацию: канал 1 моделирует термопару (шум $\sigma = 0.5$ °C, дрейф 0.01 °C/мин), канал 2 – RTD ($\sigma = 0.1$ °C, дрейф 0.002 °C/мин).

60. Снять 30 мин журнала и сравнить среднее $|u_1 - u_2|$ со случаем «два одинаковых канала».

61. Обосновать, как разнородные сенсоры влияют на коэффициент β из формулы (1).

Вариант 8.

59. Запустить каналы и арбитр на разных машинах в одной сети; на канале 2 включить `tc qdisc add dev eth0 root netem delay 30ms`.

60. Снять график $|u_1 - u_2|(t)$ при медленно изменяющейся уставке (T_{set} меняется на 0.05 °C/с).

61. Объяснить связь дрейфа $|u_1 - u_2|$ с задержкой канала и скоростью изменения сигнала.

Вариант 9.

- 60. Канал 1 шлёт пакеты с пропусками: каждый второй пакет не отправляется (моделирует частичную потерю связи).
- 61. Подобрать T_{alive} , при котором канал ещё считается живым.
- 62. Снять журнал и убедиться, что $diag$ не выходит за пределы $\{0, 2\}$; зафиксировать долю $diag = 2$.

Вариант 10.

- 59. Реализовать в арбитраже debounce: переход в SAFE только после M подряд тиков с mismatch.
- 60. Подобрать минимальное M , при котором случайные выбросы шума не вызывают срабатывания за 10 мин.
- 61. Прогнать сценарий с реальным расхождением (bias 5 % на канале 2) и измерить рост задержки обнаружения относительно $M = 1$.

Вариант 11.

- 60. Изменить правило усреднения с $(u_1 + u_2) / 2$ на $\min(u_1, u_2)$ (fail-safe для нагревателя).
- 61. Прогнать сценарий нормы и сценарий «канал 2 завис на $u_2 = 100\%$ »; снять профили $u_{out}(t)$.
- 62. Сделать вывод, при каком классе актуаторов $\min$ является fail-safe.

Вариант 12.

- 60. Изменить правило на $\max(u_1, u_2)$.
- 61. Прогнать сценарий нормы и сценарий «канал 2 завис на $u_2 = 0\%$ »; снять профили $u_{out}(t)$.
- 62. Указать класс актуаторов (например, клапан аварийного сброса), для которых $\max$ является fail-safe.

Вариант 13.

- 59. Добавить третий процесс-канал; в арбитраже реализовать голосование 2oo3 (медиана).

60. Снять 10 мин журнала при тех же шумах $\sigma = 0.3$ °C и сравнить false-mismatch rate с 1002.

61. Прогнать сценарий отказа одного канала и убедиться, что u_{out} не уходит в SAFE (medium из двух оставшихся).

Вариант 14.

60. Снять выборку $\Delta = u_1 - u_2$ за 30 минут (около 36 000 точек).

61. Построить гистограмму Δ и оценить эмпирическое значение 6σ .

62. Сравнить полученное 6σ с порогом $\Delta_{tol} = 2$ %; обосновать, корректен ли выбор Δ_{tol} в смысле « 3σ выше уровня шума».

Вариант 15.

60. По формуле (1) рассчитать PFD_1002 при $\lambda_{DU} = 1e-6$ 1/час, $T_{PT} = 4380$ ч (полгода) для $\beta \in \{0.01, 0.05, 0.10\}$.

61. Сравнить с PFD одноканальной схемы $\lambda_{DU} \cdot T_{PT} / 2$.

62. Построить график PFD(β) и определить, начиная с какого β выигрыш дублирования становится менее одного порядка.

Лабораторная работа 16. Распределённая транзакция: Saga (хореография)

Цель работы: освоить реализацию распределённой транзакции по образцу Saga (хореографический вариант) в системе из трёх независимых сервисов, обменивающихся событиями через middleware DDS. Реализовать последовательное резервирование трёх ресурсов А, В, С по событиям Reserved; при отказе любого из шагов организовать каскадную компенсацию (откат) уже выполненных шагов через события Failed / Compensated. Снять временные характеристики: время успешной саги, время компенсации при отказе на разных шагах, гарантию того, что после завершения саги ни один ресурс не остался в зарезервированном состоянии при неуспехе.

Теоретическое обоснование

1. Проблема распределённой транзакции и место Saga. Когда бизнес-операция требует согласованного изменения данных в нескольких независимых сервисах, классический подход – двухфазная фиксация (2PC) – обеспечивает атомарность за счёт блокировки ресурсов на время голосования. В средах с длительными операциями или ненадёжными каналами связи блокировка превращается в узкое место и при отказе координатора может привести к зависанию сервисов (in-doubt state). Альтернативный подход Saga, предложенный Н. Garcia-Molina и К. Salem в 1987 году, отказывается от строгой атомарности в пользу слабой согласованности «либо все шаги успешно выполнены, либо все ранее выполненные шаги компенсированы обратными операциями». Каждый шаг T_i имеет парную операцию компенсации C_i , удовлетворяющую условию семантического отката: выполнение T_i ; C_i оставляет систему в исходном состоянии. Гарантия Saga – не serializability, а «либо happy path $T_1 T_2 \dots T_n$, либо префикс $T_1 \dots T_k$ с компенсацией $C_k C_{(k-1)} \dots C_1$ ».

2. Хореография vs оркестрация. Saga реализуется в двух стилях. В оркестрации существует выделенный координатор (Saga Orchestrator),

который сам отправляет команды каждому участнику и собирает ответы – фактически конечный автомат, ведущий состояние транзакции. В хореографии (рассматривается в данной работе) выделенного координатора нет: каждый сервис подписан на события общего шинного протокола; получение одного события автоматически становится триггером следующего шага у соответствующего участника. Преимущество хореографии – отсутствие единой точки отказа и простота добавления новых шагов; недостаток – сложнее отлаживать (нет «карты» саги в одном месте) и важно тщательно проектировать множество событий, чтобы избежать гонок.

3. Событийная модель и темы DDS. Реализация на DDS использует одну тему-«шину» Saga с типом сообщения SagaEvent. Каждый сервис публикует и подписывается на эту тему; фильтрация выполняется по полю kind. Минимальный набор событий приведён в таблице 1. В поле saga_id хранится идентификатор экземпляра транзакции (UUIDv4), что позволяет одновременно выполнять несколько саг.

Таблица 1 – События хореографической саги (тема DDS «Saga»)

kind	Издатель	Реакция подписчиков
SagaStart	Initiator	Service A резервирует ресурс и публикует A.Reserved
A.Reserved	Service A	Service B пробует зарезервировать; B.Reserved / B.Failed
B.Reserved	Service B	Service C пробует зарезервировать; C.Reserved / C.Failed
C.Reserved	Service C	Initiator публикует SagaCommitted
X.Failed	Любой сервис	Initiator публикует SagaAborted
SagaAborted	Initiator	Все, кто резервировал, публикуют X.Compensated
SagaCompleted	Initiator	Сага завершена; чтение журнала

kind	Изда тель	Реакция подписчиков
mitted	tor	может быть выгружено

4. Свойства корректности. Хорошо спроектированная Saga обеспечивает свойства: (а) если сервисы выполнили все шаги Reserved, рано или поздно публикуется SagaCommitted; (б) если хоть один шаг завершился X.Failed, ни один из ресурсов не остаётся в зарезервированном состоянии (eventually consistent rollback); (в) компенсирующие операции идемпотентны (повторное получение SagaAborted не должно ломать сервис, который уже откатился). Идемпотентность достигается хранением статуса по saga_id: при повторном событии сервис просто отвечает уже опубликованным результатом.

5. QoS DDS для надёжного обмена событиями. Для саги критична доставка каждого события до всех подписчиков; потеря SagaAborted, например, оставит участников с зарезервированным ресурсом. На теме Saga применяются политики RELIABILITY = RELIABLE (повторная отправка до подтверждения), DURABILITY = TRANSIENT_LOCAL (новый подписчик получает уже опубликованные сообщения), HISTORY = KEEP_LAST с глубиной не меньше числа ожидаемых событий саги. Поле saga_id отмечается как ключ темы (@key в IDL), что позволяет DDS поддерживать отдельную историю и порядок для каждой саги.

6. Сценарии успешного выполнения и компенсации приведены на рисунке 1. На левой панели – «happy path»: SagaStart запускает цепочку A -> B -> C, после C.Reserved Initiator подтверждает успех. На правой – отказ на шаге B: после B.Failed Initiator публикует SagaAborted, в ответ на которое все, кто успел зарезервировать (в примере – только A), публикуют X.Compensated; сервисы, которые ничего не резервировали, отвечают по-ор.

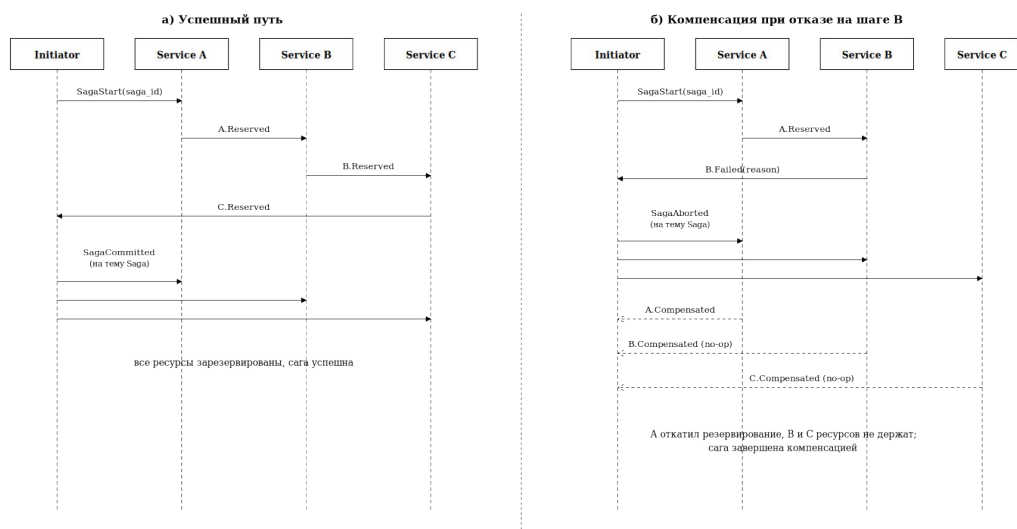


Рисунок 1 – Хореографическая Saga: успешный путь и компенсация при отказе

Пример выполнения задания

В качестве разбираемого «нулевого» варианта рассмотрим три сервиса A, B, C и инициатор Initiator, моделирующие резервирование пакета: A – место в авиарейсе, B – номер в отеле, C – трансфер. Все процессы – Python с библиотекой fastdds 3.x (eProsima Fast DDS, namespace eprosima::fastdds::dds; при необходимости стенд воспроизводится и на Eclipse Cyclone DDS). Транспорт – DDS, тема – Saga, тип сообщения описан в IDL. Отказ моделируется случайной величиной: с вероятностью p_{fail} сервис публикует X.Failed вместо X.Reserved. ОС стенда – Ubuntu 24.04 LTS.

Шаг 1. Описание типа сообщения в IDL. Поле `saga_id` помечается как ключ темы, чтобы каждая сага имела свою историю.

Листинг 1 – Определение типа сообщения SagaEvent (saga.idl)

```
module saga {
    struct SagaEvent {
        @key string saga_id;
        string kind;      // 'SagaStart' | 'A.Reserved' | ...
        string sender;    // 'A' | 'B' | 'C' | 'Initiator'
        long long ts_ns;
        string payload;   // JSON, опционально
    };
};
```

Шаг 2. Базовый класс сервиса. Открывает Participant, Publisher и Subscriber на тему Saga с указанными QoS; обрабатывает входящие события через метод on_event, который переопределяется потомками.

Листинг 2 – Базовый класс сервиса хореографической саги (saga_node.py, Fast DDS 3.x)

```
import time, json, uuid
# Fast DDS 3.x: пакет fastdds, namespace eprosima::fastdds::dds
from fastdds import (DomainParticipantFactory, TopicQos,
                     DataReaderQos, DataWriterQos,
                     ReliabilityQosPolicyKind,
                     DurabilityQosPolicyKind)
from saga_types import SagaEvent, SagaEventPubSubType

class SagaNode:
    def __init__(self, name):
        self.name = name
        self.state = {} # saga_id -> 'reserved' | 'failed' | ...
        f = DomainParticipantFactory.get_instance()
        self.dp = f.create_participant(0)
        self.type = SagaEventPubSubType()
        self.dp.register_type(self.type, 'SagaEvent')
        tq = TopicQos()
        self.topic = self.dp.create_topic('Saga', 'SagaEvent', tq)
        # QoS: RELIABLE + TRANSIENT_LOCAL
        rq = DataReaderQos()
        rq.reliability().kind = (
            ReliabilityQosPolicyKind.RELIABLE_RELIABILITY_QOS)
        rq.durability().kind = (
            DurabilityQosPolicyKind.TRANSIENT_LOCAL_DURABILITY_QOS)
        wq = DataWriterQos()
        wq.reliability().kind = (
            ReliabilityQosPolicyKind.RELIABLE_RELIABILITY_QOS)
        wq.durability().kind = (
            DurabilityQosPolicyKind.TRANSIENT_LOCAL_DURABILITY_QOS)
        sub = self.dp.create_subscriber()
        pub = self.dp.create_publisher()
        self.reader = sub.create_datareader(self.topic, rq, self)
        self.writer = pub.create_datawriter(self.topic, wq)

    def publish(self, kind, saga_id, payload=""):
        ev = SagaEvent()
        ev.saga_id = saga_id
        ev.kind = kind
        ev.sender = self.name
        ev.ts_ns = time.monotonic_ns()
        ev.payload = payload
        self.writer.write(ev)
```

```

print(f'{ev.ts_ns} {self.name} -> {kind} saga={saga_id}')

def on_data_available(self, reader):
    ev, info = SagaEvent(), reader.take_next_sample(...)
    if info.valid_data:
        self.on_event(ev)

def on_event(self, ev):
    raise NotImplementedError

```

Шаг 3. Сервис-резервист (А, В, С). Реагирует на «предыдущее» событие в цепочке: А на SagaStart, В на A.Reserved, С на B.Reserved. На SagaAborted публикует X.Compensated, если ранее был в состоянии reserved.

Листинг 3 – Сервис резервирования ресурса (worker.py)

```

import random, sys
from saga_node import SagaNode

TRIGGER = {'A': 'SagaStart',
           'B': 'A.Reserved',
           'C': 'B.Reserved'}

class Worker(SagaNode):
    def __init__(self, name, p_fail=0.0):
        super().__init__(name)
        self.p_fail = p_fail

    def on_event(self, ev):
        sid = ev.saga_id
        if ev.kind == TRIGGER[self.name]:
            if sid in self.state:
                return # идемпотентность
            if random.random() < self.p_fail:
                self.state[sid] = 'failed'
                self.publish(f'{self.name}.Failed', sid,
                           '{"reason":"injected"}')
            else:
                self.state[sid] = 'reserved'
                self.publish(f'{self.name}.Reserved', sid)
        elif ev.kind == 'SagaAborted':
            if self.state.get(sid) == 'reserved':
                self.state[sid] = 'compensated'
                self.publish(f'{self.name}.Compensated', sid)
            else:
                self.publish(f'{self.name}.Compensated', sid,
                           '{"noop":true}')

if __name__ == '__main__':

```

```
Worker(sys.argv[1], p_fail=float(sys.argv[2])).run_forever()
```

Шаг 4. Инициатор. Стартует сагу, ждёт либо C.Reserved (тогда публикует SagaCommitted), либо X.Failed от любого сервиса (тогда публикует SagaAborted). Замеряет время от SagaStart до терминального события и пишет в журнал.

Листинг 4 – Инициатор саги (initiator.py)

```
import uuid, time
from saga_node import SagaNode

class Initiator(SagaNode):
    def __init__(self):
        super().__init__('Initiator')
        self.start_ts = {} # saga_id -> ts

    def start(self):
        sid = str(uuid.uuid4())
        self.start_ts[sid] = time.monotonic_ns()
        self.publish('SagaStart', sid)
        return sid

    def on_event(self, ev):
        sid = ev.saga_id
        if ev.kind == 'C.Reserved':
            dt = (time.monotonic_ns() - self.start_ts[sid]) / 1e6
            self.publish('SagaCommitted', sid)
            print(f'SAGA {sid} COMMIT in {dt:.2f} ms')
        elif ev.kind.endswith('.Failed'):
            self.publish('SagaAborted', sid,
                        f'{{{ "failed_at": "{ev.sender}" }}}')
            dt = (time.monotonic_ns() - self.start_ts[sid]) / 1e6
            print(f'SAGA {sid} ABORT after {dt:.2f} ms')
```

Шаг 5. Запуск стенда. Три worker-а с разной вероятностью отказа и initiator, генерирующий 100 саг с интервалом 50 мс.

Листинг 5 – Запуск стенда (4 терминала или фоновых процесса)

```
python3 worker.py A 0.0 &
python3 worker.py B 0.2 &
python3 worker.py C 0.0 &
python3 initiator_run.py 100
```

Шаг 6. Ожидаемый журнал успешной саги. На сервисах А, В, С последовательно появляются Reserved, инициатор фиксирует Committed примерно за 15-30 мс.

Листинг 6 – Журнал успешной саги (выдержка)

```
100000000 Initiator -> SagaStart saga=4f...
100002145 A -> A.Reserved saga=4f...
100004810 B -> B.Reserved saga=4f...
100007121 C -> C.Reserved saga=4f...
100008900 Initiator -> SagaCommitted saga=4f...
SAGA 4f... COMMIT in 8.90 ms
```

Шаг 7. Ожидаемый журнал саги с компенсацией (отказ на В).

Листинг 7 – Журнал саги с отказом на шаге В

```
200000000 Initiator -> SagaStart saga=9c...
200002270 A -> A.Reserved saga=9c...
200004621 B -> B.Failed saga=9c...
200005982 Initiator -> SagaAborted saga=9c...
200008441 A -> A.Compensated saga=9c...
200008720 C -> C.Compensated (noop) saga=9c...
SAGA 9c... ABORT after 5.98 ms
```

Шаг 8. Метрики для отчёта. По журналу 100 саг считаются: доля успешных и компенсированных, среднее и СКО времени успешной саги, среднее время от X.Failed до полной компенсации (последнее Compensated), проверка инварианта «после ABORT у каждого сервиса state[saga_id] != reserved», подсчёт сообщений на тему Saga в Wireshark / счётчиках DDS для оценки сетевой нагрузки.

Задания по вариантам

Номер варианта определяется как $(N - 1) \bmod 15 + 1$, где N – порядковый номер студента в журнале группы. Конкретные числовые параметры задачи берутся из приведённой ниже таблицы. Составить отчет со скриншотами, показывающие выполнение заданий.

Вариант 1.

- 68. Развернуть стенд из примера на Ubuntu 24.04 с Fast DDS 3.x: три worker-a и initiator, $p_fail = 0$ для всех.
- 69. Прогнать 100 саг подряд с интервалом 50 мс между запусками.
- 70. Снять среднее, СКО, минимум и максимум времени happy-path; построить гистограмму.
- 71. Сделать вывод о типовых задержках хореографической саги на локальном стенде.

Вариант 2.

- 61. Поднять стенд при $p_fail = 0.3$ на сервисе А; В и С – без сбоев.
- 62. Прогнать 100 саг и посчитать долю компенсированных.
- 63. Снять среднее время от SagaStart до SagaAborted; сравнить его с happy-path из варианта 1.
- 64. Подтвердить инвариант: после SagaAborted ни у одного worker-a в state нет 'reserved'.

Вариант 3.

- 61. $p_fail = 0.3$ на сервисе В; А и С – без сбоев.
- 62. Прогнать 100 саг; зафиксировать долю компенсированных и среднее время до полной компенсации (последнее .Compensated у А).
- 63. Сравнить с вариантом 2: показать, что число X.Compensated-сообщений выросло, потому что компенсация теперь требуется и для А.

Вариант 4.

- 61. $p_fail = 0.3$ на сервисе С; А и В – без сбоев.

62. Прогнать 100 саг; зафиксировать, что при отказе на С компенсируются оба предыдущих шага А и В.

63. Подсчитать суммарное число сообщений в теме Saga на одну сагу с компенсацией и сравнить с happy-path.

Вариант 5.

61. Установить $p_fail = 0.1$ на всех трёх сервисах.

62. Прогнать 500 саг и измерить экспериментальную долю успешных.

63. Сопоставить с теоретическим значением $(1 - p)^3 \approx 0.729$; оценить статистическую погрешность.

Вариант 6.

60. Изменить DURABILITY с TRANSIENT_LOCAL на VOLATILE на теме Saga для DataReader-a worker-a А.

61. Запустить initiator на 2 с раньше worker-a А; зафиксировать, что SagaStart, опубликованный до подъёма А, теряется.

62. Вернуть TRANSIENT_LOCAL и повторить эксперимент; убедиться, что SagaStart доставляется новому подписчику.

Вариант 7.

62. Изменить RELIABILITY на BEST_EFFORT для всех сущностей темы Saga.

63. Включить эмуляцию потерь: `sudo tc qdisc add dev lo root netem loss 5%`.

64. Прогнать 200 саг и подсчитать долю «зависших» (без терминального Committed или Aborted за 5 с).

65. Вернуть RELIABLE и повторить; сравнить долю зависших.

Вариант 8.

62. Добавить четвёртый сервис D в цепочку с триггером C.Reserved.

63. Обновить initiator: SagaCommitted публикуется после получения D.Reserved.

64. Прогнать 100 саг при $p_fail = 0$ и сравнить среднее время happy-path с базовым 3-шаговым стендом.

Вариант 9.

63. Реализовать оркестрационный вариант: вынести КА саги в отдельный процесс SagaOrchestrator, шлющий команды через тему Saga.Cmd и принимающий ответы через Saga.Reply.

64. Прогнать 100 саг для оркестрации и для хореографии.

65. Подсчитать общее число DDS-сообщений на сагу в обоих вариантах и сравнить.

Вариант 10.

62. Добавить в initiator таймаут на каждый шаг (200 мс на ответ Reserved).

63. Сэмулировать отказ: на сервисе В при получении триггера не публиковать ничего.

64. Зафиксировать, что initiator публикует SagaAborted по таймауту даже без X.Failed; снять долю аборта по таймауту.

Вариант 11.

63. Модифицировать initiator: одновременно стартует $N = 50$ саг с интервалом 5 мс.

64. Прогнать стенд; проверить корректность изоляции по saga_id (в каждой саге фигурирует ровно три Reserved и один Committed).

65. Измерить общую пропускную способность системы (саг в секунду) и среднее время одной саги под параллельной нагрузкой.

Вариант 12.

63. В initiator реализовать повторную публикацию SagaAborted через 100 мс после первой (имитация повторной доставки).

64. По логам worker-а А убедиться, что состояние не переоткатывается (статус остаётся 'compensated').

65. Снять, сколько раз worker реально шлёт X.Compensated при повторных SagaAborted.

Вариант 13.

62. Реализовать частичный отказ компенсации: worker A с вероятностью 0.5 не публикует A.Compensated.

63. В initiator добавить retry: переотправка SagaAborted каждые 200 мс до получения всех X.Compensated.

64. Прогнать 100 саг при $p_{fail} = 0.3$ на B и измерить среднее число retry до полной компенсации.

Вариант 14.

63. Развернуть стенд на двух машинах в одной подсети: A и B на одной ВМ, C и initiator – на второй.

64. Прогнать 100 саг happy-path; снять среднее и СКО времени.

65. Сравнить с локальным стендом из варианта 1 и оценить долю сетевой задержки в общем времени саги.

Вариант 15.

63. Реализовать упрощённый 2PC: SagaCoordinator рассылает Prepare всем участникам, ждёт всех Prepared, шлёт Commit; при таймауте Prepared – Abort.

64. Прогнать 100 транзакций при $p_{fail} = 0.3$ на B на сценарии Prepare.

65. Измерить время удержания «блокировки ресурса» у A с момента Prepared до Abort; сравнить с временем от A.Reserved до A.Compensated у Saga.

Лабораторная работа 17. Интеграция DDS с ROS 2 (распределённая робототехника)

Цель работы: освоить интеграцию промежуточного слоя Data Distribution Service (DDS) с фреймворком ROS 2 Jazzy Jalisco на примере двухмашинной распределённой системы. Развернуть ROS 2 на двух виртуальных машинах под управлением Ubuntu 24.04, переключить RMW-реализацию на eProsima Fast DDS 3.x, запустить пару узлов talker / listener на разных хостах, проверить процесс обнаружения (discovery), настроить профили QoS, измерить сквозную задержку доставки сообщений и научиться диагностировать состояние графа с помощью утилит `ros2 daemon`, `ros2 doctor`, `ros2 topic info -v`.

Теоретическое обоснование

1. Архитектура коммуникаций ROS 2. В отличие от ROS 1, где все узлы регистрируются у центрального процесса `roscore`, в ROS 2 коммуникация полностью децентрализована: используется промежуточный слой DDS (Data Distribution Service) – стандарт OMG, описывающий публикацию–подписку с богатым набором политик качества обслуживания (Quality of Service, QoS). Каждый узел встраивает в себя DDS DomainParticipant, и узлы обнаруживают друг друга самостоятельно через протокол Simple Participant Discovery Protocol (SPDP) – multicast-объявления на адрес 239.255.0.1, порт 7400 + 250·domain_id.

Между прикладным API ROS 2 (`rclcpp` / `rclpy`) и конкретной реализацией DDS лежит абстракция ROS Middleware Interface (RMW). RMW – это тонкий C-API, позволяющий подменять реализацию DDS без перекомпиляции прикладных узлов. В дистрибутиве Jazzy Jalisco штатно поддерживаются три реализации: `rmw_fastrtps_cpp` (eProsima Fast DDS 3.x, поставляется по умолчанию через пакет `ros-jazzy-rmw-fastrtps-cpp`), `rmw_cyclonedds_cpp` (Eclipse Cyclone DDS) и `rmw_connextdds` (RTI Connext,

требует отдельной лицензии). Реализация выбирается переменной окружения `RMW_IMPLEMENTATION` на момент запуска каждого узла.

2. Понятие домена и изоляция трафика. Все участники DDS объединяются в логические группы – домены, идентифицируемые целым числом 0..232 (рекомендуется 0..101). Участники разных доменов невидимы друг для друга на сетевом уровне: SPDP использует разные multicast-порты, поэтому несколько лабораторных систем могут сосуществовать в одной подсети, не пересекаясь. В ROS 2 номер домена задаётся переменной `ROS_DOMAIN_ID`; если она не установлена, используется домен 0.

Дополнительная изоляция достигается переменной `ROS_LOCALHOST_ONLY=1`, при которой Fast DDS использует только loopback-интерфейс. В лабораторной работе эта переменная должна быть выключена, иначе хосты не «увидят» друг друга. В Jazzy доступна также более тонкая настройка `ROS_AUTOMATIC_DISCOVERY_RANGE` со значениями `OFF / LOCALHOST / SUBNET / SYSTEM_DEFAULT`, позволяющая ограничить область обнаружения без правки XML.

3. Жизненный цикл соединения publisher–subscriber. После создания `DomainParticipant` в Fast DDS выполняются следующие этапы (см. рисунок 2): SPDP объявляет существование участника; затем срабатывает `Endpoint Discovery Protocol (EDP)`, который обменивается метаданными о публикаторах и подписчиках – именем топики, полным именем типа сообщения, профилем QoS, идентификатором GUID. Если QoS совместимы, происходит `matching` – пара `DataWriter` ↔ `DataReader`, после чего поток пользовательских данных идёт unicast-пакетами RTPS DATA. При политике `Reliable` приёмная сторона отвечает submessage `ACKNACK`; при `Best Effort` ACK не используется.

4. Профили QoS, используемые в работе. ROS 2 предоставляет несколько predefined профилей QoS, перечисленных в таблице 1. Каждый из них задаёт три ключевые политики: `Reliability` (`Reliable / Best Effort`), `Durability` (`Volatile / Transient Local`) и `History` (`Keep Last N / Keep All`).

При несовпадении политик publisher и subscriber matching не происходит и ros2 topic echo возвращает пустой результат.

Таблица 1 – Стандартные QoS-профили rclcpp/rclpy в ROS 2 Jazzy

Профиль	Reliability	Durability	History	Назначение
rmw_qos_profile_default	Reliable	Volatile	Keep last 10	обычные топики типа /chatter
rmw_qos_profile_sensor_data	Best Effort	Volatile	Keep last 5	сенсоры (LiDAR, IMU); потеря пакета не критична
rmw_qos_profile_services_default	Reliable	Volatile	Keep last 10	RPC через сервисы (rosidl)
rmw_qos_profile_parameters	Reliable	Volatile	Keep last 1000	обновление параметров узла
rmw_qos_profile_system_default	system default	system default	system default	берётся из XML - профиля DDS

5. Конфигурирование Fast DDS через XML-профиль. Тонкая настройка транспорта, локаторов и QoS выполняется через XML-файл, путь к которому передаётся переменной FASTDDS_DEFAULT_PROFILES_FILE (в Fast DDS 3.x это единственное поддерживаемое имя; старая переменная FASTRTPS_DEFAULT_PROFILES_FILE удалена). В файле описываются профили участника, DataWriter и DataReader, конкретные UDP-локаторы, белый список интерфейсов (interfaceWhiteList), параметры сторожевого таймера leaseDuration.

6. Discovery Server – альтернатива multicast. В сетях, где multicast запрещён администратором (например, корпоративный Wi-Fi или несколько VLAN), вместо SPDP используется централизованный сервис Discovery

Server. Один из участников запускается командой `fastdds discovery -i 0`; остальные узлы указывают его адрес через `ROS_DISCOVERY_SERVER`. Сам пользовательский трафик по-прежнему идёт peer-to-peer; через сервер проходят только метаданные SPDP/EDP.

7. `ros2 daemon`. Утилиты командной строки `ros2` (`topic list`, `node info`, ...) для ускорения работают через локальный демон – процесс, который держит открытым `DomainParticipant` и кэширует список обнаруженных узлов. Если граф изменился, а демон ещё не обновил кэш, рекомендуется перезапустить его командами `ros2 daemon stop`, `ros2 daemon start`. Демон существует только для удобства CLI – на runtime коммуникацию между узлами он не влияет.

8. Архитектура двухмашинной системы для данной лабораторной работы показана на рисунке 1: оба хоста находятся в одной подсети 192.168.10.0/24, имеют одинаковые `ROS_DOMAIN_ID` и `RMW_IMPLEMENTATION`; на хосте А запускается узел `/talker` из пакета `demo_nodes_cpp`, на хосте В – узел `/listener`. Логический обмен данными иллюстрирует sequence-диаграмма на рисунке 2.

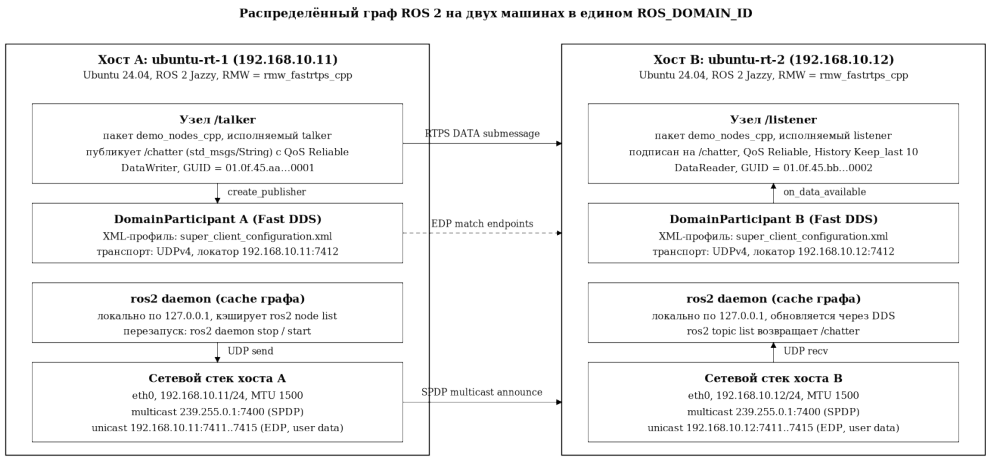


Рисунок 1 – Архитектура двухмашинной системы ROS 2 Jazzy + Fast DDS 3.x

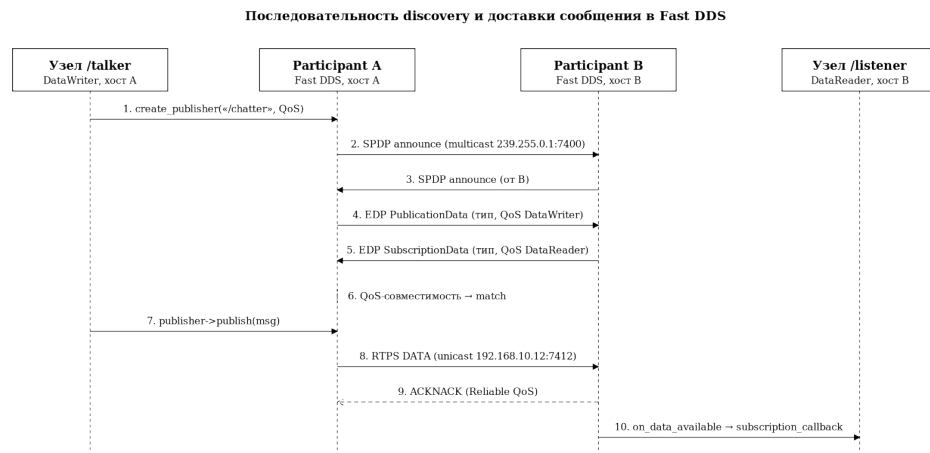


Рисунок 2 – Последовательность discovery и доставки сообщения

9. Оценка задержки. Сквозная задержка end-to-end между публикацией и получением сообщения складывается из четырёх слагаемых:

$$T_{e2e} = T_{serial} + T_{user_kernel} + T_{net} + T_{deserial} + T_{callback} \quad (1)$$

где T_{serial} – сериализация в CDR, T_{user_kernel} – переход в ядро (sendmsg), T_{net} – распространение по сети (включая коммутатор), $T_{deserial}$ – десериализация. На локальной гигабитной сети характерное значение T_{e2e} для сообщения типа std_msgs/String длиной 64 байта составляет 0.3..1.0 мс при профиле Reliable.

Пример выполнения задания

Шаг 1. Подготовка виртуальных машин. Создаются две ВМ под управлением Ubuntu 24.04 LTS («Noble Numbat») – единственный Tier-1 дистрибутив для ROS 2 Jazzy Jalisco (срок поддержки – до мая 2029 года). Сетевой интерфейс каждой ВМ переводится в режим «мост» (bridged) или «внутренняя сеть» (internal), но не NAT, иначе multicast не пройдёт. Назначаются статические адреса 192.168.10.11 (хост А) и 192.168.10.12 (хост В). Проверяется двусторонний ping и доступность UDP-порта 7400 (sudo ufw disable либо открытие диапазона 7400..7500/udp).

Шаг 2. Установка ROS 2 Jazzy. На обоих хостах выполняются одинаковые команды; листинг 1 приведён для одного хоста. Ключи ROS GPG скачиваются с packages.ros.org.

Листинг 1 – Установка ROS 2 Jazzy Jalisco на Ubuntu 24.04

```
sudo apt update && sudo apt install -y curl gnupg lsb-release
sudo curl -sSL https://raw.githubusercontent.com/ros/rosdistro/master/ros.key \
  -o /usr/share/keyrings/ros-archive-keyring.gpg
echo "deb [arch=$(dpkg --print-architecture) "\
"signed-by=/usr/share/keyrings/ros-archive-keyring.gpg] "\
"http://packages.ros.org/ros2/ubuntu $(lsb_release -cs) main" \
  | sudo tee /etc/apt/sources.list.d/ros2.list > /dev/null
sudo apt update
sudo apt install -y ros-jazzy-desktop ros-dev-tools \
  ros-jazzy-demo-nodes-cpp ros-jazzy-demo-nodes-py
echo 'source /opt/ros/jazzy/setup.bash' >> ~/.bashrc
source ~/.bashrc
```

Шаг 3. Настройка окружения. В файл `~/.bashrc` на обоих хостах добавляются переменные `ROS_DOMAIN_ID`, `RMW_IMPLEMENTATION` и путь к XML-профилю Fast DDS. Номер домена выбирается произвольно (рекомендуется не 0, чтобы исключить пересечение с соседними студентами); в примере используется 42.

Листинг 2 – Дополнения к `~/.bashrc` для обоих хостов

```
source /opt/ros/jazzy/setup.bash
export ROS_DOMAIN_ID=42
export RMW_IMPLEMENTATION=rmw_fastrtps_cpp
export ROS_AUTOMATIC_DISCOVERY_RANGE=SUBNET
export FASTDDS_DEFAULT_PROFILES_FILE=~/.fastdds_profile.xml
```

Шаг 4. XML-профиль Fast DDS. Профиль ограничивает участников одним сетевым интерфейсом `eth0` (на случай нескольких NIC) и задаёт `leaseDuration` 10 с – через это время «молчащий» участник будет считаться отключившимся. Формат XML совместим с Fast DDS 3.x – тег `<name>` внутри `<rtps>` остался прежним, а на уровне `<profiles>` используется новый namespace `fastdds`.

Листинг 3 – Файл ~/fastdds_profile.xml – профиль для обоих хостов

```
<?xml version="1.0" encoding="UTF-8" ?>
<profiles xmlns="http://www.eprosima.com/XMLSchemas/fastdds">
  <participant profile_name="lab_participant" is_default_profile="true">
    <rtps>
      <name>lab_participant</name>
      <builtin>
        <discovery_config>
          <leaseDuration>
            <sec>10</sec>
          </leaseDuration>
          <leaseAnnouncement>
            <sec>3</sec>
          </leaseAnnouncement>
        </discovery_config>
      </builtin>
      <userTransports>
        <transport_id>udp_eth0</transport_id>
      </userTransports>
      <useBuiltinTransports>false</useBuiltinTransports>
    </rtps>
  </participant>
  <transport_descriptors>
    <transport_descriptor>
      <transport_id>udp_eth0</transport_id>
      <type>UDIPv4</type>
      <interfaceWhiteList>
        <address>192.168.10.0</address>
      </interfaceWhiteList>
    </transport_descriptor>
  </transport_descriptors>
</profiles>
```

Шаг 5. Запуск talker и listener. На хосте А в первом терминале стартует встроенный пример talker, на хосте В – listener. Оба узла принадлежат пакету demo_nodes_cpp, поставляются бинарником в составе ros-jazzy-demo-nodes-cpp.

Листинг 4 – Запуск пары talker / listener на разных хостах

```
# терминал 1, хост А (192.168.10.11)
ros2 run demo_nodes_cpp talker
# [INFO] [talker]: Publishing: 'Hello World: 1'
# [INFO] [talker]: Publishing: 'Hello World: 2'

# терминал 2, хост В (192.168.10.12)
ros2 run demo_nodes_cpp listener
# [INFO] [listener]: I heard: [Hello World: 1]
```

```
# [INFO] [listener]: I heard: [Hello World: 2]
```

Шаг 6. Проверка обнаружения и состояния графа. С любого хоста должны быть видны оба узла. Команды `ros2 node list`, `ros2 topic list`, `ros2 topic info -v` показывают состав графа и согласованность QoS. Если хост В не видит `/talker`, типовые причины: разные `ROS_DOMAIN_ID`, разные `RMW_IMPLEMENTATION`, блокировка multicast на firewall, разные адреса в `interfaceWhiteList`.

Листинг 5 – Диагностика графа с хоста В

```
ros2 daemon stop && ros2 daemon start
ros2 node list
# /listener
# /talker
ros2 topic list -t
# /chatter [std_msgs/msg/String]
# /parameter_events [rcl_interfaces/msg/ParameterEvent]
ros2 topic info /chatter --verbose
# Type: std_msgs/msg/String
# Publisher count: 1
# Node name: talker
# QoS profile:
#   Reliability: RELIABLE
#   Durability: VOLATILE
#   History (Depth): KEEP_LAST (7)
# Subscription count: 1
# Node name: listener
# QoS profile:
#   Reliability: RELIABLE
#   Durability: VOLATILE
#   History (Depth): KEEP_LAST (10)
ros2 doctor --report | head -n 20
```

Шаг 7. Собственный измеритель задержки. Стандартный `talker` публикует только строку, поэтому для измерения сквозной задержки необходимо написать пару узлов, обменивающихся сообщением со штампом времени. Узлы вынесены в пакет `latency_meter`; их код приведён в листингах 4 и 5.

Листинг 6 – Создание пакета и описание сообщения

```
cd ~/ros2_ws/src
ros2 pkg create --build-type ament_python latency_meter \
```

```
--dependencies rclpy std_msgs builtin_interfaces
# затем в setup.py зарегистрировать два entry-point:
# 'lat_pub = latency_meter.lat_pub:main',
# 'lat_sub = latency_meter.lat_sub:main',
```

Листинг 7 – Узел-публикатор latency_meter/lat_pub.py

```
import rclpy
from rclpy.node import Node
from rclpy.qos import QoSProfile, ReliabilityPolicy, HistoryPolicy
from std_msgs.msg import String

class LatencyPub(Node):
    def __init__(self):
        super().__init__('lat_pub')
        qos = QoSProfile(
            reliability=ReliabilityPolicy.RELIABLE,
            history=HistoryPolicy.KEEP_LAST,
            depth=10)
        self.pub = self.create_publisher(String, '/ping', qos)
        self.create_timer(0.05, self.on_tick) # 20 Hz
        self.seq = 0

    def on_tick(self):
        msg = String()
        t_ns = self.get_clock().now().nanoseconds
        self.seq += 1
        msg.data = f'{self.seq};{t_ns}'
        self.pub.publish(msg)

def main():
    rclpy.init()
    rclpy.spin(LatencyPub())
    rclpy.shutdown()
```

Листинг 8 – Узел-приёмник latency_meter/lat_sub.py с подсчётом задержки

```
import rclpy, statistics
from rclpy.node import Node
from rclpy.qos import QoSProfile, ReliabilityPolicy, HistoryPolicy
from std_msgs.msg import String

class LatencySub(Node):
    def __init__(self):
        super().__init__('lat_sub')
        qos = QoSProfile(
            reliability=ReliabilityPolicy.RELIABLE,
            history=HistoryPolicy.KEEP_LAST,
```

```

        depth=10)
self.create_subscription(String, '/ping', self.on_msg, qos)
self.samples = []
self.create_timer(5.0, self.report)

def on_msg(self, msg: String):
    seq, t_send_ns = msg.data.split(';')
    t_rcv_ns = self.get_clock().now().nanoseconds
    dt_us = (t_rcv_ns - int(t_send_ns)) / 1000.0
    self.samples.append(dt_us)

def report(self):
    if not self.samples:
        return
    s = self.samples
    self.get_logger().info(
        f'N={len(s)} mean={statistics.mean(s):.1f} us '
        f'median={statistics.median(s):.1f} us '
        f'p95={sorted(s)[int(0.95*len(s))]:.1f} us '
        f'max={max(s):.1f} us')
    self.samples = []

def main():
    rclpy.init()
    rclpy.spin(LatencySub())
    rclpy.shutdown()

```

Шаг 8. Запуск измерения. На хосте А собирается рабочее пространство, на хосте В – такое же; источник окружения выполняется через `source install/setup.bash`.

Листинг 9 – Сборка и запуск измерителя

```

# на обоих хостах в ~/ros2_ws
colcon build --packages-select latency_meter
source install/setup.bash

# хост А
ros2 run latency_meter lat_pub

# хост В
ros2 run latency_meter lat_sub
# [INFO] [lat_sub]: N=100 mean=612.4 us median=587.0 us
#           p95=910.0 us max=2104.0 us

```

Замечание о синхронизации времени. Результат корректен только если часы двух ВМ синхронизированы – иначе измеренное «время задержки»

будет содержать сдвиг между шкалами. На обоих хостах должен работать chronyd (sudo systemctl status chrony) или PTP-демон ptp4l из linuxptp 4.x, иначе задержка может оказаться отрицательной.

Шаг 9. Эксперимент с разными QoS. Изменяя профиль публикатора и подписчика, студент наблюдает поведение системы при несовместимом QoS. Например, при reliability=BEST_EFFORT у одной стороны и reliability=RELIABLE у другой matching не происходит: ros2 topic info -v показывает «Publisher count: 1, Subscription count: 1», но сообщения не доставляются, а в журнале появляется QoS incompatibility warning. Эту ситуацию необходимо привести в отчёте.

Шаг 10. Переключение на Cyclone DDS. На одном из хостов ставится альтернативный RMW и сравнивается задержка.

Листинг 10 – Установка и активация Cyclone DDS

```
sudo apt install -y ros-jazzy-rmw-cyclonedds-cpp
export RMW_IMPLEMENTATION=rmw_cyclonedds_cpp
ros2 run latency_meter lat_pub # повторить измерение
```

В отчёте необходимо привести таблицу: профиль QoS, RMW, средняя и 95-процентильная задержка, число пропусков, наблюдения по поведению ros2 daemon при изменении графа.

Задания по вариантам

Номер варианта определяется как $(N - 1) \bmod 15 + 1$, где N – порядковый номер студента в журнале группы. Конкретные числовые параметры задачи берутся из приведённой ниже таблицы. Составить отчет со скриншотами, показывающие выполнение заданий.

Вариант 1.

72. Развернуть две ВМ под Ubuntu 24.04 в одной подсети, поставить ROS 2 Jazzy и rmw_fastrtps_cpp, согласовать ROS_DOMAIN_ID=11.

73. Реализовать пару узлов lat_pub / lat_sub с типом std_msgs/String, передающим (seq, t_ns), и частотами 10, 50 и 200 Гц.

74. Использовать QoS Reliable, Keep_last 10; снять задержку на каждой частоте; построить гистограмму распределения.
75. Объяснить, почему рост частоты до 200 Гц увеличивает p95-задержку (загруженность сетевого буфера, ACKNACK).
76. Снять измерения end-to-end задержки (mean, median, p95, max) за интервал не менее 60 с и оформить таблицу результатов.
77. Подготовить отчёт: схема сети, листинги конфигурации, вывод `ros2 topic info -v`, графики распределения задержки, выводы о применимости выбранной конфигурации.

Вариант 2.

65. Развернуть две VM, ROS 2 Jazzy, домен 12; `rmw_fastrtps_cpp`.
66. Реализовать узлы, обменивающиеся `sensor_msgs/PointCloud2` на 10 000 точек (XYZ + intensity) с частотой 1 Гц.
67. Сравнить задержку и долю потерянных сообщений на двух профилях: `rmw_qos_profile_sensor_data` (Best Effort) и `rmw_qos_profile_default` (Reliable).
68. Объяснить причину разрыва: при больших payload Reliable вынужден ждать ACKNACK и фрагментировать на RTPS DATA_FRAG.
69. Снять измерения end-to-end задержки (mean, median, p95, max) за интервал не менее 60 с и оформить таблицу результатов.
70. Подготовить отчёт: схема сети, листинги конфигурации, вывод `ros2 topic info -v`, графики распределения задержки, выводы о применимости выбранной конфигурации.

Вариант 3.

64. Запустить три узла-публикатора на хосте А (топики `/multi1`, `/multi2`, `/multi3`) и один узел-подписчик на хосте В, подписанный на все три.
65. Подобрать связку Reliability и History так, чтобы потерь не было ни на одной частоте (5, 50, 100 Гц).
66. Оценить влияние depth (1, 5, 50, 500) на пиковую задержку и количество переотправок.

67. Показать поведение при превышении `depth`: какой кадр вытесняется и виден ли `drop` в `ros2 topic hz`.
68. Снять измерения `end-to-end` задержки (`mean`, `median`, `p95`, `max`) за интервал не менее 60 с и оформить таблицу результатов.
69. Подготовить отчёт: схема сети, листинги конфигурации, вывод `ros2 topic info -v`, графики распределения задержки, выводы о применимости выбранной конфигурации.

Вариант 4.

64. Развернуть две ВМ, поставить параллельно `rmw_fastrtps_cpp` и `rmw_cyclonedds_cpp` на обоих хостах.
65. Прогнать одну и ту же пару `lat_pub / lat_sub` при двух RMW; обеспечить идентичные QoS и `payload` (1 КБ, 50 Гц).
66. Сравнить среднюю и пиковую задержку, оценить разброс `p95`.
67. Объяснить, при каком классе нагрузки выбирают Cyclone DDS (меньший `footprint`, иные настройки `flow-control`).
68. Снять измерения `end-to-end` задержки (`mean`, `median`, `p95`, `max`) за интервал не менее 60 с и оформить таблицу результатов.
69. Подготовить отчёт: схема сети, листинги конфигурации, вывод `ros2 topic info -v`, графики распределения задержки, выводы о применимости выбранной конфигурации.

Вариант 5.

64. Заблокировать `multicast` на хосте В (`iptables -A INPUT -d 224.0.0.0/4 -j DROP`); убедиться, что обычное обнаружение перестаёт работать.
65. Поднять Discovery Server командой `fastdds discovery -i 0` на третьем хосте; задать `ROS_DISCOVERY_SERVER=<addr>:11811` на А и В.
66. Восстановить связность талкера и листенера через сервер; подтвердить `ros2 node list`.
67. Сравнить время первичного `matching` при `multicast`-режиме и при Discovery Server.

68. Снять измерения end-to-end задержки (mean, median, p95, max) за интервал не менее 60 с и оформить таблицу результатов.

69. Подготовить отчёт: схема сети, листинги конфигурации, вывод `ros2 topic info -v`, графики распределения задержки, выводы о применимости выбранной конфигурации.

Вариант 6.

63. Сравнить три профиля QoS: `rmw_qos_profile_default`, `rmw_qos_profile_sensor_data`, `rmw_qos_profile_parameters` на одном и том же топики `std_msgs/Int32MultiArray` (1 КБ).

64. Снять статистику задержки и потерь на частотах 10, 100, 1000 Гц; зафиксировать поведение при перегрузке.

65. Построить сводный график: профиль × частота × p95-задержка.

66. Объяснить, почему параметрический профиль (`depth 1000`) снижает потери при бёрст-нагрузке, но увеличивает память.

67. Снять измерения end-to-end задержки (mean, median, p95, max) за интервал не менее 60 с и оформить таблицу результатов.

68. Подготовить отчёт: схема сети, листинги конфигурации, вывод `ros2 topic info -v`, графики распределения задержки, выводы о применимости выбранной конфигурации.

Вариант 7.

66. Запустить пару `talker / listener` с QoS `Reliable`; параллельно на третьем хосте записать `/chatter` командой `ros2 bag record`.

67. Повторить с QoS `Best Effort` у писателя и `Best Effort` у `ros2 bag` (QoS `override` через YAML-файл).

68. Измерить долю записанных сообщений в обоих режимах при частоте 200 Гц и `payload 4 КБ`.

69. Объяснить разницу: `Best Effort` не делает `retransmit`, поэтому при загрузке сети `bag` теряет часть кадров.

70. Снять измерения end-to-end задержки (mean, median, p95, max) за интервал не менее 60 с и оформить таблицу результатов.

71. Подготовить отчёт: схема сети, листинги конфигурации, вывод `ros2 topic info -v`, графики распределения задержки, выводы о применимости выбранной конфигурации.

Вариант 8.

65. Поднять четыре ВМ; разделить на две пары с `ROS_DOMAIN_ID` 41 и 42; запустить `talker` / `listener` внутри каждой пары.

66. Подтвердить взаимную невидимость командой `ros2 node list` с каждого хоста.

67. На одном из хостов «на лету» сменить `ROS_DOMAIN_ID` и перезапустить `talker`; проверить, что `ros2 daemon` при этом тоже надо перезапустить (`stop` / `start`).

68. Объяснить, как `multicast`-порт зависит от `ROS_DOMAIN_ID` ($7400 + 250 \cdot id$) и какие порты заняты в данном эксперименте.

69. Снять измерения end-to-end задержки (mean, median, p95, max) за интервал не менее 60 с и оформить таблицу результатов.

70. Подготовить отчёт: схема сети, листинги конфигурации, вывод `ros2 topic info -v`, графики распределения задержки, выводы о применимости выбранной конфигурации.

Вариант 9.

66. Реализовать пару `sender` / `receiver` на `rclcpp` (C++) для сообщения `std_msgs/String` длиной 256 байт, 100 Гц.

67. Реализовать аналогичную пару на `rclpy` (Python) с теми же параметрами.

68. Сравнить mean / p95 задержку обеих реализаций при одном RMW (`rmw_fastrtps_cpp`).

69. Объяснить разрыв через профилирование сериализации CDR (оверхед GIL и Python-биндингов `rosidl`).

70. Снять измерения end-to-end задержки (mean, median, p95, max) за интервал не менее 60 с и оформить таблицу результатов.

71. Подготовить отчёт: схема сети, листинги конфигурации, вывод `ros2 topic info -v`, графики распределения задержки, выводы о применимости выбранной конфигурации.

Вариант 10.

65. Реализовать публикатор с QoS Durability=TRANSIENT_LOCAL и depth=5, публикующий /state каждые 2 с.

66. Запустить публикатор, выждать 15 с, затем поднять подписчик с теми же QoS на другом хосте.

67. Убедиться, что подписчик сразу получает последние 5 сохранённых сообщений; повторить эксперимент с VOLATILE и показать, что в этом случае подписчик не получает истории.

68. Сравнить ситуацию с очередью брокера (Kafka retain) и объяснить ограничения TRANSIENT_LOCAL (хранится только у живого писателя).

69. Снять измерения end-to-end задержки (mean, median, p95, max) за интервал не менее 60 с и оформить таблицу результатов.

70. Подготовить отчёт: схема сети, листинги конфигурации, вывод `ros2 topic info -v`, графики распределения задержки, выводы о применимости выбранной конфигурации.

Вариант 11.

66. Реализовать DataReader с deadline QoS = 100 мс на топике /heartbeat и зарегистрировать callback `on_requested_deadline_missed`.

67. Реализовать DataWriter с offered deadline 100 мс и callback `on_offered_deadline_missed`.

68. Искусственно занизить частоту публикатора до 1 Гц; продемонстрировать вызовы обоих callback-ов в журналах.

69. Восстановить частоту 20 Гц; подтвердить отсутствие пропусков deadline в течение 60 с.

70. Снять измерения end-to-end задержки (mean, median, p95, max) за интервал не менее 60 с и оформить таблицу результатов.

71. Подготовить отчёт: схема сети, листинги конфигурации, вывод `ros2 topic info -v`, графики распределения задержки, выводы о применимости выбранной конфигурации.

Вариант 12.

66. Применить на одном из хостов сетевые потери: `sudo tc qdisc add dev eth0 root netem loss 5 %`.

67. Сравнить долю доставленных кадров `std_msgs/String` длиной 1 КБ при QoS Reliable и Best Effort, 50 Гц, 60 с.

68. Построить график «потери канала → доля доставленных» в диапазоне 0 .. 20 % с шагом 2.5 %.

69. Объяснить, почему Reliable удерживает 100 % при умеренных потерях ценой роста p95-задержки (ретрансляции).

70. Снять измерения end-to-end задержки (mean, median, p95, max) за интервал не менее 60 с и оформить таблицу результатов.

71. Подготовить отчёт: схема сети, листинги конфигурации, вывод `ros2 topic info -v`, графики распределения задержки, выводы о применимости выбранной конфигурации.

Вариант 13.

65. Запустить пару `talker / listener` на хостах А и В; на третьем хосте-наблюдателе поднять одновременно `ros2 topic echo /chatter` и `ros2 topic hz /chatter`.

66. Убедиться, что наблюдатель видит сообщения, но не нарушает работу основной пары (задержка не выросла, потерь нет).

67. Через `ros2 topic info -v` показать, что у `/chatter` теперь `Publisher count=1`, `Subscription count=2`.

68. Сравнить нагрузку на сеть командой `ifstat` при одном и при двух подписчиках; объяснить, что `multicast` не используется для пользовательских данных в Fast DDS по умолчанию.

69. Снять измерения end-to-end задержки (mean, median, p95, max) за интервал не менее 60 с и оформить таблицу результатов.

70. Подготовить отчёт: схема сети, листинги конфигурации, вывод `ros2 topic info -v`, графики распределения задержки, выводы о применимости выбранной конфигурации.

Вариант 14.

66. Изменить в XML-профиле `leaseDuration` с 10 с на 2 с и `leaseAnnouncement` с 3 с на 0.5 с.

67. Запустить `talker` и `listener`; выключить хост А (`poweroff`) и засечь время, через которое `ros2 node list` на хосте В перестаёт показывать `/talker`.

68. Повторить эксперимент при `leaseDuration` = 30 с; сравнить времена.

69. Объяснить компромисс: малый `leaseDuration` ускоряет обнаружение отказов, но увеличивает background-трафик SPDP.

70. Снять измерения end-to-end задержки (mean, median, p95, max) за интервал не менее 60 с и оформить таблицу результатов.

71. Подготовить отчёт: схема сети, листинги конфигурации, вывод `ros2 topic info -v`, графики распределения задержки, выводы о применимости выбранной конфигурации.

Вариант 15.

66. Реализовать публикатор `sensor_msgs/Image` (640×480, MONO8, 30 Гц) на хосте А и подписчик на хосте В.

67. Замерить пропускную способность (МБ/с) и сквозную задержку при QoS `Reliable` и `Best Effort`.

68. Объяснить, почему при большом `payload` (≈ 300 КБ на кадр) разумен `Best Effort`: фрагментация RTPS `DATA_FRAG` и `ACKNACK`-цикл становятся узким местом.

69. Подобрать оптимальный `depth` (1 .. 20) для текущей частоты, чтобы не накапливался отставший хвост сообщений.
70. Снять измерения `end-to-end` задержки (`mean`, `median`, `p95`, `max`) за интервал не менее 60 с и оформить таблицу результатов.
71. Подготовить отчёт: схема сети, листинги конфигурации, вывод `ros2 topic info -v`, графики распределения задержки, выводы о применимости выбранной конфигурации.

Лабораторная работа 18. Итоговый проект: распределённая система управления группой симулированных роботов

Цель работы: разработать и запустить распределённую систему управления группой из трёх симулированных колёсных роботов TurtleBot3 в Gazebo Harmonic, объединяющую все ключевые технологии курса: симулятор Gazebo с штатным мостом `ros_gz_bridge`, обмен телеметрией через DDS (Fast DDS 3.x, `RMW_IMPLEMENTATION=rmw_fastrtps_cpp`), симулированную шину CAN (SocketCAN, `vcan0`) для имитации низкоуровневого контура управления приводами, синхронизацию времени по PTPv2 (`linuxptp 4.x`, IEEE 802.1AS-2020) и механизм отказоустойчивого переизбрания лидера по алгоритму `bully`. Итог работы – связное движение трёх роботов к общей цели в выбранном строю (линия / клин / круг) с избеганием столкновений и автоматическим продолжением миссии при отключении любого члена формации, включая текущего лидера.

Теоретическое обоснование

1. Постановка задачи. В качестве объекта управления рассматривается группа однотипных мобильных роботов $R = \{R1, R2, R3\}$. Каждый робот моделируется в Gazebo Harmonic (LTS-релиз, поддерживается до 2028 года) как стандартный TurtleBot3 Burger (диаметр шасси 138 мм, дифференциальный привод, 360-градусный LiDAR LDS-01, IMU). Все роботы помещены в общую сцену `multi_tb3.world` и публикуют одометрию и сканы под пространствами имён `/r1`, `/r2`, `/r3` соответственно. Цель миссии – переместиться к заданной точке (x^*, y^*) на плоскости, сохраняя заданный строй и избегая столкновений как между собой, так и со статическими препятствиями.

2. Архитектура системы (см. рисунок 1). Логически система разделена на четыре уровня. Нижний уровень – симулированная электрика и приводы каждого робота: моделируется отдельной виртуальной шиной CAN (`vcan0`), на которой передаются кадры целевой угловой скорости двух колёс. Средний

уровень – узлы ROS 2 Jazzy, превращающие cmd_vel в CAN-фреймы (can_bridge), и узлы формаций (formation_node), вычисляющие cmd_vel из текущей позы лидера и желаемого смещения. Верхний уровень – общий координационный трафик в DDS (топики /goal_pose, /leader_id, /heartbeat, /formation_state). Отдельным уровнем стоит инфраструктура: RTPv2-синхронизация часов и failover_mgr. Симулятор Gazebo Harmonic связан с ROS 2 через мост ros_gz_bridge (пакет ros-jazzy-ros-gz).

3. Распределение ролей. Один из роботов в каждый момент времени играет роль лидера; остальные – ведомые. Лидер вычисляет траекторию к /goal_pose с использованием простого регулятора (pure pursuit), ведомые поддерживают позицию относительно лидера согласно правилу формации:

$$p_i^d(t) = p_L(t) + R(\theta_L(t)) \cdot \Delta_i \quad (1)$$

где $p_L(t)$, $\theta_L(t)$ – поза лидера в глобальной СК, $R(\theta)$ – матрица поворота, Δ_i – статическое смещение ведомого i в СК лидера (для строя «линия» – $\Delta_2 = (0, +0.6)$, $\Delta_3 = (0, -0.6)$; для «клина» – $\Delta_2 = (-0.5, +0.4)$, $\Delta_3 = (-0.5, -0.4)$).

4. Профили QoS для координационных топиков. Различные классы сообщений требуют разных политик. Сводка приведена в таблице 1; правильная настройка QoS критична: ошибочный выбор Best Effort для /leader_id приведёт к разногласию в группе, а Reliable для высокочастотного /odom – к переполнению очереди при временной потере связи.

Таблица 1 – QoS-профили координационных топиков системы

Топик	Тип	Reliability	Durability	Частота
/goal_pose	geometry_msgs/PoseStamped	Reliable	TRANSIENT_LOCAL	по событию
/leader_id	std_msgs/UInt8	Reliable	TRANSIENT_LOCAL	по событию

Топик	Тип	Reliability	Durability	Частота события
heartbeat	builtin_interfaces/Time	Reliable	VOLATILE	10 Гц
odom	nav_msgs/Odometry	Best Effort	VOLATILE	50 Гц
scan	sensor_msgs/LaserScan	Best Effort	VOLATILE	5 Гц
cmd_vel	geometry_msgs/Twist	Reliable	VOLATILE	20 Гц

5. Симуляция CAN через SocketCAN. SocketCAN – штатная подсистема ядра Linux, предоставляющая интерфейс к шине CAN через сокеты семейства PF_CAN. Для лабораторной работы используется виртуальное устройство vcan0, создаваемое через `ip link add dev vcan0 type vcan` – оно ведёт себя как реальная шина, но без физической линии. Узел can_bridge каждого робота получает /rN/cmd_vel и кодирует в два CAN-фрейма: ID 0x100 – целевой rad/s левого колеса, ID 0x101 – правого. На приёмной стороне (имитатор «нижнего» MCU) пакеты можно слушать через `candump vcan0`.

6. Синхронизация времени. Координация движения требует единой шкалы времени с погрешностью не хуже 1 мс, иначе сравнение моментов «когда был отправлен heartbeat» и «когда он пришёл» теряет смысл. Используется протокол IEEE 1588 Precision Time Protocol версии 2 (PTPv2) в профиле gPTP IEEE 802.1AS-2020, реализация linuxptp 4.x (пакет linuxptp в Ubuntu 24.04). Один из хостов (как правило, лидер по умолчанию – R1) запускает ptp4l в режиме master; остальные хосты – в режиме slave. Демон phc2sys подтягивает системные часы CLOCK_REALTIME за PTP Hardware Clock.

В виртуальной среде, где сетевые карты не поддерживают аппаратный timestamping, PTP работает в software-режиме (-S флаг ptp4l) с погрешностью 50..200 мкс, чего достаточно для задач координации на скоростях до 1 м/с.

Поскольку начиная с Linux 6.12 поддержка PREEMPT_RT включена в mainline kernel, отдельный патч-набор больше не требуется – достаточно установить ядро серии generic-hwe-24.04 или low-latency.

7. Алгоритм выборов лидера (bully). Каждые 100 мс каждый робот публикует heartbeat (timestamp + ID) в топик /rN/heartbeat. Все узлы failover_mgr подписаны на heartbeat всех остальных роботов. Если в течение $T_{\text{timeout}} = 500$ мс ни одного heartbeat от текущего лидера не поступило, инициируется процедура bully (см. рисунок 2):

78. Подозреваемая нода рассылает /election(candidate=self_id) и ждёт $T_{\text{wait}} = 200$ мс ответа от роботов с меньшим ID.

79. Если не пришло ни одного ответа, нода объявляет себя лидером, публикуя /leader_id(self_id) с QoS Reliable + TRANSIENT_LOCAL (чтобы поздно подключающиеся узлы получили актуальное значение).

80. Получив /election с ID меньше своего, узел отправляет ALIVE и сам инициирует новые выборы.

81. Получив /leader_id, все узлы обновляют состояние и переходят к выполнению формации с новым лидером.

8. Регулятор движения лидера. Используется упрощённый pure-pursuit: на каждом шаге выбирается look-ahead-точка на отрезке [текущая поза, цель] на расстоянии $L_d = 0.4$ м; вычисляется кривизна траектории к ней:

$$\kappa = 2 \cdot \sin(\alpha) / L_d \quad (2)$$

где α – угол между текущим направлением робота и вектором на look-ahead-точку. Линейная скорость задаётся постоянной $v = 0.15$ м/с, угловая – $\omega = v \cdot \kappa$. Регулятор ведомых аналогичный, но look-ahead-точкой выступает $p_i^d(t)$ из формулы (1).

9. Избегание столкновений. Каждый ведомый, прежде чем издать cmd_vel, проверяет ближайшую дальность по /rN/scan: если $\min(\text{range}) < 0.35$ м в секторе $\pm 30^\circ$, скорость v ограничивается значением $(\min(\text{range}) - 0.2) / 0.15 \cdot v_{\text{max}}$, то есть линейно снижается до нуля при дистанции 0.20 м. Это

простая reactive-стратегия, достаточная для прохождения сцены multi_tb3.world; в продуктивных задачах применяют стек Nav2 (nav2_bringup + nav2_params.yaml, контроллер DWB или MPPI) совместно с slam_toolbox.

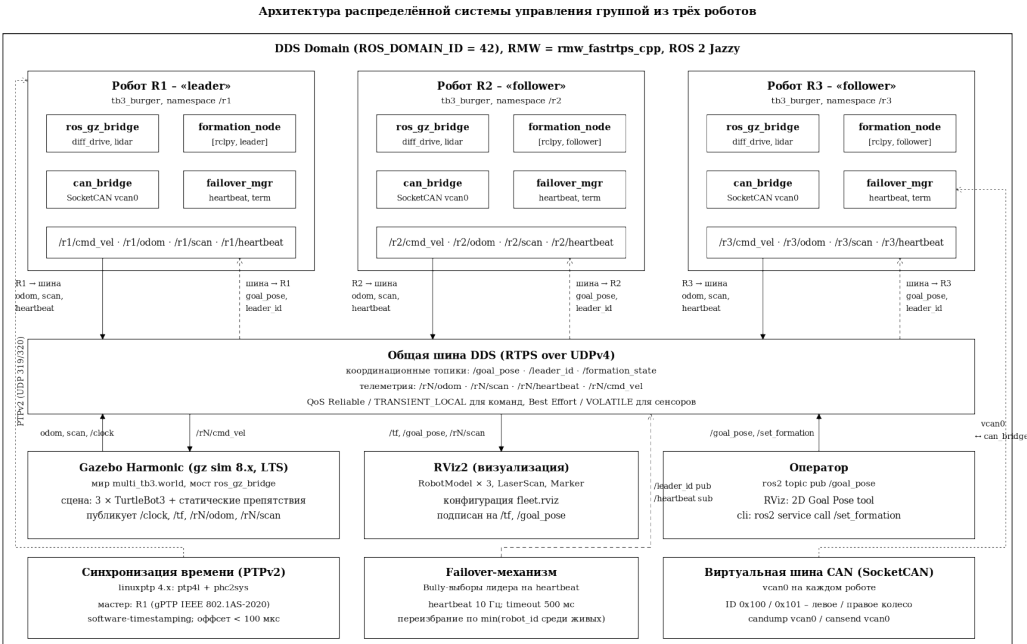


Рисунок 1 – Архитектура распределённой системы из трёх роботов

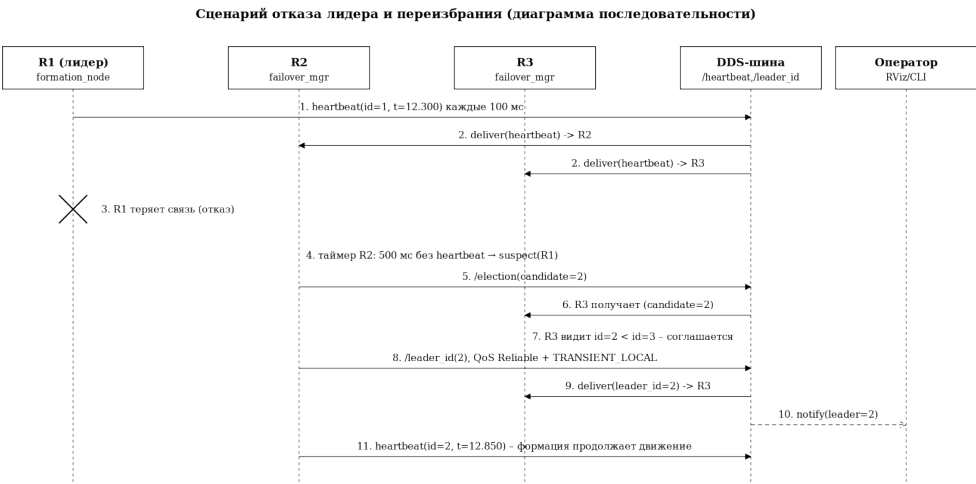


Рисунок 2 – Последовательность событий при отказе лидера

10. Гарантии итоговой системы. В нормальном режиме формация достигает целевой точки за время:

$$T_{mission} \approx ||p_L(0) - p^*|| / v + T_{react} \quad (3)$$

где T_{react} – суммарное время реакции на препятствия. При отказе одного из роботов миссия должна продолжаться без ручного вмешательства; верхняя оценка времени восстановления при отказе лидера составляет $T_{\text{recover}} = T_{\text{timeout}} + T_{\text{wait}} + T_{\text{msg}} \approx 500 + 200 + 50 = 750$ мс.

Пример выполнения задания

Рассмотрим эталонный сценарий: три робота расположены на линии вдоль оси Y с шагом 0.6 м (R1 – в (0, 0), R2 – в (0, 0.6), R3 – в (0, -0.6)); цель – точка (4.0, 0.0). Через 10 секунд после старта вручную «убивается» процесс R1, система должна автоматически выбрать новым лидером R2 и довести формацию до цели.

Шаг 1. Установка и подготовка окружения. Дополнительно к ROS 2 Jazzy ставятся пакеты TurtleBot3, мост Gazebo Harmonic (ros_gz) и linuxptp 4.x.

Листинг 1 – Установка дополнительных пакетов

```
sudo apt install -y \
  ros-jazzy-turtlebot3 \
  ros-jazzy-turtlebot3-msgs \
  ros-jazzy-turtlebot3-gazebo \
  ros-jazzy-ros-gz-bridge ros-jazzy-ros-gz-sim \
  ros-jazzy-rviz2 \
  ros-jazzy-nav2-bringup ros-jazzy-slam-toolbox \
  can-utils linuxptp
export TURTLEBOT3_MODEL=burger
sudo modprobe vcan
sudo ip link add dev vcan0 type vcan
sudo ip link set up vcan0
# проверка версии Gazebo Harmonic (должна быть 8.x)
gz sim --version
```

Шаг 2. Запуск симулятора. Используется launch-файл ros_gz_sim (gz_sim.launch.py) с пользовательским миром, в котором расставлены три робота под разными пространствами имён. Фрагмент launch-файла – в листинге 2.

Листинг 2 – fleet_bringup/launch/fleet.launch.py – запуск трёх роботов

```
from launch import LaunchDescription
```

```

from launch.actions import IncludeLaunchDescription, GroupAction
from launch.launch_description_sources import (
    PythonLaunchDescriptionSource)
from launch_ros.actions import Node, PushRosNamespace
from ament_index_python.packages import get_package_share_directory
import os

def generate_launch_description():
    pkg_gz = get_package_share_directory('ros_gz_sim')
    pkg_tb3 = get_package_share_directory('turtlebot3_gazebo')
    world = os.path.join(pkg_tb3, 'worlds', 'empty_world.world')
    gz = IncludeLaunchDescription(
        PythonLaunchDescriptionSource(
            os.path.join(pkg_gz, 'launch', 'gz_sim.launch.py')),
        launch_arguments={f'gz_args': f'-r {world}}.items())

    spawns = []
    for idx, (name, x, y) in enumerate(
        [('r1', 0.0, 0.0), ('r2', 0.0, 0.6),
        ('r3', 0.0, -0.6)], start=1):
        spawns.append(GroupAction([
            PushRosNamespace(name),
            Node(package='ros_gz_sim', executable='create',
                arguments=[f'-name', name,
                    f'-x', str(x), f'-y', str(y),
                    f'-file', '/opt/ros/jazzy/share/'
                        f'turtlebot3_description/'
                        f'urdf/turtlebot3_burger.urdf']),
            Node(package='ros_gz_bridge', executable='parameter_bridge',
                arguments=[
                    f'/{name}/cmd_vel@geometry_msgs/msg/'
                    f'Twist@gz.msgs.Twist',
                    f'/{name}/odom@nav_msgs/msg/'
                    f'Odometry@gz.msgs.Odometry',
                    f'/{name}/scan@sensor_msgs/msg/'
                    f'LaserScan@gz.msgs.LaserScan',
                ]),
            Node(package='fleet_bringup',
                executable='formation_node',
                parameters=[{'robot_id': idx,
                    'formation': 'line'}]),
            Node(package='fleet_bringup',
                executable='failover_mgr',
                parameters=[{'robot_id': idx,
                    'timeout_ms': 500}]),
            Node(package='fleet_bringup',
                executable='can_bridge',
                parameters=[{'iface': 'vcan0',
                    'id_left': 0x100,
                    'id_right': 0x101}]),
        ]))

```

```
return LaunchDescription([gz] + spawns)
```

Шаг 3. Узел формации. Узел получает `/goal_pose`, `/leader_id` и одометрию всех роботов; в зависимости от собственной роли (лидер / ведомый) публикует `/rN/cmd_vel`.

Листинг 3 – `fleet_bringup/formation_node.py` (фрагмент, упрощённо)

```
import math
import rclpy
from rclpy.node import Node
from rclpy.qos import QoSProfile, ReliabilityPolicy, DurabilityPolicy
from geometry_msgs.msg import PoseStamped, Twist
from nav_msgs.msg import Odometry
from std_msgs.msg import UInt8

FORMATIONS = {
    'line': {1: (0, 0), 2: (0, 0.6), 3: (0, -0.6)},
    'wedge': {1: (0, 0), 2: (-0.5, 0.4), 3: (-0.5, -0.4)},
    'circle': {1: (0, 0.4), 2: (0.35, -0.2), 3: (-0.35, -0.2)},
}

class FormationNode(Node):
    def __init__(self):
        super().__init__('formation_node')
        self.declare_parameter('robot_id', 1)
        self.declare_parameter('formation', 'line')
        self.rid = self.get_parameter('robot_id').value
        self.form = self.get_parameter('formation').value
        self.leader = 1
        self.poses = {}
        self.goal = None

        qos_state = QoSProfile(
            depth=1,
            reliability=ReliabilityPolicy.RELIABLE,
            durability=DurabilityPolicy.TRANSIENT_LOCAL)
        qos_odom = QoSProfile(
            depth=10,
            reliability=ReliabilityPolicy.BEST_EFFORT)

        self.create_subscription(PoseStamped, '/goal_pose',
                                self.on_goal, qos_state)
        self.create_subscription(UInt8, '/leader_id',
                                self.on_leader, qos_state)
        for i in (1, 2, 3):
            self.create_subscription(
                Odometry, f'/r{i}/odom',
                lambda m, k=i: self.poses.__setitem__(k, m),
```

```

        qos_odom)
    self.pub = self.create_publisher(
        Twist, f'/r{self.rid}/cmd_vel', 10)
    self.create_timer(0.05, self.tick) # 20 Hz

def on_goal(self, msg):
    self.goal = (msg.pose.position.x, msg.pose.position.y)

def on_leader(self, msg):
    if msg.data != self.leader:
        self.get_logger().info(
            f'leader changed: {self.leader} -> {msg.data}')
        self.leader = msg.data

def tick(self):
    if self.goal is None or self.rid not in self.poses:
        return
    twist = Twist()
    if self.rid == self.leader:
        tgt = self.goal
    else:
        if self.leader not in self.poses:
            return
        lp = self.poses[self.leader].pose.pose
        dx, dy = FORMATIONS[self.form][self.rid]
        yaw = self._yaw(lp.orientation)
        tgt = (lp.position.x + dx*math.cos(yaw)
              - dy*math.sin(yaw),
              lp.position.y + dx*math.sin(yaw)
              + dy*math.cos(yaw))
        mp = self.poses[self.rid].pose.pose
        myaw = self._yaw(mp.orientation)
        ex, ey = tgt[0]-mp.position.x, tgt[1]-mp.position.y
        alpha = math.atan2(ey, ex) - myaw
        ld = max(0.05, math.hypot(ex, ey))
        v = min(0.15, ld)
        twist.linear.x = v
        twist.angular.z = 2.0*v*math.sin(alpha)/0.4
        self.pub.publish(twist)

    @staticmethod
    def _yaw(q):
        s = 2.0*(q.w*q.z + q.x*q.y)
        c = 1.0 - 2.0*(q.y*q.y + q.z*q.z)
        return math.atan2(s, c)

def main():
    rclpy.init()
    rclpy.spin(FormationNode())
    rclpy.shutdown()

```

Шаг 4. Менеджер failover. Подписан на heartbeat-топики всех роботов; при пропаже лидера запускает выборы по правилу bully.

Листинг 4 – fleet_bringup/failover_mgr.py (фрагмент)

```
import rclpy
from rclpy.node import Node
from rclpy.qos import QoSProfile, ReliabilityPolicy, DurabilityPolicy
from builtin_interfaces.msg import Time
from std_msgs.msg import UInt8

class FailoverMgr(Node):
    def __init__(self):
        super().__init__('failover_mgr')
        self.declare_parameter('robot_id', 1)
        self.declare_parameter('timeout_ms', 500)
        self.rid = self.get_parameter('robot_id').value
        self.tout = self.get_parameter('timeout_ms').value / 1000.0
        self.leader = 1
        self.last_hb = {}

        qos_state = QoSProfile(
            depth=1,
            reliability=ReliabilityPolicy.RELIABLE,
            durability=DurabilityPolicy.TRANSIENT_LOCAL)

        self.pub_hb = self.create_publisher(
            Time, f'/r{self.rid}/heartbeat', 10)
        self.pub_leader = self.create_publisher(
            UInt8, '/leader_id', qos_state)
        for i in (1, 2, 3):
            if i == self.rid:
                continue
            self.create_subscription(
                Time, f'/r{i}/heartbeat',
                lambda m, k=i: self.last_hb.__setitem__(
                    k, self.get_clock().now()),
                10)
        self.create_timer(0.1, self.on_hb_tick) # 10 Hz
        self.create_timer(0.2, self.on_check)

    def on_hb_tick(self):
        t = self.get_clock().now().to_msg()
        self.pub_hb.publish(t)

    def on_check(self):
        now = self.get_clock().now()
        ldr = self.leader
        if ldr == self.rid:
            return
```

```

last = self.last_hb.get(ltr)
if last is None:
    return
dt = (now - last).nanoseconds / 1e9
if dt > self.tout:
    self._elect()

def _elect(self):
    alive = [k for k, t in self.last_hb.items()
              if (self.get_clock().now() - t).nanoseconds/1e9
                 < self.tout]
    alive.append(self.rid)
    new_leader = min(alive)
    if new_leader != self.leader:
        self.get_logger().warn(
            f'leader timeout, electing {new_leader}')
        self.leader = new_leader
        msg = UInt8(); msg.data = new_leader
        self.pub_leader.publish(msg)

def main():
    rclpy.init()
    rclpy.spin(FailoverMgr())
    rclpy.shutdown()

```

Шаг 5. CAN-мост. Подписан на `/rN/cmd_vel` и шлёт два кадра `vcan0`. На выходе `sandump` покажет реальный CAN-трафик.

Листинг 5 – `fleet_bringup/can_bridge.py` (фрагмент)

```

import socket, struct
import rclpy
from rclpy.node import Node
from geometry_msgs.msg import Twist

WHEEL_BASE = 0.160      # м, расстояние между колёсами ТВЗ
WHEEL_RADIUS = 0.033    # м

class CanBridge(Node):
    def __init__(self):
        super().__init__('can_bridge')
        self.declare_parameter('iface', 'vcan0')
        self.declare_parameter('id_left', 0x100)
        self.declare_parameter('id_right', 0x101)
        iface = self.get_parameter('iface').value
        self.id_l = self.get_parameter('id_left').value
        self.id_r = self.get_parameter('id_right').value
        self.s = socket.socket(socket.AF_CAN, socket.SOCK_RAW,
                               socket.CAN_RAW)
        self.s.bind((iface,))

```

```

self.create_subscription(Twist, 'cmd_vel', self.on_cmd, 10)

def on_cmd(self, t: Twist):
    v, w = t.linear.x, t.angular.z
    wl = (v - w*WHEEL_BASE/2) / WHEEL_RADIUS
    wr = (v + w*WHEEL_BASE/2) / WHEEL_RADIUS
    self._send(self.id_l, wl)
    self._send(self.id_r, wr)

def _send(self, can_id, value):
    data = struct.pack('<f', float(value)) + b'\x00'*4
    frame = struct.pack('=IB3x8s', can_id, 8, data)
    self.s.send(frame)

def main():
    rclpy.init()
    rclpy.spin(CanBridge())
    rclpy.shutdown()

```

Шаг 6. Синхронизация времени PTPv2. На хосте R1 запускается ptp4l в режиме мастер, на R2 и R3 – в режиме slave. После 5..10 секунд phc2sys выравнивает CLOCK_REALTIME. В Ubuntu 24.04 поставляется linuxptp 4.x, поддерживающий профиль gPTP IEEE 802.1AS-2020.

Листинг 6 – Запуск PTPv2 (linuxptp 4.x) на трёх хостах

```

# на R1
sudo ptp4l -i eth0 -m -S -p /var/run/ptp4l \
-f /etc/linuxptp/default.cfg --step_threshold=1
sudo phc2sys -a -r

# на R2 и R3 (slave, тот же файл конфигурации)
sudo ptp4l -i eth0 -m -S -s
sudo phc2sys -a -r

# проверка оффсета (рекомендуется < 100 мкс):
journalctl -u ptp4l -f | grep offset

```

Шаг 7. Запуск миссии. Открывается RViz2 с конфигурацией fleet.rviz, после чего инструментом 2D Goal Pose назначается цель – либо публикуется через CLI.

Листинг 7 – Назначение цели и наблюдение

```

# терминал 1
ros2 launch fleet_bringup fleet.launch.py

```

```

# терминал 2 (RViz)
rviz2 -d $(ros2 pkg prefix fleet_bringup)/share/\
fleet_bringup/rviz/fleet.rviz

# терминал 3 (цель)
ros2 topic pub --once /goal_pose geometry_msgs/PoseStamped \
'header: {frame_id: "map"}, pose: {position: \
{x: 4.0, y: 0.0, z: 0.0}, orientation: {w: 1.0}}}'

# терминал 4 (мониторинг CAN-трафика R1)
candump vcan0
# vcan0 100 [8] CD CC 4C 3E 00 00 00 00
# vcan0 101 [8] CD CC 4C 3E 00 00 00 00

# через ~10 секунд: остановить R1
ros2 lifecycle set /r1/formation_node shutdown
# либо: pkill -f 'robot_id:=1'

```

Шаг 8. Ожидаемое поведение. До отключения R1 ведёт формацию к (4, 0); все три `cmd_vel` ненулевые, расстояние между роботами удерживается в окрестности 0.6 м. После `kill R1` в течение примерно 500..750 мс `failover_mgr` R2 фиксирует пропажу `heartbeat` и публикует `/leader_id(2)`. Затем R2 переключается в режим лидера, R3 – пересчитывает целевое смещение относительно R2, и формация (уже из двух роботов) доходит до цели. В журнале наблюдается строка вида «`leader changed: 1 → 2`».

Шаг 9. Анализ результатов. В отчёт включаются: график траекторий трёх роботов (получается из `rosbag`, записанного командой `ros2 bag record /r1/odom /r2/odom /r3/odom /leader_id /goal_pose`), измеренное время восстановления `T_recover`, оффсет часов по данным `journalctl ptp4l`, сравнительная таблица среднего и пикового расхождения между фактическими и теоретическими (по формуле 1) позициями ведомых для каждой из трёх формаций («линия», «клин», «круг»).

Задания по вариантам

Номер варианта определяется как $(N - 1) \bmod 15 + 1$, где N – порядковый номер студента в журнале группы. Конкретные числовые

параметры задачи берутся из приведённой ниже таблицы. Составить отчет со скриншотами, показывающие выполнение заданий.

Вариант 1.

- 71. Формация «линия» поперёк направления движения; цель (4.0, 0.0); расстояние между роботами 0.6 м.
- 72. На 10-й секунде миссии «убить» процесс лидера R1 командой `kill -f 'robot_id:=1'`.
- 73. Замерить `T_recover` (от `kill` до публикации `/leader_id`) и максимальное отклонение R3 от ожидаемой траектории за время восстановления.
- 74. Записать миссию через `ros2 bag record` (топики `/rN/odom`, `/leader_id`, `/goal_pose`); построить график траекторий и временную метку срабатывания `failover_mgr`.
- 75. Подготовить отчёт: схема архитектуры, перечень изменённых файлов, графики траекторий, измеренные времена реакции, выводы об устойчивости системы.

Вариант 2.

- 70. Формация «клин» вершиной вперёд (R1 впереди, R2 и R3 симметрично сзади); цель (5.0, 2.0).
- 71. На 8-й секунде «убить» ведомого R3 (не лидера).
- 72. Показать в логах, что R2 не пытается стать лидером, формация продолжает движение в составе R1 + R2.
- 73. Измерить, сохраняется ли заданное смещение R2 при поворотах траектории.
- 74. Записать миссию через `ros2 bag record` (топики `/rN/odom`, `/leader_id`, `/goal_pose`); построить график траекторий и временную метку срабатывания `failover_mgr`.
- 75. Подготовить отчёт: схема архитектуры, перечень изменённых файлов, графики траекторий, измеренные времена реакции, выводы об устойчивости системы.

Вариант 3.

- 70. Формация «круг» радиуса 0.5 м с равномерным распределением 120° (R1 впереди, R2 и R3 по бокам сзади).
- 71. Цель – точка (3.0, 3.0); скорость 0.15 м/с.
- 72. На 12-й секунде отказ R2; проверить, что новым лидером выбран R3 ($\min(\text{alive_ids})$) либо переоценить алгоритм по $\min(\text{robot_id})$ среди живых.
- 73. Показать перестроение оставшейся пары в одну линию.
- 74. Записать миссию через `ros2 bag record` (топики `/rN/odom`, `/leader_id`, `/goal_pose`); построить график траекторий и временную метку срабатывания `failover_mgr`.
- 75. Подготовить отчёт: схема архитектуры, перечень изменённых файлов, графики траекторий, измеренные времена реакции, выводы об устойчивости системы.

Вариант 4.

- 70. Поставить в Gazebo Harmonic статическое препятствие – ящик 0.4×0.4 м в точке (2.0, 0.0) на траектории формации.
- 71. Запустить миссию в формации «линия»; убедиться, что reactive-стратегия в коде ведомых обрабатывает $\min(\text{range}) < 0.35$ м.
- 72. Зафиксировать минимальную дистанцию до препятствия и время прохождения; сравнить с базовым сценарием без препятствия.
- 73. Записать миссию через `ros2 bag record` (топики `/rN/odom`, `/leader_id`, `/goal_pose`); построить график траекторий и временную метку срабатывания `failover_mgr`.
- 74. Подготовить отчёт: схема архитектуры, перечень изменённых файлов, графики траекторий, измеренные времена реакции, выводы об устойчивости системы.

Вариант 5.

- 69. Реализовать сервис `/set_formation` типа `std_srvs/SetString` со значениями «line», «wedge», «circle».

70. Запустить миссию с цели (5.0, 0.0); на 5-й секунде вызвать `ros2 service call /set_formation "{data: 'wedge'}"`.
71. Замерить время перестроения (от service-вызова до достижения нового шаблона с погрешностью ± 0.05 м).
72. Построить графики траекторий всех трёх роботов с отметками момента переключения.
73. Записать миссию через `ros2 bag record` (топики `/rN/odom`, `/leader_id`, `/goal_pose`); построить график траекторий и временную метку срабатывания `failover_mgr`.
74. Подготовить отчёт: схема архитектуры, перечень изменённых файлов, графики траекторий, измеренные времена реакции, выводы об устойчивости системы.

Вариант 6.

72. Имитировать отказ канала связи у лидера R1: на 10-й секунде выполнить `sudo tc qdisc add dev eth0 root netem loss 100 %` на R1.
73. Сравнить поведение системы с полным killом процесса R1: должен сработать тот же механизм bully за то же $T_{\text{recover}} \approx 500..750$ мс.
74. Через 5 с после переизбрания снять netem (`tc qdisc del`); показать, что R1 снова появляется в графе как ведомый, а не пытается отобрать роль лидера обратно.
75. Записать миссию через `ros2 bag record` (топики `/rN/odom`, `/leader_id`, `/goal_pose`); построить график траекторий и временную метку срабатывания `failover_mgr`.
76. Подготовить отчёт: схема архитектуры, перечень изменённых файлов, графики траекторий, измеренные времена реакции, выводы об устойчивости системы.

Вариант 7.

71. Имитировать рассинхронизацию часов: на R3 остановить phc2sys (sudo systemctl stop phc2sys) и принудительно сместить часы командой sudo date -s '+200 ms'.
72. Запустить базовый сценарий с отказом R1 на 10-й секунде.
73. Показать, как смещение часов R3 сдвигает измеряемое им T_{recover} и может привести к ложному срабатыванию таймера heartbeat.
74. Обосновать необходимость PTPv2 и привести оффсет до и после восстановления phc2sys.
75. Записать миссию через ros2 bag record (топики /rN/odom, /leader_id, /goal_pose); построить график траекторий и временную метку срабатывания failover_mgr.
76. Подготовить отчёт: схема архитектуры, перечень изменённых файлов, графики траекторий, измеренные времена реакции, выводы об устойчивости системы.

Вариант 8.

72. Сценарий двойного отказа: одновременно убить R1 и R2 на 8-й секунде миссии.
73. Должен остаться один R3, продолжающий движение к цели в одиночку (deg-формация).
74. Замерить, через сколько времени R3 объявит себя лидером (должно быть $\leq T_{\text{timeout}} + T_{\text{wait}}$).
75. Показать, что R3 корректно публикует /leader_id(3) и далее ведёт себя как лидер без ведомых.
76. Записать миссию через ros2 bag record (топики /rN/odom, /leader_id, /goal_pose); построить график траекторий и временную метку срабатывания failover_mgr.

77. Подготовить отчёт: схема архитектуры, перечень изменённых файлов, графики траекторий, измеренные времена реакции, выводы об устойчивости системы.

Вариант 9.

71. Реализовать сценарий ручной передачи лидерства оператором: новый сервис `/set_leader` типа `std_srvs/SetUInt8`.

72. При вызове `/set_leader 3` текущий лидер (R1) добровольно снимает с себя роль и публикует `/leader_id(3)` с QoS Reliable + TRANSIENT_LOCAL.

73. R2 и R3 переходят в режим ведомых / лидера соответственно без срабатывания таймера heartbeat.

74. Показать в логах, что bully-выборы не запускались.

75. Записать миссию через `ros2 bag record` (топики `/rN/odom`, `/leader_id`, `/goal_pose`); построить график траекторий и временную метку срабатывания `failover_mgr`.

76. Подготовить отчёт: схема архитектуры, перечень изменённых файлов, графики траекторий, измеренные времена реакции, выводы об устойчивости системы.

Вариант 10.

72. Сценарий нагрузки CAN-шины: запустить параллельно с миссией поток `cangen vcan0 -g 5 -p 2` (один кадр каждые 5 мс).

73. Убедиться, что `can_bridge` продолжает доставлять кадры `cmd_vel` без потерь (проверить `candump | grep '100\|101'`).

74. Замерить пропускную способность шины (`ifstat` по `vcan0`); сравнить с нагрузкой только от `cmd_vel`.

75. Объяснить, какие приоритеты ID следует выбрать для `cmd_vel` при подключении настоящих датчиков на шину.

76. Записать миссию через `ros2 bag record` (топики `/rN/odom`, `/leader_id`, `/goal_pose`); построить график траекторий и временную метку срабатывания `failover_mgr`.

77. Подготовить отчёт: схема архитектуры, перечень изменённых файлов, графики траекторий, измеренные времена реакции, выводы об устойчивости системы.

Вариант 11.

72. Сменить RMW с `rmw_fastrtps_cpp` на `rmw_cyclonedds_cpp` на всех трёх роботах одновременно (export `RMW_IMPLEMENTATION=rmw_cyclonedds_cpp` перед запуском).

73. Повторить базовый сценарий (формация «линия», отказ R1 на 10-й секунде).

74. Сравнить `T_recover` для Fast DDS и Cyclone DDS; снять минимум 10 повторов и привести среднее.

75. Объяснить, влияет ли смена RMW на параметры `TRANSIENT_LOCAL` для `/leader_id` и `/goal_pose`.

76. Записать миссию через `ros2 bag record` (топики `/rN/odom`, `/leader_id`, `/goal_pose`); построить график траекторий и временную метку срабатывания `failover_mgr`.

77. Подготовить отчёт: схема архитектуры, перечень изменённых файлов, графики траекторий, измеренные времена реакции, выводы об устойчивости системы.

Вариант 12.

71. Расширить парк до четырёх роботов: добавить R4 как горячий резерв (не публикует `cmd_vel`, только `heartbeat`).

72. В `fleet.launch.py` создать R4 в стороне (10, 10), он не входит в текущую формацию.

73. При отказе любого из R1..R3 R4 должен занять его место по правилу `bully` (min ID среди живых) и встроиться в формацию.

74. Замерить дополнительное время на перемещение R4 в позицию выбывшего; сравнить полное `T_recover_full` с базовым `T_recover`.

75. Записать миссию через `ros2 bag record` (топики `/rN/odom`, `/leader_id`, `/goal_pose`); построить график траекторий и временную метку срабатывания `failover_mgr`.

76. Подготовить отчёт: схема архитектуры, перечень изменённых файлов, графики траекторий, измеренные времена реакции, выводы об устойчивости системы.

Вариант 13.

72. Заменить QoS `/leader_id` с `TRANSIENT_LOCAL` на `VOLATILE` на всех роботах.

73. Запустить миссию, выполнить отказ R1 на 10-й секунде, получить лидера R2.

74. На 20-й секунде перезапустить `failover_mgr` на R3; показать, что он не получает текущего значения `/leader_id` и работает с устаревшим (либо 0 по умолчанию).

75. Сравнить с поведением при `TRANSIENT_LOCAL`: после перезапуска R3 сразу получает `leader_id=2` «в нагрузку».

76. Записать миссию через `ros2 bag record` (топики `/rN/odom`, `/leader_id`, `/goal_pose`); построить график траекторий и временную метку срабатывания `failover_mgr`.

77. Подготовить отчёт: схема архитектуры, перечень изменённых файлов, графики траекторий, измеренные времена реакции, выводы об устойчивости системы.

Вариант 14.

72. Сценарий неподвижной формации и сложной траектории по «восьмёрке»: оператор публикует последовательно четыре `/goal_pose` в точках (4, 0), (4, 4), (0, 4), (0, 0).

73. Использовать формацию «клин»; интервал между публикациями цели – по достижении предыдущей с допуском 0.2 м.

74. Замерить, удерживается ли заданное Δ_i при поворотах на 90° (фиксировать максимальное отклонение).
75. Подготовить график траекторий всех трёх роботов в одной системе координат.
76. Записать миссию через `ros2 bag record` (топики `/rN/odom`, `/leader_id`, `/goal_pose`); построить график траекторий и временную метку срабатывания `failover_mgr`.
77. Подготовить отчёт: схема архитектуры, перечень изменённых файлов, графики траекторий, измеренные времена реакции, выводы об устойчивости системы.

Вариант 15.

20. Записать полную миссию с отказом лидера через `ros2 bag record -a` в формате `msar`.
21. После завершения миссии воспроизвести `bag` через `ros2 bag play` в режиме `--rate 0.5`, открытом параллельно `RViz2`.
22. Оценить, можно ли по записи восстановить точный момент срабатывания `failover_mgr` (мс-точность) и доказать это через `ros2 bag info` + анализ `/leader_id`.
23. Объяснить, какие QoS-настройки записи (`override`-файл) нужны для корректного воспроизведения `TRANSIENT_LOCAL`-топиков.
24. Записать миссию через `ros2 bag record` (топики `/rN/odom`, `/leader_id`, `/goal_pose`); построить график траекторий и временную метку срабатывания `failover_mgr`.
25. Подготовить отчёт: схема архитектуры, перечень изменённых файлов, графики траекторий, измеренные времена реакции, выводы об устойчивости системы.

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

по организации и проведению самостоятельной работы по дисциплине
«Распределённые системы управления в реальном времени» для студентов
направления подготовки
09.03.04 Программная инженерия
Направленность (профиль)
«Разработка и сопровождение программного обеспечения»

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ	3
ОБЩАЯ ХАРАКТЕРИСТИКА САМОСТОЯТЕЛЬНОЙ РАБОТЫ	3
МЕТОДИЧЕСКИЕ РЕКОМЕНДАЦИИ ПО ИЗУЧЕНИЮ ТЕОРЕТИЧЕСКОГО МАТЕРИАЛА	4
МЕТОДИЧЕСКИЕ УКАЗАНИЯ ПО ВЫПОЛНЕНИЮ ПРАКТИЧЕСКИХ РАБОТ	7
ВОПРОСЫ ДЛЯ СОБЕСЕДОВАНИЯ ПО САМОСТОЯТЕЛЬНОМУ ИЗУЧЕНИЮ ЛИТЕРАТУРЫ	8
КРИТЕРИИ ОЦЕНИВАНИЯ КОМПЕТЕНЦИЙ	9

ВВЕДЕНИЕ

Настоящие методические указания определяют порядок организации и проведения самостоятельной работы студентов по дисциплине «Распределённые системы управления в реальном времени» и являются дополнением к методическим указаниям по выполнению лабораторных работ. Самостоятельная работа рассматривается как обязательная составляющая освоения дисциплины и направлена на закрепление теоретических знаний, развитие практических навыков и подготовку к защите лабораторных работ.

Цель самостоятельной работы — сформировать у студентов устойчивые навыки самостоятельного освоения методов проектирования и анализа распределённых систем управления реальным временем, включая измерение задержек, реализацию периодических задач, работу с промышленными протоколами обмена, настройку DDS, синхронизацию времени, построение отказоустойчивых схем и интеграцию распределённых компонентов в робототехнику.

Указания содержат общую характеристику видов самостоятельной работы, рекомендации по изучению теоретического материала, порядок подготовки к выполнению практических (лабораторных) работ, перечень вопросов для собеседования и критерии оценивания сформированности компетенций.

ОБЩАЯ ХАРАКТЕРИСТИКА САМОСТОЯТЕЛЬНОЙ РАБОТЫ

Самостоятельная работа студента по дисциплине включает проработку теоретического материала, подготовку к лабораторным работам, самостоятельное выполнение индивидуальных заданий и подготовку к их защите. Основные виды самостоятельной работы и формы контроля приведены в таблице 1.

Таблица 1 — Виды самостоятельной работы и формы контроля

Вид самостоятельной работы	Содержание деятельности	Форма контроля
Проработка теоретического материала	Изучение лекций, методических указаний, учебной литературы, стандартов и документации по Linux RT, FreeRTOS, CAN, EtherCAT, DDS, PTP, ROS 2 и связанным инструментам.	Собеседование, тестирование
Подготовка к лабораторной работе	Изучение теоретического обоснования и примера выполнения, подготовка рабочего окружения, виртуальных машин, исходных данных, стенда или симулятора.	Допуск к выполнению работы
Выполнение индивидуального задания	Самостоятельная реализация варианта задания, запуск измерений и экспериментов, сохранение логов, графиков, таблиц, конфигураций и исходного кода.	Проверка отчёта
Подготовка к защите работы	Подготовка ответов на контрольные вопросы, объяснение выбранных параметров, алгоритмов, временных характеристик, ошибок и ограничений реализованного решения.	Защита отчёта

МЕТОДИЧЕСКИЕ РЕКОМЕНДАЦИИ ПО ИЗУЧЕНИЮ ТЕОРЕТИЧЕСКОГО МАТЕРИАЛА

Изучение теоретического материала рекомендуется проводить последовательно, в соответствии с порядком следования лабораторных работ. По каждой теме необходимо ознакомиться с основными понятиями распределённых систем реального времени, разобрать приведённый в работе пример, подготовить программный стенд и проверить понимание материала по вопросам для самоконтроля. Перечень тем для самостоятельного изучения приведён в таблице 2.

Таблица 2 — Темы для самостоятельного изучения теоретического материала

№	Тема для самостоятельного изучения	Вопросы для проработки
1	Установка и настройка среды для РВ-разработки	назначение RT-ядра, CONFIG_PREEMPT_RT, инструменты GCC/CMake/GDB, trace-cmd, cyclictest и POSIX-симулятор FreeRTOS
2	Измерение задержек и джиттера в Linux (cyclictest)	источники задержек в Linux, min/avg/max, гистограммы задержек, влияние приоритетов, фоновой нагрузки и PREEMPT_RT
3	Реализация периодической задачи в FreeRTOS	периодические задачи, тики ядра, xTaskDelayUntil, vTaskDelay, джиттер фазы и накопление drift
4	Сравнение планировщиков RMA и EDF (симуляция)	модель периодической задачи, загрузка процессора, критерий Лиу–Лейленда, статические и динамические приоритеты, пропущенные дедлайны
5	Анализ CAN-трафика с помощью PCAN-View / Wireshark + CAN-USB	структура CAN-кадра, битрейт, арбитраж, загрузка шины, CAN 2.0, CAN FD и CAN XL
6	Отправка и приём CAN-сообщений на STM32 с CAN-контроллером	bxCAN/FDCAN, фильтры приёма, почтовые ящики передачи, прерывания HAL, UART-вывод и симуляция через python-can
7	Программная эмуляция EtherCAT	топология EtherCAT, processing on the fly, SOEM, процесс-образ, цикл обмена и роль master/slave
8	Производительность EtherCAT vs стандартный TCP/IP	RTT, джиттер, дисперсия задержек, TCP_NODELAY, raw-сокеты и критерии применимости в системах жёсткого реального времени
9	Установка Fast DDS и создание Publisher/Subscriber	DomainParticipant, Topic, Publisher, Subscriber, DataWriter, DataReader, IDL и Simple Discovery Protocol
10	Настройка QoS для гарантированной доставки	BEST_EFFORT и RELIABLE, совместимость

	(RELIABLE)	QoS, Heartbeat/ACKNACK, потери пакетов и процент доставленных сообщений
11	DDS с DEADLINE и LATENCY_BUDGET	временные QoS-политики, listener-коллбеки, offered/requested deadline, измерение задержки доставки и реакция приложения
12	Распределённая система с несколькими участниками (Domain, Partition)	domain_id, PDP/EDP, multicast discovery, PARTITION, изоляция трафика и диагностика обнаружения участников
13	Синхронизация времени с PTP (Linux PTP)	IEEE 1588, grandmaster, ordinary clock, ptp4l, phc2sys, offset, drift и точность синхронизации
14	Симуляция отказоустойчивой системы (Leader-Follower)	роли leader/follower, heartbeat, election timeout, кворум, split vote, safety/liveness и время восстановления
15	Двухканальное резервирование (1oo2)	архитектура 1oo2, диагностический порог, безопасное состояние, ложные срабатывания и обнаружение расхождения каналов
16	Распределённая транзакция: Saga (хореография)	локальные транзакции, события Reserved/Failed/Compensated, компенсация, happy path и отличие хореографии от оркестрации
17	Интеграция DDS с ROS 2	RMW, Fast DDS, discovery между хостами, ROS_DOMAIN_ID, QoS-профили, ros2 doctor и измерение сквозной задержки
18	Итоговый проект: распределённая система управления группой симулированных роботов	архитектура проекта, Gazebo, ros_gz_bridge, DDS, SocketCAN, PTP, failover, формация роботов и анализ устойчивости миссии

МЕТОДИЧЕСКИЕ УКАЗАНИЯ ПО ВЫПОЛНЕНИЮ ПРАКТИЧЕСКИХ РАБОТ

Выполнение каждой лабораторной работы осуществляется самостоятельно в соответствии с полученным вариантом индивидуального задания. Рекомендуется придерживаться последовательности этапов, приведённой в таблице 3.

Таблица 3 — Этапы выполнения лабораторной работы

Этап	Содержание деятельности
Подготовительный	Изучение цели, теоретического обоснования и примера выполнения; подготовка окружения Linux/FreeRTOS/STM32/ROS 2, виртуальных машин, симуляторов, сетевых интерфейсов и исходных данных.
Основной	Самостоятельное выполнение индивидуального задания, реализация алгоритмов и узлов обмена, настройка параметров, запуск экспериментов и сохранение результатов.
Измерительный	Снятие временных характеристик, задержек, джиттера, пропусков дедлайнов, процента доставленных сообщений, времени восстановления или показателей устойчивости системы.
Аналитический	Сравнение полученных результатов, выявление влияния параметров, анализ ошибок, ограничений, отказов и применимости решения в системах реального времени.
Оформление отчёта	Подготовка отчёта в соответствии с требованиями работы: цель, ход выполнения, листинги, скриншоты, таблицы, графики, логи, конфигурации и выводы.
Защита	Ответы на контрольные вопросы, обоснование выбранных методов и параметров, демонстрация работоспособности программы, стенда или симулятора.

Отчёт оформляется в текстовом редакторе, листинги программ, команд и конфигураций приводятся моноширинным шрифтом. По итогам работы формулируются выводы, отражающие достигнутые результаты, значения измеренных характеристик, особенности настройки системы, выявленные

ограничения и приобретённые навыки разработки распределённых систем управления реального времени.

ВОПРОСЫ ДЛЯ СОБЕСЕДОВАНИЯ ПО САМОСТОЯТЕЛЬНОМУ ИЗУЧЕНИЮ ЛИТЕРАТУРЫ

Для подготовки к собеседованию по результатам самостоятельного изучения литературы рекомендуется проработать следующие вопросы:

1. Почему стандартный Linux без PREEMPT_RT не считается системой жёсткого реального времени?
2. Какие показатели cyclictst необходимо сравнивать при оценке задержек планировщика и почему максимум важнее среднего значения?
3. Чем xTaskDelayUntil отличается от vTaskDelay и почему первый вариант лучше подходит для периодических задач?
4. Как вычисляется коэффициент загрузки процессора для набора периодических задач и что означает условие $U > 1$?
5. В чём различие между RMA и EDF с точки зрения назначения приоритетов?
6. Как устроен стандартный кадр CAN 2.0A и за счёт чего реализуется неразрушающий арбитраж?
7. Какие функции выполняют фильтры приёма и почтовые ящики передачи в CAN-контроллере STM32?
8. В чём состоит принцип processing on the fly в EtherCAT и почему он уменьшает время цикла?
9. Почему стандартный TCP/IP может иметь высокий джиттер даже при малой средней задержке?
10. Какие сущности входят в модель DDS и какую роль выполняет DomainParticipant?
11. Чем BEST_EFFORT отличается от RELIABLE и какие накладные расходы возникают при гарантированной доставке?
12. Как политики DEADLINE и LATENCY_BUDGET помогают приложению контролировать временные ограничения обмена?

13. Как domain_id и PARTITION разделяют трафик в распределённой DDS-системе?

14. Чем RTP отличается от NTP и почему RTP применяется в локальных системах управления реального времени?

15. Какие условия должны выполняться, чтобы система Leader-Follower сохраняла safety и liveness?

16. В чём состоит компромисс схемы резервирования 1oo2 между безопасностью и ложными срабатываниями?

17. Как в Saga-компенсации обеспечивается отсутствие зарезервированных ресурсов после неуспешного завершения операции?

18. Как ROS 2 использует DDS через RMW и какие параметры нужно проверить при запуске узлов на разных хостах?

19. Какие подсистемы объединяются в итоговом проекте управления группой роботов и как проверяется отказоустойчивость миссии?

КРИТЕРИИ ОЦЕНИВАНИЯ КОМПЕТЕНЦИЙ

Оценивание результатов самостоятельной работы осуществляется по итогам выполнения и защиты лабораторных работ, а также собеседования. Критерии оценивания приведены в таблице 4.

Таблица 4 — Критерии оценивания

Оценка	Критерии оценивания
«Отлично»	Задание выполнено в полном объёме и без ошибок; отчёт оформлен в соответствии с требованиями; студент уверенно объясняет алгоритмы, параметры, временные характеристики, результаты измерений и ограничения реализованного решения.
«Хорошо»	Задание выполнено в полном объёме с незначительными недочётами; отчёт оформлен с небольшими замечаниями; студент в целом объясняет ход работы и полученные результаты.
«Удовлетворительно»	Задание выполнено частично; отчёт содержит ошибки и недочёты; студент испытывает затруднения при объяснении отдельных этапов работы, выбранных параметров и результатов измерений.
«Неудовлетворительно»	Задание не выполнено или выполнено с

	грубыми ошибками; отчёт отсутствует или не соответствует требованиям; студент не может объяснить ход выполнения работы.
--	---