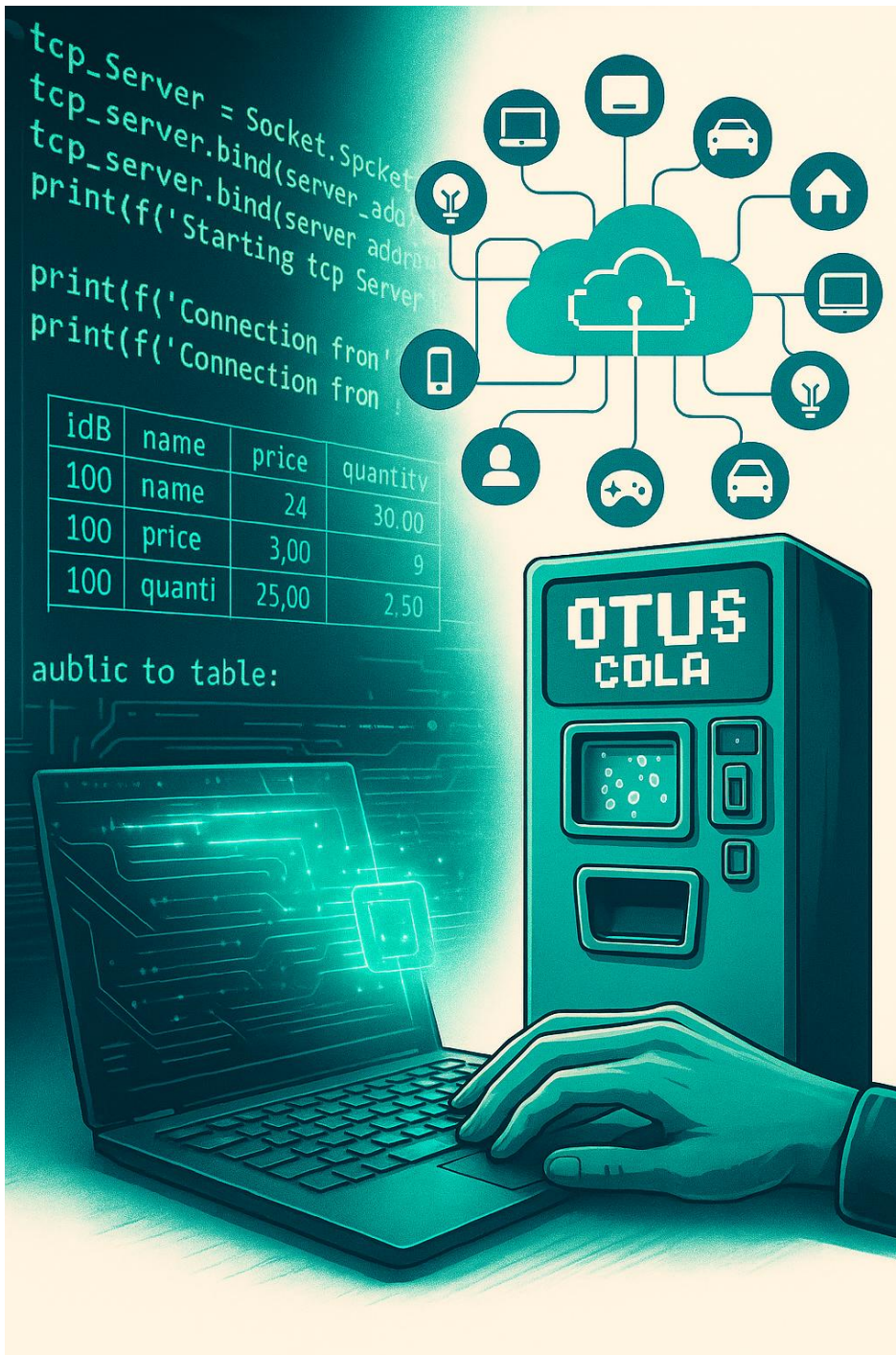


Дипломная работа

«Высоконагруженная эффективная платформа управления вендинговыми устройствами: Архитектура и реализация»



Провайдер: OTUS
Курс: Highload Architect
Студент: Томшин И.М.
Руководитель: Юшкевич Виталий

Защита: 16 сентября 2025г

- Стек:
- C++
 - TLV
 - Redis
 - PostgreSQL
 - Шардирование
 - Репликация
 - Ext JS
 - PHP

Анотация

Работа посвящена проектированию и реализации высоконагруженной платформы управления вендинговыми устройствами. Цель платформы — обеспечить надёжный приём телеметрии и команд по TCP, низкую задержку обработки потока сообщений в формате TLV, кэширование горячих данных в Redis, устойчивое хранение данных в PostgreSQL с горизонтальным масштабированием (шардирование) и отказоустойчивостью (репликация), агрегирование данных для аналитической обработки. Рассмотрены подходы к протоколам, моделям данных, масштабируемости и построению системы back-office. Разработаны архитектурные решения ключевых компонентов системы, приведены демонстрации и результаты нагрузочного профилирования. Отличительной особенностью проекта является стремление использования наиболее простых технологий, не требующих значительных затрат на администрирование системы.

Введение

Современные вендинговые аппараты — это IoT-узлы со множеством датчиков: температурные и расходомерные сенсоры, купюроприёмники/валидаторы, NFC/EMV-терминалы, электромагнитные клапаны, перистальтические насосы и пр. Для операторов системы важно:

- наблюдать состояние и телеметрию в реальном времени;
- удалённо конфигурировать устройства (стоимость, работу);
- быстро реагировать на инциденты и простои;
- проводить аналитику продаж, брака, топологии размещения.

Проектирование

Проектирование системы будет вестись с использованием подхода построения микросервисной архитектуры.

Архитектура системы

- 1.1** Программное обеспечение для IoT устройства (вендинговый аппарат)
- 1.2** Прием, обработка и хранение данных (серверная часть)
 - 1.2.1** TCP сервер приема данных
 - 1.2.2** Кеширование Redis
 - 1.2.3** Парсинг данных, структура БД PostgreSQL и шардирование, агрегация
- 1.3** Backoffice и интеграция с внешними системами.



Этап 1.1. IoT устройства.

Референс

IoT устройство — **Контроллер**, управляющий вендинговым аппаратом имеющий выход в интернет, HMI интерфейс, входы телеметрии и выходы для управления исполнительными механизмами.

Контроллер передаёт следующий набор данных:

- Уникальный идентификатор устройства UID;
- Давление воды в сети, бар.;
- Сумму покупки, коп.;
- Кол-во отлитой воды, мл.;
- Показания счетчика входной воды;

Контроллер бережно относится к трафику, поэтому:

- При бездействии передаёт данные 1 раз в 2 минуты (ping);
- Передаёт только изменяемые данные, с момента последней передачи;
- Передаёт весь набор данных после загрузки;

В условиях нестабильного интернета контроллер хранит критически важные данные (продажу), если они небыли приняты сервером до следующего сеанса связи. Некритические данные (ping) не сохраняются.

Выбор транспорта

- Протокол TCP надёжен, доступен для модемов LTE, поддерживает длительные сессии;
- Для экономии трафика и универсальности будет использоваться компактный, бинарный формат TLV (Type-Length-Value). В сравнении с JSON, формат TLV более компактный и его парсинг требует значительно меньших ресурсов;
- Для защиты данных возможно использовать шифрование — TLS (за рамками проекта);
- Так как не предполагается массированный обмен данными Контроллера с сервером, соединение не будет сохраняться открытым, каждая передача данных будет инициировать новое соединение. Это позволит экономить серверные ресурсы;

Формат кадра



Хозяйке на заметку: В рамках демонстрации проекта рассматривается упрощенная версия TLV. Размер сообщения не более 65535 байт, размер тэгов 1 байт, длина тэгов определяется 1 байтом. Длин шаблона определяется фиксировано 2 байтами.

Frame := Template(1) | Tag1(4) | Tag2(4) | Tag3(4) | CRC16(2)

Template – шаблон протокола, 1 байт. / 0x0/

COUNT –Номер сообщения – 4 байта;

UID - Уникальный идентификатор устройства – 4 байт;

Tag1 TagXX – тэги с данными. Для экономии трафика, примем часто используемые тэги (0-7F) длиной 1 байт. Другие тэги 0x8000-0xFFFF примем как 2-х байтные. В итоге контроллер сможет получать и передавать 32894 тэга (параметра), достаточных с большим запасом для взаимодействия Контроллера с сервером.

Пример пакета данных Клиента -> Серверу.

Template	Len
0x00	0x1C

TAG_COUNT	Len	DATA_COUNT			
0x01	0x04	0x00	0x00	0x00	0x01

TAG_UID	Len	DATA_UID			
0x02	0x04	0x12	0x34	0x56	0x78

TAG_PRESSURE	Len	DATA_PRESSURE	
0x10	0x02	0x01	0x00

TAG_SUMM	Len	DATA_SUMM	
0x11	0x02	0x08	0x00

TAG_VAL	Len	DATA_VAL	
0x12	0x02	0x01	0x00

TAG_COUNT1	Len	DATA_COUNT1			
0x13	0x04	0x12	0x34	0x56	0x78

TAG_CRC16	Len	DATA_CRC16	
0x7F	0x02	0x02	0x68

Итоговый пакет:

Count = 0x01; (Номер сообщения)

DeviceUID = 0x12345678; (Идентификатор Контроллера)

Pressure = 0x0100; (Давление воды: 0,256 бар)

Sum = 0x0800; (Сумма: 20 рублей 48 коп.)

Val = 0x0100; (Отливо: 256 мл.)

Count1 = 0x12345678; (Показания счётчика воды: 0x12345678)

CRC16 = 0x0268; (Контрольная сумма)

0x00, 0x1E, 0x00, 0x04, 0x12, 0x34, 0x56, 0x78, 0x01, 0x02, 0x01, 0x00, 0x02, 0x02, 0x08, 0x00, 0x03, 0x02, 0x01, 0x00, 0x04,
0x04, 0x12, 0x34, 0x56, 0x78, 0x7F, 0x02, 0x02, 0x68

При успешном приеме сообщения сервер отвечает Контроллеру командой, что данное сообщение (с номером count) принято успешно.

Пример пакет данных Сервера -> Клиенту.

Template	Len
0x00	0x1C

TAG_COUNT	Len	DATA_COUNT			
0x01	0x04	0x00	0x00	0x00	0x01



TAG_CRC16	Len	DATA_CRC16	
0x7F	0x02	0x02	0x68

Балансировка обмена

Укажем в DNS множество A-записей для домена gate.vendotus.com, чтобы Контроллер сам выбирал сервер из нескольких, для балансировки TCP-нагрузки.

```
gate.vendotus.com A 192.168.1.0
gate.vendotus.com A 192.168.1.1
gate.vendotus.com A 192.168.1.2
```

Контроллер при обмене данными с сервером будет использовать сценарий:

1. Делает DNS-запрос getaddrinfo("gate.vendotus.com") и получается все IP-адреса в ответ.
2. Выбирает случайным образом IP из списка.
3. Устанавливает TCP соединение с выбранным IP, производит обмен данными;
4. Если обмен прошёл успешно, в следующий сеанс обмена переходит к шагу 2.
5. Если обмен по всем IP адресам произошёл неуспешно, переходит к шагу 1.

Плюсы такого подхода, в сравнении с использованием HAProxy для TCP балансировки:

- Минимальная простота инфраструктуры, не требуется развернуть и обслуживать HAProxy;
- Максимально возможная пропускная способность (RPS);
- Минимальное количество хопов (1 соединение напрямую);
- Latency (задержка) – нулевая, соединение с сервером напрямую;
- Возможность использования комбинированного подхода в дальнейшем, указав в DNS единственную A-запись для домена gate.vendotus.com и развернув на сервере HAProxy для балансировки нагрузки;

Минусы:

- При высокой нагрузке (1000+ RPS) HAProxy может быть устойчивее, даже если немного медленнее;
- Нетиповой подход;

MQTT?



На практике TCP-сервер быстрее MQTT до 100 раз и предоставляет минимальные задержки, абсолютную скорость, минимальное потребление трафика и ресурсов CPU.

Итоговая конфигурация IoT устройства

- Транспорт: tcp сокет;
- Протокол обмена: TLV;
- Балансировка: множество A-записей в DNS;
- Язык разработки C;

Пример тестового клиента

<https://github.com/IvanTomshin/otusha/tree/main/diplom/client>

Файл для стрессового тестирования. Позволяет запускать одновременно NUM_CLIENTS потоков, каждый выполняет COUNT соединений.

<https://github.com/IvanTomshin/otusha/blob/main/diplom/client/stress.sh>

```
#!/bin/bash
```

```
COUNT=10000
```

```
NUM_CLIENTS=200
```

```
IP=127.0.0.1
```

```
for ((i=1; i<=NUM_CLIENTS; i++)); do  
    ./tcp_client "$IP" 9000 "$COUNT" &  
    sleep 0.01  
done
```

Пример работы клиента

```
root@gatedev:/home/tomshin/otus# ./stress.sh  
Использование: ./tcp_client <IP> <PORT> <Попыток>  
IP: 46.48.35.178  
PORT: 9000  
Соединений: 1000  
Соединений всего: 1000, успешных: 1000, latency: 0.1890 ms
```



Этап 1.2. Прием, обработка и хранение данных (серверная часть).

1.2.1 TCP сервер приема данных

Референс

- Сервис должен быть доступен всегда для приема данных; Изменение тэгов, форматов шаблонов не должно приводить к перерывам в приеме данных;
- Сервис должен принимать данные с минимальной задержкой;
- Сервис должен иметь минимальное потребление CPU;
- Время последнего сеанса связи контроллера и сервера важный параметр для реагирования, доступный в online;

Реализация

- Язык разработки TCP сервера – C;
- Сервис принимает данные от Контроллера, проверяет контрольную сумму;
- Сервис отправляет Контроллеру ответ — данные приняты;

Пример тестового сервера

https://github.com/IvanTomshin/otusha/blob/main/diplom/server/tcp_server.c

Пример работы серверной утилиты

```
root@ivan-vm-Ubuntu24:/disk2# ./tcp_server
Использование: ./tcp_server -redis
Server listening on port 9000...
SERVER RPS: 00084 |
SERVER RPS: 00005 |
```

1.2.2 Кеширование Redis

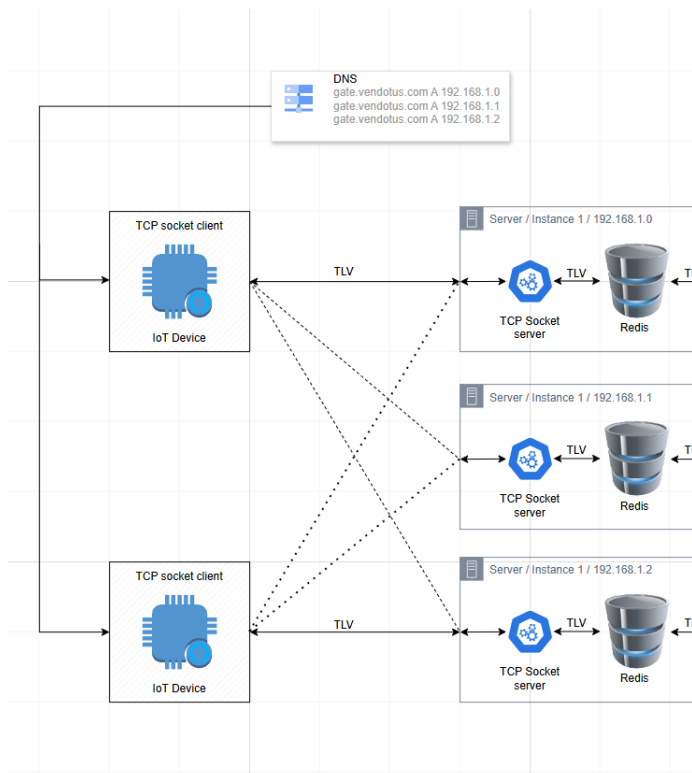
Референс

- Данные поступающие от Контроллеров необходимы для дальнейшего анализа и обработки;
- Данные должны сохраняться в базу данных;
- Парсинг и обработка данных не должна приводить к задержкам приема данных от Контроллера;
- Парсинг, сохранение и обработка данных при приеме напрямую в PostgreSQL будет приводить к задержкам приема данных;

Реализация

- Сырые данные (TLV пакет) от Контроллеров сохраняются в Redis без обработки;

Архитектурная схема

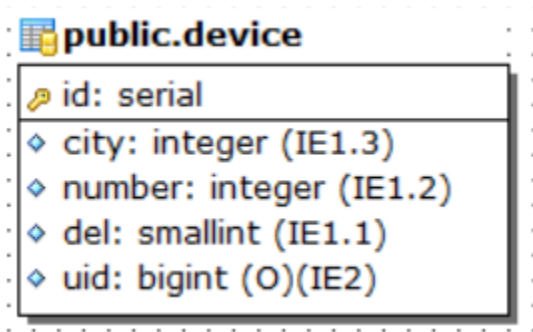


1.2.3 Парсинг данных

Референс

- Поступающие в Redis данные от Контроллеров необходимо разбирать (парсить TLV пакет) и сохранять в БД PostgreSQL;

Структура БД PostgreSQL



public.device	
id	serial
city	integer (IE1.3)
number	integer (IE1.2)
del	smallint (IE1.1)
uid	bigint (O)(IE2)

Таблица **device** содержит данные устройств, по ней сопоставляется код Контроллера UID с публичным номером аппарата и городом. Поле — del — демонтированные аппараты.

Для нужд системы backoffice используем индекс.

```
CREATE INDEX device_idx ON public.device
USING btree (city, number);
```

При изменении данных о аппаратах используем триггер:

```
CREATE TRIGGER device_tr_u_i_d
AFTER INSERT
ON public.device
FOR EACH ROW
EXECUTE PROCEDURE public.f_trigger_device();
```

Смысл триггера — при добавлении новых Контроллеров добавить их порядковый ID во все необходимые таблицы и проинформировать парсер о добавлении новых устройств.

```

CREATE OR REPLACE FUNCTION public.f_trigger_device (
)
RETURNS trigger AS
$body$
BEGIN

    IF    TG_OP = 'INSERT' THEN
        INSERT INTO data_current (device) values (NEW.id);
        INSERT INTO data_current_dt (device) values (NEW.id);
        INSERT INTO data_yesterday (device) values (NEW.id);
        execute format('notify device, "id: %s"', NEW.id);
/* и другие таблицы */
        RETURN NEW;
    ELSE
        RETURN NEW;
    END IF;

END;
$body$
LANGUAGE 'plpgsql'
VOLATILE
CALLED ON NULL INPUT
SECURITY INVOKER
PARALLEL UNSAFE
COST 100;

ALTER FUNCTION public.f_trigger_device ()
OWNER TO psql;

```

Стратегия обработки данных

- Данные вставляются в таблицу data. Таблица шардируется по device и партиционируется по timestamp, так как для нас оперативной работы важны данные за текущие сутки. Данные за предыдущие сутки агрегируются и удаляются (рассматривается в описании шардирования);
- Срабатывает триггер вставки который:
 - Обновляет текущие значения параметров в таблице data_current

- Обновляет дату получения параметра значения в таблице data_current_dt
- Генерируется событие **notify** для обработчиков.
- При наступлении новых суток данные агрегируются в таблице data_day, переносятся текущие значения в значения вчерашнего дня (yesterday) через pg_cron, создаётся новая партиция на (текущие сутки +1) для опережения, удаляются данные партиции за предыдущие сутки.

Хранение данных, шардирование и партиционирование Citus

Таблицы с основными данными

-- Текущие показатели

```
CREATE TABLE data_current (
  device_id bigint NOT NULL,
  p_10 bigint NULL,
  p_11 bigint NULL,
  p_12 bigint NULL,
  p_13 bigint NULL
);
```

-- Время получения текущих показателей

```
CREATE TABLE data_current_dt (
  device_id bigint NOT NULL,
  ts timestampz NOT NULL,
  p_10 timestampz NULL,
  p_11 timestampz NULL,
  p_12 timestampz NULL,
  p_13 timestampz NULL
);
```

-- Вчерашние показатели

```
CREATE TABLE data_yesterday (
  device_id bigint NOT NULL,
  p_10 bigint NULL,
  p_11 bigint NULL,
  p_12 bigint NULL,
  p_13 bigint NULL
);
```

-- Агрегированные данные за предыдущие сутки

```
CREATE TABLE data_day (
  device_id bigint NOT NULL,
  dt date NOT NULL,
  p_11 bigint NULL
);
```

Таблицы с данными, используется решение Citus

-- Основная таблица хранения данных статистики, разбиваемая на шарды и партиции

```
CREATE TABLE data (
  ts timestampz NOT NULL,
  device_id bigint NOT NULL,
  p_10 bigint NULL,
  p_11 bigint NULL,
  p_12 bigint NULL,
  p_13 bigint NULL
) PARTITION BY RANGE (ts);
```

-- создание шардирования таблицы

```
SELECT create_distributed_table('data', 'device_id');
```

В результате:

- device_id → определяет шард (через hash).
- ts → определяет локальную партицию внутри шарда.

В результате появляется двухуровневое распределение:

- Citus (по устройствам, горизонтально по кластерам).
- Postgres (по времени, вертикально внутри каждого шарда).

-- пример создания партиций таблицы на каждый день

```
CREATE TABLE data_2025_09_13
PARTITION OF data
```

```

FOR VALUES FROM ('2025-09-13') TO ('2025-09-14');

CREATE TABLE data_2025_09_14
PARTITION OF data
FOR VALUES FROM ('2025-09-14') TO ('2025-09-15');

CREATE TABLE data_2025_09_16
PARTITION OF data
FOR VALUES FROM ('2025-09-15') TO ('2025-09-16');

CREATE TABLE data_2025_09_17
PARTITION OF data
FOR VALUES FROM ('2025-09-16') TO ('2025-09-17');

-- пример удаления партиции за день.
DROP TABLE data_2025_09_16;

```

Используем функцию `aggregate_and_drop_yesterday_citus()`, для агрегации данных за прошедшие сутки с каждой партиции в таблицу `data_day`, удаления вчерашних партиций и создания новых партиций.

Используя `pg_cron` будем запускать функцию `aggregate_and_drop_yesterday_citus()` ежедневно в 0:00

```
SELECT cron.schedule('0 0 * * *', $$SELECT aggregate_and_drop_yesterday_citus();$$);
```

-- Функция агрегирования и удаления вчерашней партиции в Citus

```

CREATE OR REPLACE FUNCTION public.aggregate_and_drop_yesterday_citus()
RETURNS void
LANGUAGE plpgsql
AS $function$
DECLARE
    yesterday_date text;
    tomorrow_date text;
    yesterday_val date;
    tomorrow_val date;

```

```

sql_agg text;
sql_drop text;
sql_create text;
BEGIN
yesterday_date := to_char(current_date - interval '1 day', 'YYYY_MM_DD');
tomorrow_date := to_char(current_date + interval '1 day', 'YYYY_MM_DD');
yesterday_val := current_date - interval '1 day';
tomorrow_val := current_date + interval '1 day';

-- 1) Агрегируем вчерашнюю партицию
sql_agg := format('$sql$
INSERT INTO data_day (device_id, dt, p_11)
SELECT device_id,
        %L::date AS dt,
        sum(p_11)
FROM data
WHERE ts >= DATE %L AND ts < DATE %L and p_11 is not null
GROUP BY device_id
$sql$', yesterday_val, yesterday_val, current_date);

RAISE NOTICE 'Aggregating: %', sql_agg;
EXECUTE sql_agg;

-- 2) Удаляем вчерашнюю партицию
sql_drop := format('DROP TABLE IF EXISTS data_%s CASCADE;', yesterday_date);
RAISE NOTICE 'Dropping: %', sql_drop;
EXECUTE sql_drop;

-- 3) Создаём завтрашнюю партицию
sql_create := format('$sql$
CREATE TABLE data_%s
PARTITION OF data
FOR VALUES FROM ('%s') TO ('%s');
$sql$',
tomorrow_date,
tomorrow_val::text,
(tomorrow_val + 1)::text
);

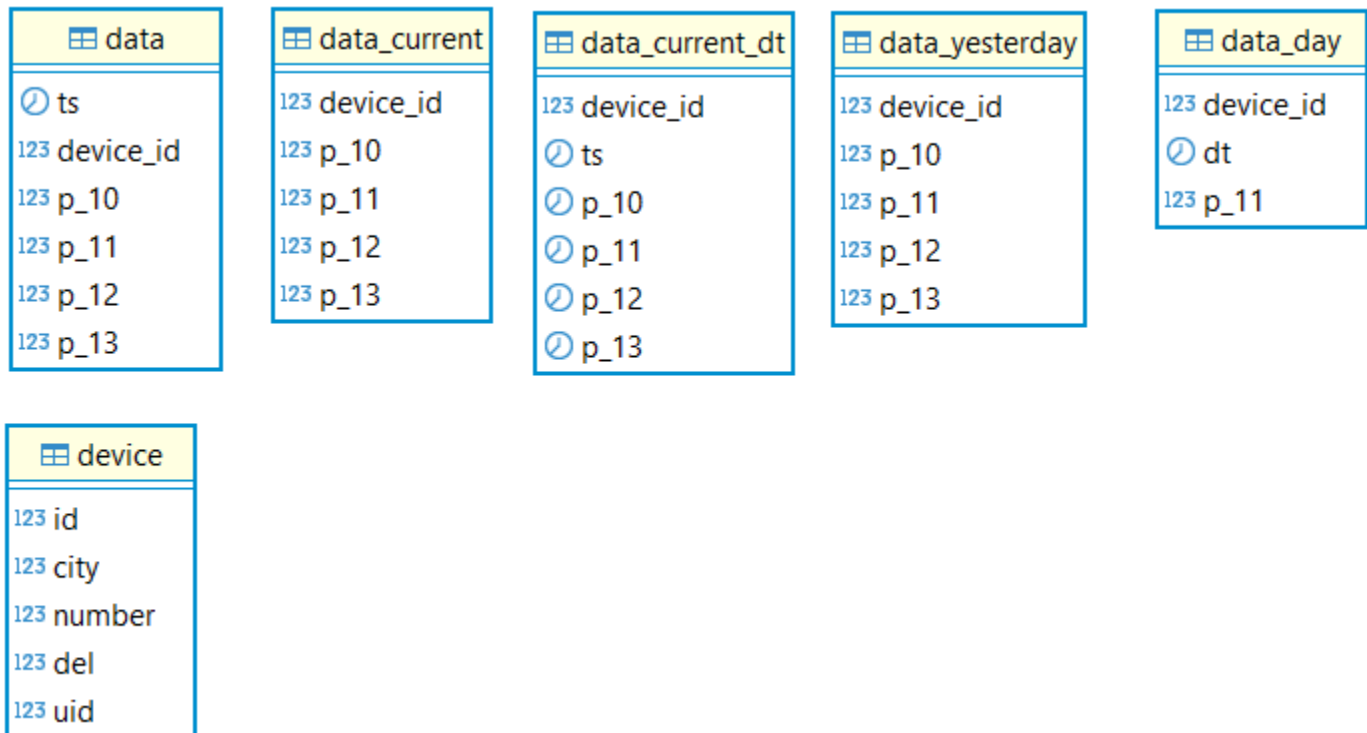
RAISE NOTICE 'Creating: %', sql_create;
EXECUTE sql_create;

```



```
END;
$function$
;
```

Схема базы данных



Реализация

- Серверный обработчик (парсер) на языке C;
- Использование триггеров PostgreSQL для обработки данных;
- Использование механизма уведомлений Notify БД PostgreSQL;
- Использование расширения Citus для шардинга и партиционирования

Пример тестового парсера

https://github.com/IvanTomshin/otusha/blob/main/diplom/server/tcp_parser.c

Пример работы парсера

```
root@ivan-vm-Ubuntu24:/disk2# ./tcp_parser  
|  PARSER RPS: 00271
```

Структура базы данных

<https://github.com/IvanTomshin/otusha/blob/main/diplom/postgresql/db.sql>

Инфраструктура проекта в Docker Compose

<https://github.com/IvanTomshin/otusha/blob/main/diplom/docker-compose.yml>

Этап 1.3. Backoffice и интеграция с внешними системами.

Референс

Современная система управления «backoffice» — это web-интерфейс, работающий на различных платформах, доступный 24/7 и предоставляющий:

- данные для оперативного управления сетью устройств,
- аналитические отчёты,
- информацию для принятия управленческих решений.

Основные параметры системы управления:

- Эффективная работа на мониторах «4K» и мобильных устройствах;
- Групповое управление и работа с десятками тысяч устройств;
- Доступность через web-интерфейс.

Выбор платформы

Ext JS — это зрелый JavaScript-фреймворк, который:

- Содержит сотни визуальных компонентов, работающих как единое целое;
- Позволяет создавать интерфейсы, масштабируемые под мобильные устройства и планшеты;
- Использует архитектуру MVC/MVVM для подключения к нескольким источникам данных и удобного разделения логики;
- Обеспечивает эффективную работу с фильтрами, сортировками и таблицами (гридами) при больших объёмах данных;
- Поддерживает визуализацию данных (графики, диаграммы, KPI-дашборды);
- Позволяет интегрировать внешние продукты и модули через iframe (подходит для микросервисной архитектуры);
- Обеспечивает кроссбраузерную совместимость и единый UI/UX без необходимости ручной доработки стилей;
- Имеет встроенные механизмы локализации и поддержку интернационализации;
- Гарантирует долгосрочную поддержку и обновления (Sencha/Ext JS используется в корпоративных системах десятилетиями);
- Позволяет быстро прототипировать интерфейсы и сокращает время вывода продукта на рынок.

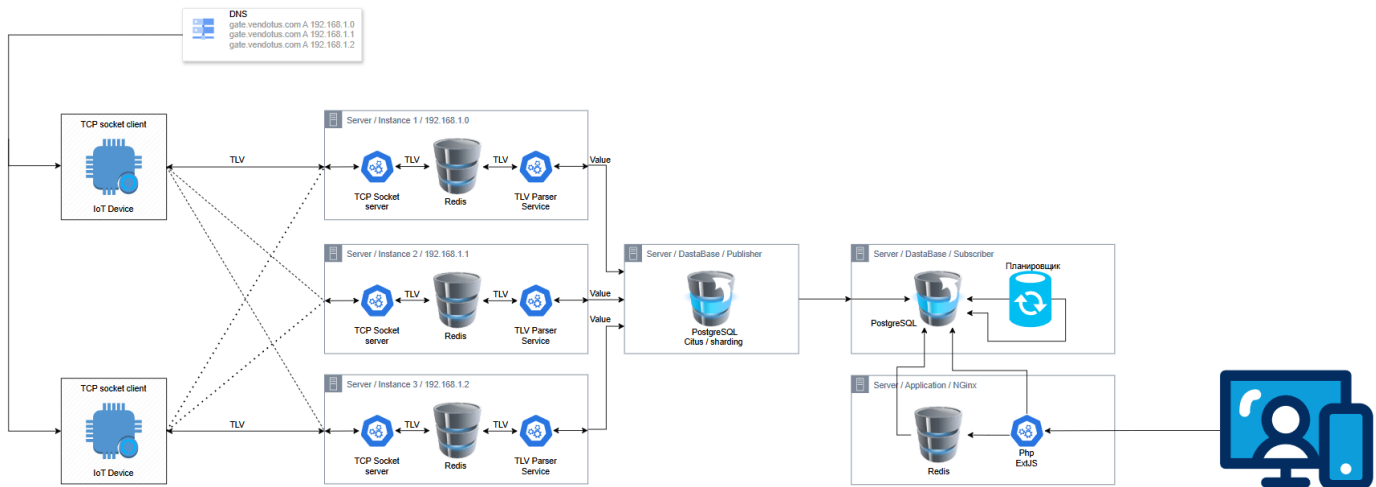
Работа с данными

- Данные для работы «backoffice» передаются от публикатора PostgreSQL на подписчика.
- Ext JS запрашивает данные с подписчика PostgreSQL и при выполнении ресурсоемких расчетов кеширует данные (дашборды) в Redis
- Для аналитической обработки дополнительно возможно использование Microsoft Power BI, а так же передавать данные в OLAP системы;

<https://www.sencha.com/>



Итоговый вариант архитектуры проекта



Инструменты и используемые технологии

- 2 Docker — платформа контейнеризации.
- 3 Ext JS — JavaScript-фреймворк для веб-приложений.
- 4 PostgreSQL — реляционная СУБД.
- 5 Citus — расширение PostgreSQL для шардинга и партиционирования.
- 6 Redis — высокопроизводительное хранилище данных в памяти (кэш).
- 7 C/C++ — языки разработки системных компонентов.
- 8 PHP — язык для разработки API back-office.

Выводы

1. Реализована высоконагруженная архитектура для управления сетью вендинговых устройств, обеспечивающая надёжный приём телеметрии и команд по протоколу TCP с минимальными задержками.
2. Разработан протокол обмена сообщениями (TLV), позволяющий унифицировать передачу данных между устройствами и сервером, упростить парсинг и повысить надёжность обработки.
3. Внедрено кэширование горячих данных в Redis, что существенно снизило нагрузку на основную СУБД и повысило отклик системы при обработке запросов.
4. Использована PostgreSQL с расширением Citus, что позволило реализовать горизонтальное масштабирование (шардирование) и обеспечить отказоустойчивое хранение телеметрии.

OTUS – Highload Architect

5. Разработаны методы агрегации данных для аналитической обработки, что обеспечивает операторов актуальной и наглядной информацией о работе устройств, продажах и инцидентах.
6. Сформирована архитектура back-office с возможностью интеграции во внешние системы, предоставляющая операторам удобный веб-интерфейс для мониторинга и управления.
7. Проведено нагрузочное профилирование, результаты которого подтвердили эффективность предложенных решений: система сохраняет стабильность при росте числа подключённых устройств.
8. Отличительной особенностью проекта стало использование простых и доступных технологий, минимизирующих затраты на администрирование и обеспечивающих гибкость при дальнейшем развитии.

Репозиторий студента

<https://github.com/IvanTomshin/otusha/tree/main/diplom>

