

Университет ИТМО  
Факультет программной инженерии и компьютерной техники

**Лабораторная работа №4**  
по дисциплине «Распределенные системы хранения данных»

Выполнил:

Студент группы Р3331

Дворкин Борис Александрович

**Вариант: 36852**

Преподаватель:  
Николаев Владимир Вячеславович

г. Санкт-Петербург

2024 г.

# Содержание

Описание задания.....	3
Этап 1. Конфигурация.....	3
Этап 2. Симуляция и обработка сбоя.....	3
2.1 Подготовка:.....	3
2.2 Сбой:.....	3
2.3 Обработка:.....	3
Этап 3. Восстановление.....	3
Выполнение.....	4
Этап 1 Конфигурация.....	4
1.1 Физическая организация кластера.....	5
ДАЛЕЕ БУДУТ НАСТРОЙКИ ДЛЯ FEDORA WORKSTATION 39!.....	8
1.2 Настройка узлов.....	8
Далее на каждой малинке обновляем кластер.....	8
Настройка репликации для малинки 1 (синхронная):.....	9
Настройка репликации для малинки 2 (асинхронная с задержкой):.....	9
На основном узле создаем слоты репликации:.....	9
Небольшая теоретическая пометочка:.....	10
Создадим тестовую таблицу для репликации:.....	10
Проверим результат репликации на обоих standby:.....	10
1.3 Настройка pgpool-ii.....	11
docker exec -it pgpool bash -c "cat /opt/bitnami/pgpool/conf/pgpool.conf".....	11
1.3 РАБОТАЕТ.....	14
Тестирование репликации (Этап 1):.....	14
Этап 2: Симуляция и обработка сбоя.....	15
Симуляция сбоя:.....	15
Логи и Переключение:.....	16
Этап 3: Восстановление.....	16
На ноуте:.....	16
Новый pgpool2:.....	17
Вывод.....	19
Конфигурация.....	19
Габариты.....	19
Удешевление.....	19
Узкие места.....	20
Софтверная часть.....	20
Ура! Это закончилось, стипа за Selectel на базу :D.....	22

## Описание задания

Цель работы - ознакомиться с методами и средствами построения отказоустойчивых решений на базе СУБД Postgres; получить практические навыки восстановления работы системы после отказа.

Работа рассчитана на двух человек (а я конченый, делаю один 😊) и выполняется в три этапа: настройка, симуляция и обработка сбоя, восстановление.

### Этап 1. Конфигурация

Настроить репликацию postgres на трёх узлах: А - основной, В и С - резервные. Для управления использовать pgpool-II. Репликация с А на В синхронная. Репликация с А на С асинхронная.

Продемонстрировать, что новые данные реплицируются на В в синхронном режиме, а на С с задержкой.

### Этап 2. Симуляция и обработка сбоя

#### 2.1 Подготовка:

- Установить несколько клиентских подключений к СУБД.
- Продемонстрировать состояние данных и работу клиентов в режиме чтение/запись.

#### 2.2 Сбой:

Симулировать ошибку диска на основном узле - удалить директорию PGDATA со всем содержимым.

#### 2.3 Обработка:

- Найти и продемонстрировать в логах релевантные сообщения об ошибках.
- Выполнить переключение (failover) на резервный сервер.
- Продемонстрировать состояние данных и работу клиентов в режиме чтение/запись.

### Этап 3. Восстановление

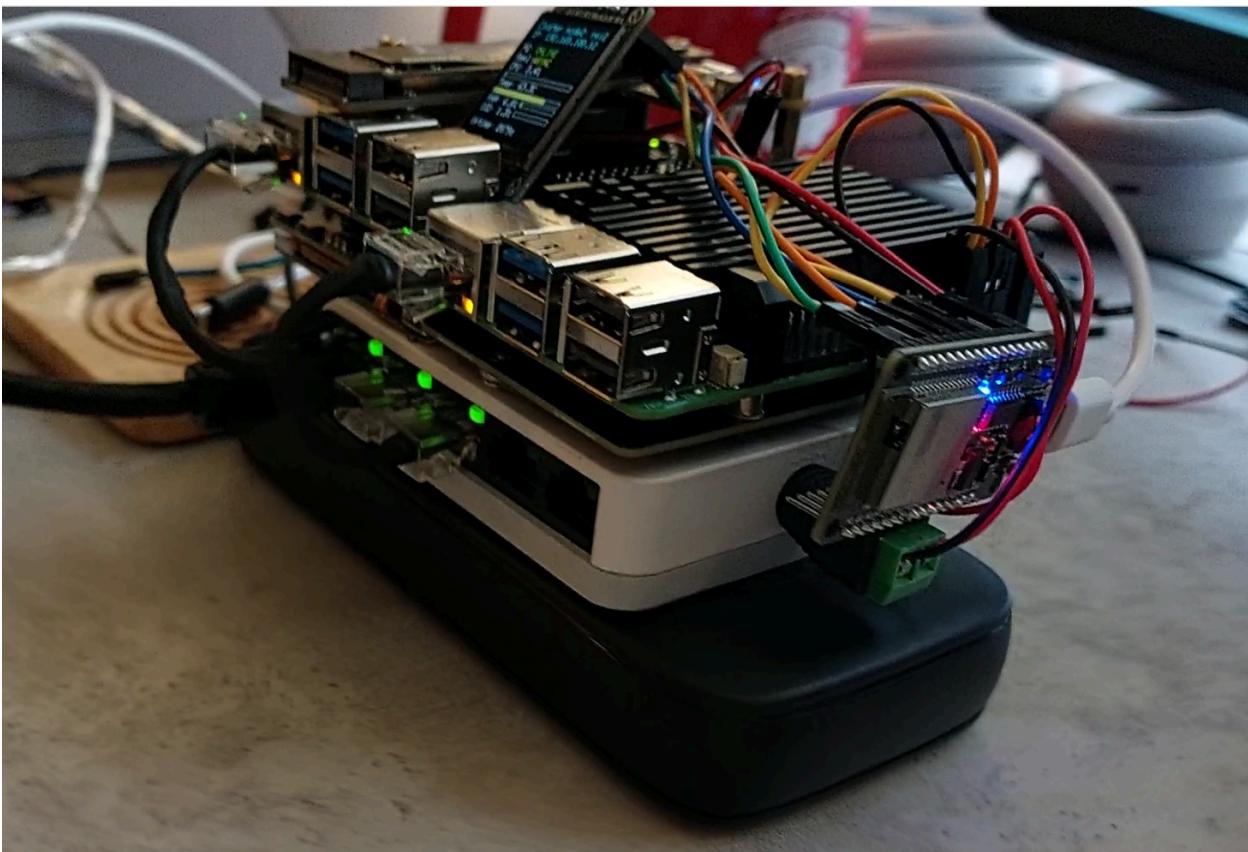
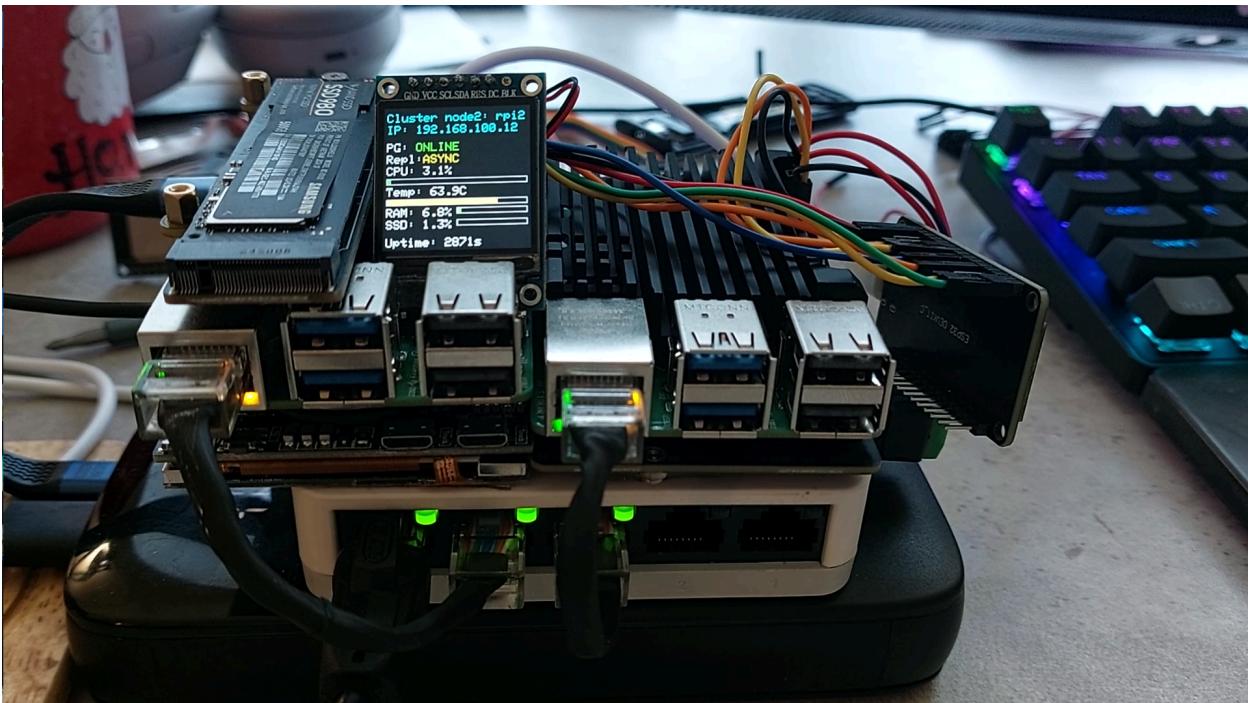
- Восстановить работу основного узла - откатить действие, выполненное с виртуальной машиной на этапе 2.2.

- Актуализировать состояние базы на основном узле - накатить все изменения данных, выполненные на этапе 2.3.
- Восстановить исправную работу узлов в исходной конфигурации (в соответствии с этапом 1).
- Продемонстрировать состояние данных и работу клиентов в режиме чтение/запись.

## Выполнение

### Этап 1 Конфигурация

## 1.1 Физическая организация кластера



В этой лабораторной работе предстоит настроить репликацию на собственных узлах, и так как я системщик и мне скучно, то я решил а почему бы не сделать собственный кластер на малинках. Описание конфигурации - узлы В и С - резервные - я сделаю на raspberry pi 5 (cpu: cortex a76, ram: LPDDR4X 8gb + 512gb nvme ssd) (как бы симулируя не особо производительное железо для резервных узлов), назначу им статические IPv4 и соединю в сеть с ноутбуком (узел А - основной), который будет играть роль основного узла (cpu: r7 5800h, ram: LPDDR4X 16gb, 2tb ssd), с помощью самого дешёвого концентратора который у меня валялся в ящике.

Тестировал производительность в домашней сети, где был гигабитный интернет и dhcp на роутере, и хоть в жизни серверы будут находиться в похожих условиях, мне было интересно сделать кластер переносным, иначе не работал бы в embedded. Именно поэтому я назначил узлам статические ip и вывел их на экранчики, посыпая по uart/i2c системные данные для мониторинга каждого узла на свою esp32 и показывая по spi шине на мелком ips экране (*типа уменьшил нагрузку на вычислительные гр5 узлы, заставив рисовать esp32, но по факту меня просто задолбали кривые либы на малинку. Её GPIO более чем достаточно, но на esp32 всё легко зовести, а на гр5 ужас*).

Питает это всё дело power bank с фичей power delivery, а соответственно по 20-25w каждой малине, плюс 5w на концентратор он спокойно выдаёт (заявлен потолок в 100w), и по мелочи для esp32 (~ 0.15w на обе)

Способ подключения к узлам:

1. Основной узел: `ssh boris@192.168.100.1`
2. Резервный узел В: `ssh boris@192.168.100.11`
2. Резервный узел В: `ssh boris@192.168.100.12`

```
└─[boris at fedora in /]
└─[λ rpi1 --show
ssh boris@192.168.100.11
└─[boris at fedora in /
└─[λ rpi2 --show
ssh boris@192.168.100.12
```

```
enp3s0f4u1u1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 192.168.100.1 netmask 255.255.255.0 broadcast 192.168.100.255
    inet6 fe80::ac94:d371:4223:8f36 prefixlen 64 scopeid 0x20<link>
        ether 00:e0:4c:3b:04:30 txqueuelen 1000 (Ethernet)
        RX packets 7972 bytes 1804004 (1.7 MiB)
        RX errors 0 dropped 0 overruns 0 frame 0
        TX packets 9143 bytes 842027 (822.2 KiB)
        TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

UPD: (пересел на ubuntu из-за злой\*учего роупол2)

```
└─[boris at boris-CREM-WXX9 in ✘
└─[λ sudo nmcli connection add \
    type ethernet \
    con-name cluster \
    ifname enx00e04c3b0430 \
    ip4 192.168.100.1/24 \
    autoconnect[]

└─[boris at boris-CREM-WXX9 in ✘
└─[λ cat /etc/hosts
127.0.0.1 localhost
127.0.1.1 boris-CREM-WXX9

# The following lines are desirable for IPv6 capable hosts
::1      ip6-localhost ip6-loopback
fe00::0  ip6-localnet
ff00::0  ip6-mcastprefix
ff02::1  ip6-allnodes
ff02::2  ip6-allrouters

192.168.100.1  laptop postgres-primary
192.168.100.11  rpi1 postgres-standby1
192.168.100.12  rpi2 postgres-standby2
```

## ДАЛЕЕ БУДУТ НАСТРОЙКИ ДЛЯ FEDORA WORKSTATION 39!

### 1.2 Настройка узлов

Для начала копируем ssh-ключи, настраиваем оболочки и [оборачиваем pgpool и постгрес в докер на основном узле]. Необходимость обёртки постгреса в докер не такая явная, но это правило хорошего тона, чтобы не ломать собственную систему.

```
# На хосте с Docker
docker exec -it postgres-primary bash -c "mkdir -p /tmp/backup"
docker exec -it postgres-primary pg_basebackup -D /tmp/backup -Fp -X
stream -v -P -U postgres

# Копируем из контейнера на хост
docker cp postgres-primary:/tmp/backup ./backup

# Копируем на обе малинки
scp -r ./backup boris@192.168.100.11:/tmp/pg_backup
scp -r ./backup boris@192.168.100.12:/tmp/pg_backup
```

Далее на каждой малинке обновляем кластер

```
# Остановка PostgreSQL
sudo systemctl stop postgresql

# Очистка директории данных
sudo rm -rf /var/lib/postgresql/15/main/

sudo mkdir /var/lib/postgresql/15/main

# Копирование базового бэкапа
sudo cp -r /tmp/pg_backup/* /var/lib/postgresql/15/main/

# Установка прав
sudo chown -R postgres:postgres /var/lib/postgresql/15/main
sudo chmod 700 /var/lib/postgresql/15/main
```

Настройка репликации для машины 1 (синхронная):

```
# Создаем standby.signal
sudo -u postgres touch /var/lib/postgresql/15/main/standby.signal

# Настраиваем конфигурацию репликации
sudo -u postgres bash -c 'cat >
/var/lib/postgresql/15/main/postgresql.auto.conf << EOF
primary_conninfo = """host=192.168.100.1 port=5432 user=postgres
password=postgres application_name=standby_1"""
primary_slot_name = """standby_1"""
hot_standby = on
EOF'

# Запускаем PostgreSQL
sudo systemctl start postgresql
```

Настройка репликации для машины 2 (асинхронная с задержкой):

```
# Создаем standby.signal
sudo -u postgres touch /var/lib/postgresql/15/main/standby.signal

# Настраиваем конфигурацию репликации
sudo -u postgres bash -c 'cat >
/var/lib/postgresql/15/main/postgresql.auto.conf << EOF
primary_conninfo = """host=192.168.100.1 port=5432 user=postgres
password=postgres application_name=standby_2"""
primary_slot_name = """standby_2"""
hot_standby = on
recovery_min_apply_delay = """3s"""
EOF'

# Запускаем PostgreSQL
sudo systemctl start postgresql
```

На основном узле создаем слоты репликации:

```
docker exec -it postgres-primary psql -U postgres -c "SELECT * FROM
```

```
pg_create_physical_replication_slot('standby_1', true);"  
docker exec -it postgres-primary psql -U postgres -c "SELECT * FROM  
pg_create_physical_replication_slot('standby_2', true);"  
  
# Проверяем статус репликации  
docker exec -it postgres-primary psql -U postgres -c "SELECT  
application_name, client_addr, state, sync_state FROM  
pg_stat_replication;"
```

Небольшая теоретическая пометочка:

Назначение standby.signal:

- Это пустой файл-маркер, который сообщает PostgreSQL, что экземпляр должен запуститься в режиме реплики
- PostgreSQL ищет этот файл при запуске в корне директории данных
- До PostgreSQL 12 использовался файл recovery.conf, сейчас это комбинация standby.signal + настройки в postgresql.auto.conf

По режимам репликации: ASYNC используется по умолчанию для всех standby узлов, а SYNC явно настраивается через `synchronous\_standby\_names` на primary узле.

Primary узел не "находит" replica узлы самостоятельно - это replica подключаются к primary с помощью параметра **primary\_conninfo**

Создадим тестовую таблицу для репликации:

(на primary)

```
CREATE TABLE replication_test ( id SERIAL PRIMARY KEY, data TEXT, );  
  
INSERT INTO replication_test (data) VALUES ('test-entry');
```

Проверим результатом репликации на обоих standby:

```
SELECT NOW() AS current_time, * FROM replication_test;
```

Сорян за мелкий шрифт, но если увеличить то всё видно:

```
boris at raspberrypi in ~
└─$ db
could not change directory to "/home/boris": Permission denied
pgsql (15.12 (Debian 15.12-0+deb12u2))
Type "help" for help.

postgres# SELECT NOW() AS current_time, * FROM replication_test;
      current_time | id | data | created_at
-----+-----+-----+-----+
2025-05-19 13:52:13.781715+03 | 1 | test-entry | 2025-05-19 13:51:46.362677+03
(1 row)

postgres#
```

```
boris at raspberrypi in ~
└─$ db
could not change directory to "/home/boris": Permission denied
pgsql (15.12 (Debian 15.12-0+deb12u2))
Type "help" for help.

postgres# SELECT NOW() AS current_time, * FROM replication_test;
      current_time | id | data | created_at
-----+-----+-----+-----+
2025-05-19 13:52:55.428101+03 | 1 | test-entry | 2025-05-19 13:51:46.362677+03
(1 row)

postgres#
```

```
boris at fedora in ~/dev/pgsql-cluster
└─$ db
pgsql (15.13 (Debian 15.13-1.pgdg120+1))
Type "help" for help.

postgres# CREATE TABLE replication_test ( id SERIAL PRIMARY KEY, data TEXT, created_at TIMESTAMPZ DEFAULT NOW());
CREATE TABLE
INSERT INTO replication_test (data) VALUES ('test-entry');
CREATE TABLE
INSERT 0 1
postgres# SELECT NOW() AS current_time, * FROM replication_test;
      current_time | id | data | created_at
-----+-----+-----+-----+
2025-05-19 13:51:47.335198+03 | 1 | test-entry | 2025-05-19 13:51:46.362677+03
(1 row)

postgres#
```

### 1.3 Настройка pgpool-II

```
docker exec -it pgpool bash -c "cat
/opt/bitnami/pgpool/conf/pgpool.conf"
```

```
backend_hostname0 = '192.168.100.1'
backend_port0 = 5432
backend_weight0 = 1
backend_data_directory0 = '/opt/bitnami/pgpool/data'
backend_flag0 = 'ALLOW_TO_FAILOVER'
backend_hostname1 = '192.168.100.11'
backend_port1 = 5432
backend_weight1 = 1
backend_data_directory1 = '/opt/bitnami/pgpool/data'
```

```
backend_flag1 = 'ALLOW_TO_FAILOVER'
backend_hostname2 = '192.168.100.12'
backend_port2 = 5432
backend_weight2 = 1
backend_data_directory2 = '/opt/bitnami/pgpool/data'
backend_flag2 = 'ALLOW_TO_FAILOVER'
```

Примечательно, что сама по себе настройка pgpool происходит за счёт переменных окружения в docker compose скрипте

```
  ↵ docker-compose.yml ✘
  ↗ services > ↗ pgpool > [] environment > ... 16
  ↘ 27   pgpool:
  ↗ 26     image: pgpool-ssh
  ↗ 25     container_name: pgpool
  ↗ 24     profiles: ['pgpool', 'all']
  ↘ 23     ports:
  ↗ 22       - '0.0.0.0:5433:5432'
  ↗ 21     environment:
  ↗ 20       - PGPOOL_ADMIN_USERNAME=admin
  ↗ 19       - PGPOOL_ADMIN_PASSWORD=admin
  ↗ 18       - PGPOOL_POSTGRES_USERNAME=postgres
  ↗ 17       - PGPOOL_POSTGRES_PASSWORD=postgres
  ↗ 16
  ↗ 15       - PGPOOL_BACKEND_NODES=0:192.168.100.1:5432,1:192.168.100.11:5432,2:192.168.100.12:5432
  ↗ 14
  ↗ 13       - PGPOOL_SR_CHECK_USER=postgres
  ↗ 12       - PGPOOL_SR_CHECK_PASSWORD=postgres
  ↗ 11       - PGPOOL_HEALTH_CHECK_USER=postgres
  ↗ 10       - PGPOOL_HEALTH_CHECK_PASSWORD=postgres
  ↗ 9       - PGPOOL_HEALTH_CHECK_PERIOD=5
  ↗ 8       - PGPOOL_HEALTH_CHECK_MAX_RETRIES=2
  ↗ 7
  ↗ 6       - PGPOOL_FAILOVER_ON_BACKEND_ERROR=on
  ↗ 5       - PGPOOL_AUTO_FAILBACK=no
  ↗ 4       - PGPOOL_BACKEND_FLAG0=ALLOW_TO_FAILOVER
  ↗ 3       - PGPOOL_BACKEND_FLAG1=ALLOW_TO_FAILOVER
  ↗ 2       - PGPOOL_BACKEND_FLAG2=ALLOW_TO_FAILOVER
  ↗ 1
  ↗ 53      - PGPOOL_FAILOVER_COMMAND=/bin/bash /etc/pgpool2/failover.sh %d %h %p %D %m %H %M %P %r %R
```

```
boris at fedora in ~/dev/pgsql-cluster
└ λ PGPASSWORD=postgres psql -h localhost -p 5433 -U postgres -c "\x" -c "SHOW pool_nodes;"
```

Expanded display is on.

```
-[ RECORD 1 ]-----+
node_id          | 0
hostname         | 192.168.100.1
port             | 5432
status            | up
pg_status         | up
lb_weight         | 0.333333
role              | primary
pg_role           | primary
select_cnt        | 1
load_balance_node | true
replication_delay | 0
replication_state |
replication_sync_state |
last_status_change | 2025-05-19 11:45:16
-[ RECORD 2 ]-----+
```

```
- [ RECORD 2 ]-----+
node_id           | 1
hostname          | 192.168.100.11
port              | 5432
status             | down
pg_status          | up
lb_weight          | 0.333333
role               | standby
pg_role            | unknown
select_cnt         | 0
load_balance_node | false
replication_delay | 0
replication_state |
replication_sync_state |
last_status_change | 2025-05-19 11:43:48
```

```
- [ RECORD 3 ] -----
node_id          | 2
hostname         | 192.168.100.12
port             | 5432
status            | down
pg_status         | down
lb_weight         | 0.333333
role              | standby
pg_role           | unknown
select_cnt        | 0
load_balance_node | false
replication_delay | 0
replication_state | 
replication_sync_state | 
last_status_change | 2025-05-19 11:43:48
```

## 1.3 РАБОТАЕТ

```
postgres=# SELECT application_name, sync_state, client_addr
  FROM pg_stat_replication;
   application_name | sync_state | client_addr
-----+-----+-----
  standby_2          | async      | 192.168.100.12
  standby_1          | sync       | 192.168.100.11
(2 rows)
```

```
postgres=# SELECT
    now() AS current_time,
    pg_last_xact_replay_timestamp() AS last_replay,
    now() - pg_last_xact_replay_timestamp() AS replay_lag;
   current_time | last_replay | replay_lag
-----+-----+-----
2025-05-19 14:16:13.966249+03 | 2025-05-19 14:16:12.722243+03 | 00:00:01.244006
(1 row)

postgres# [REDACTED]
(1 row)

postgres# SELECT
    now() AS current_time,
    pg_last_xact_replay_timestamp() AS last_replay,
    now() - pg_last_xact_replay_timestamp() AS replay_lag;^C
postgres# SELECT
    now() AS current_time,
    pg_last_xact_replay_timestamp() AS last_replay,
    now() - pg_last_xact_replay_timestamp() AS replay_lag;
   current_time | last_replay | replay_lag
-----+-----+-----
2025-05-19 14:15:43.943182+03 | 2025-05-19 14:15:36.117811+03 | 00:00:07.825371
(1 row)

postgres# SELECT
    now() AS current_time,
    pg_last_xact_replay_timestamp() AS last_replay,
    now() - pg_last_xact_replay_timestamp() AS replay_lag;
   current_time | last_replay | replay_lag
-----+-----+-----
2025-05-19 14:16:13.131291+03 | 2025-05-19 14:16:11.129011+03 | 00:00:02.00228
(1 row)

postgres# [REDACTED]
```

```
postgres=# DO $$ BEGIN
    FOR i IN 1..1000 LOOP
        INSERT INTO delay_test (id) VALUES (i);
        COMMIT;
    END LOOP;
END $$;
DO
postgres=# DO $$ BEGIN
    FOR i IN 1..1000 LOOP
        INSERT INTO delay_test (id) VALUES (i);
        COMMIT;
    END LOOP;
END $$;
DO
postgres=# DO $$ BEGIN
    FOR i IN 1..3000 LOOP
        INSERT INTO delay_test (id) VALUES (i);
        COMMIT;
    END LOOP;
END $$;
DO
postgres#
```

Я постарался показать задержку асинхронного узла, произвождя имитацию нагрузки (инсерт 3к строк в табличку). С помощью `pg_last_xact_replay_timestamp()` я вывел тайминг последней транзакции на реплике и видно что асинхронный узел, который внизу, чуток отстает. По факту нормального способа проверить деляй между синхронным и асинхронным узлом нету, не нагружив их просто так до отказа.

Тестирование репликации (Этап 1):

Уже сделал выше

## Этап 2: Симуляция и обработка сбоя

Подготовка клиентских подключений:

```
# Проверяем статус узлов
PGPASSWORD=postgres psql -h Localhost -p 5433 -U postgres -d postgres
-c "SHOW pool_nodes;"
```

```
# Или так:
PGPASSWORD=postgres psql -h Localhost -p 5433 -U postgres -c "\x" -c
"SHOW pool_nodes;"
```

```
boris at fedora in ~/dev/pgsql-cluster
└─$ PGPASSWORD=postgres psql -h localhost -p 5433 -U postgres -d postgres -c "SHOW pool_nodes;"
```

node_id	hostname	port	status	pg_status	lb_weight	role	pg_role	select_cnt	load_balance_node	replication_delay	replication_state	replication_sync_state	last_status_change
0	192.168.100.1	5432	up	up	0.333333	primary	primary	0					2025-05-19 12:15:33
1	192.168.100.11	5432	up	up	0.333333	standby	standby	0					2025-05-19 12:15:33
2	192.168.100.12	5432	up	up	0.333333	standby	standby	0					2025-05-19 12:15:33

(3 rows)

Видим что все “up” -> отлично, всё работает.

Симуляция сбоя:

```
# Перед симуляцией сбоя, создадим тестовую таблицу
PGPASSWORD=postgres psql -h Localhost -p 5433 -U postgres -c "CREATE
TABLE failover_test (id SERIAL, data TEXT, created_at TIMESTAMPTZ
DEFAULT NOW());"

PGPASSWORD=postgres psql -h Localhost -p 5433 -U postgres -c "INSERT
INTO failover_test (data) VALUES ('pre-failover data');"

# Отказ primary узла
docker compose --profile postgres stop
sudo rm -rf ./pgdata-primary

# Логи pgpool
docker logs pgpool -f
```

## Логи и Переключение:

```
# Переключение должно было произойти само через failover.sh  
# Но если что, демонстративно переключать приоритеты узлов можно  
вот так:  
ssh boris@192.168.100.11 "sudo -u postgres pg_ctlcluster 15 main  
promote"  
  
# Проверяем, что pgpool подключается к новому основному узлу  
PGPASSWORD=postgres psql -h Localhost -p 5433 -U postgres -d postgres  
-c "SHOW pool_nodes;"  
PGPASSWORD=postgres psql -h Localhost -p 5433 -U postgres -c "INSERT  
INTO failover_test (data) VALUES ('post-failover data');"  
PGPASSWORD=postgres psql -h Localhost -p 5433 -U postgres -c "SELECT  
* FROM failover_test;"
```

## Этап 3: Восстановление

```
sudo chmod -R 755 /tmp/backup  
  
# Делаем бэкап  
sudo -u postgres pg_basebackup -h 127.0.0.1 -p 5432 -U postgres -D  
/tmp/backup -Fp -X stream -v -P
```

## На ноуте:

```
scp -r boris@192.168.100.11:/tmp/backup backup/  
sudo cp -r backup/backup/* ./pgdata-primary/  
sudo touch ./pgdata-primary/recovery.signal  
sudo chown -R 999:999 ./pgdata-primary  
docker compose --profile postgres up -d
```

Скриншотов того как красиво система просыпается около 15 секунд делая health чеки и потом делает failover и standby становится primary у меня не осталось, я думал что успею наделать ещё, но тысяча проблем в наносекунду, образующихся по ходу работы с постгресом, особенно на арме raspberry pi 5 плат, мне чуть помешали 😊

UPD2: Вся настройка была произведена с нуля на системе Ubuntu 24.04.2 LTS. Подробности я прикладывать, конечно же, не буду :D

Самое важное это pgpool2. Я почитал доку, узнал про управление с помощью pcp (pgpool control protocol) и написал нормальную конфигу. Ну а потом, после выставления нормальной аутентификации в конфигах pg\_hba.conf, pool\_hba.conf, pcp.conf - всё заработало корректно.

Новый pgpool2:

```
backend_clustering_mode = 'streaming_replication'
master_slave_mode = on
master_slave_sub_mode = 'stream'

# Connection
listen_addresses = '*'
port = 5433
socket_dir = '/var/run/postgresql'

backend_hostname0 = '192.168.100.1'
backend_port0 = 5432
backend_weight0 = 1
backend_data_directory0 = '/var/lib/postgresql/16/main'
backend_flag0 = 'ALLOW_TO_FAILOVER'
backend_application_name0 = 'laptop'

backend_hostname1 = '192.168.100.11'
backend_port1 = 5432
backend_weight1 = 1
backend_data_directory1 = '/var/lib/postgresql/16/main'
backend_flag1 = 'ALLOW_TO_FAILOVER'
backend_application_name1 = 'rpi1'
```

```
backend_hostname2 = '192.168.100.12'
backend_port2 = 5432
backend_weight2 = 1
backend_data_directory2 = '/var/lib/postgresql/16/main'
backend_flag2 = 'ALLOW_TO_FAILOVER'
backend_application_name2 = 'rpi2'

# Pool settings
num_init_children = 32
max_pool = 4

# Replication check
sr_check_period = 10
sr_check_user = 'postgres'
sr_check_password = 'postgres'
sr_check_database = 'postgres'
delay_threshold = 10000000

# Health check
health_check_period = 30
health_check_timeout = 15
health_check_user = 'replicator'
health_check_password = 'password'
health_check_database = 'postgres'

# Load balancing
load_balance_mode = on
ignore_leading_white_space = on
white_function_list = ''
black_function_list = 'currval, lastval, nextval, setval'

# PCP
pcp_listen_addresses = '*'
pcp_port = 9898
pcp_socket_dir = '/var/run/postgresql'

# Auth
enable_pool_hba = off
pool_passwd = ''
```

```
allow_clear_text_frontend_auth = on

# Logging
log_destination = 'sysLog'
log_connections = on
log_statement = on
log_per_node_statement = on
log_min_messages = info
```

## Вывод

### Конфигурация

#### Габариты

Потенциально уменьшить по габаритам данный сетап можно купив PoE hat, PoE коммутатор и модуль питания через акумы.

Проводочеков будет меньше, всего лишь по одному rj45 на плату, супер чисто и красиво, но проблем потенциально больше, да и больше полусотни на небольшой проект тратить не хочется 😊

Также можно было на одной из малинок снять экранчик, который я так старательноставил на ножки. Он сенсорный и супер удобен в повседневке, но в кластере не нужен. Оставил просто потому что лень снимать, и пространство на которое он поднимает всю плату как раз идеально подходит чтобы в него протянуть typec кабель для питания второй малинки. Если их просто поставить рядом, то коннекторы упираются друг в друга.

#### Удешевление

Можно юзать обычные sd карточки по сколько нужно gb в совокупности с гри4. Там нет pcie, поэтому без ssd будет всё работать сильно медленнее, но сетап выйдет в 3 раза дешевле, т.к. не нужны ssd m.2 hat (2000р.), сам ssd (7000р.), и более старая версия малины, всего лишь в 2 раза проигрывающая по производительности в сри (6000р против 9000р за гри5). У меня уже была гри5 и с ssd этом сетап наиболее приближается к реальному, поэтому так интереснее.

Минимальные системные требования на postgres кластер - 2gb ram, 512 mb hdd, 1 GHz cpu.

Cortex A76 работает на 92 GHz, тут вопросов никаких нет.

Rpi4 платы можно брать на 2-4 гигабайта и хоть в реальном мире этого будет мало, но постгрес в теории может работать и такая нода в кластере в качестве резерва не будет лишней. Экономия огромная (минус 2000р. За версию на 4gb.)

По накопителю аналогично - можно взять на 8gb и туда поместится и ось и постгрес. Но место для бэкапа закончится через 1 секунду)

## Узкие места

Мой помойный концентратор, к сожалению, работает на 100Мбит/с, в то время как максимальная зафиксированная скорость с ssd'шников была >1 GT/s, что в совокупности с гигабитными ethernet портами на rpi5 и гигабитным коммутатором дало бы просто отличные скорости, конкурирующие с текущими бэкендами в продакшене. Не брал я гигабитный коммутатор очевидно почему - этом проектик абсолютно никак не пострадает от скоростей в 100Мбит/с, а снимать основной коммутатор из дома, или покупать ещё один за просто так выходит за рамки бюджета на этом проект. Он и так многократно превысил ожидания)

## Софтверная часть

UPD: ВСЯ проблема этой части - тут написанный дегенератами  
-> pgpool2 <-

Первичный сепарант (fedora workstation 39):

В целом я всё организовал достаточно грамотно. Зависимости pgpool2 тупо нету на fedora39, поэтому я её завернул в докер; башку я завернул в докер на основном узле просто для удобства, потому что с ней можно как в песочнице делать что угодно, мониторить и управлять жизненным циклом средствами докера. И аналогично с pgpool. В целом, у меня была идея и она была бы тут не лишней - т.к. я недавно установил ssd на 2 терабайта в ноут, я бы мог накатить ubuntu рябушком и на ней нативно всё развернуть. Проблем было

бы в теории сильно меньше, но это путь для слабых. Докер интересно тыкать. Главное с ума не сойти 😊

По поводу поднятия башек на малинках - всё же надо было имхо поднимать их тоже в докере. Конечно нативно тыкаться круто и приятно, но ровно до того момента, как всё перестанет работать с новой randomной ошибкой, которую надо рассчитать и фиксить. Просто взять и пересобрать контейнер было бы куда проще, особенно в домашних условиях с гигабитом. Конфигурации я собрал достаточно мощные.

Самых по себе проблем достаточно много и тут отчёта не хватит описывать всё что я пофиксил и переделал, но могу постараться уместить их в определённые группы:

1. Проблемы архитектуры (`arm` + специфическая ось) => другое управление кластером (`pg_ctl` просто нету), иногда что-то не собирается, мелкие проблемы относящиеся именно к специфическим зависимостям.
2. Проблемы прав доступа: докер инкапсулирует; постгрес требует своих 0700 на `$PGDATA`, debian и fedora имеют свои фаерволы и роли, поэтому постоянно нужно делать `chown` и `chmod` чтобы банально сделать бэкап, запустить корректно кластер, чтоб мог корректно работать `rdpool2` и т.п.
3. Проблемы `rdpool2`: Не слишком гибкая утилита и если её запустить и кластер запущен не в той конфигурации в которой нужно или в нём что-то хотя бы чуть-чуть не так, начинает сходить с ума, и может начать ломать кластеры в соответствии с [`failover.sh`](#) скриптом или своими прочими действиями, пытаясь как бы их "восстановить".
4. Фантомные проблемы: такие проблемы сложно куда-то отнести, я бы назвал их "skill issue". Тыркаешь постгрес - он падает с каким-то бредом. Оказывается что надо инитнуть кластер с правильной локалью. Отсоединяешь узел от коммутатора и подсоединяешь обратно и вся бд перестаёт работать с левой ошибкой, или просто реинитишь всё железо и наоборот из состояния краснющих `eggos` и `fatal` ошибок всё рекавериться в идеальную согласованную работу. И всё это зависит от каких-то императивных команд которые надо ручками прописать. Напоминает чем-то `wildfly`, не должно быть всё так криво косо и страхолюдно.

Урд: Вторичный сетан (Ubuntu 24.04.2 LTS):

Да, я сдался и поставил рядышком Ubuntu 24.04.2 LTS. На ней было куда проще настроить сетевую часть и разобраться с зависимостями - rfroo12 ставился "нативно" (через встроенный apt), что помогло при дебаге исключить докер из возможных усложняющих прослоек проекта. Хосты просто указываются в /etc/hosts, для wired интерфейса я создал тривиальное правило (nmcli):

```
boris at boris-CREM-WXX9 in ~
λ sudo nmcli connection add \
    type ethernet \
    con-name cluster \
    ifname enx00e04c3b0430 \
    ip4 192.168.100.1/24 \
    autoconnect
```

В целом, конечно, проблема была не в ОС, а в тупе. Но как опыт мне хотелось нативно потыкать последнюю убунту с новым гномом, и заодно я круто настроил grub2:

