Составные части системы и их описание.

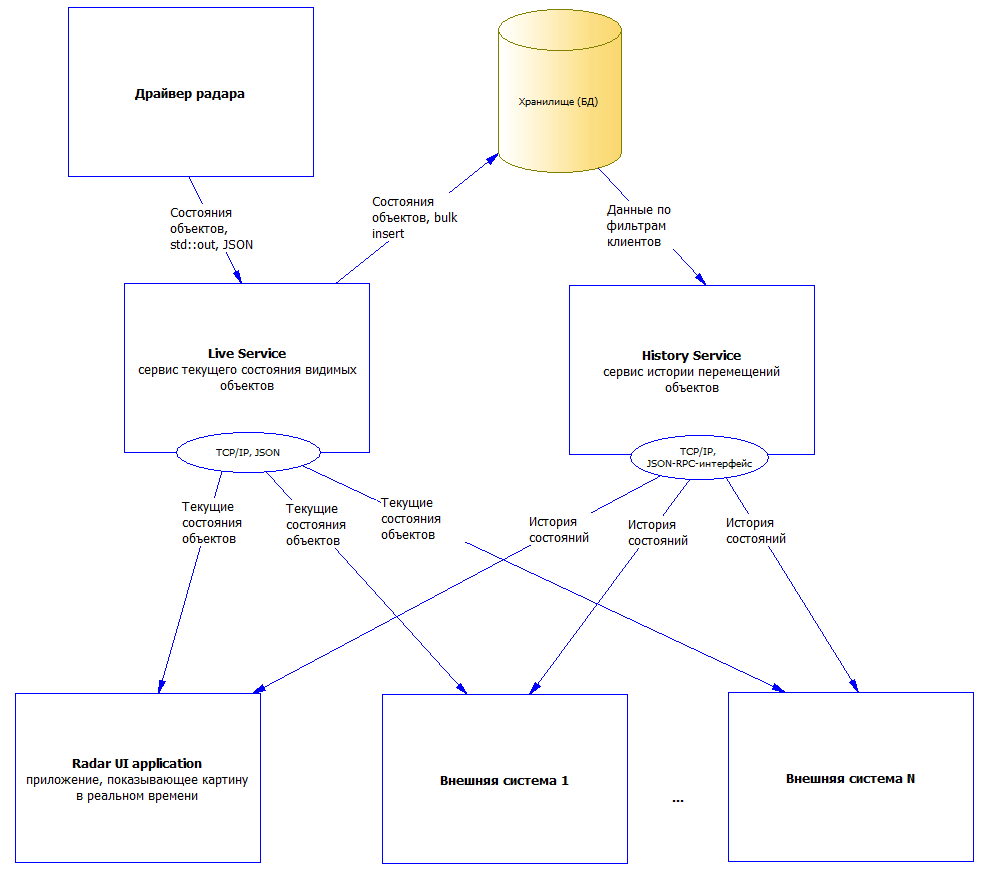


Рис. 1. Диаграмма компонентов и потоков данных в вольном виде.

1. ***Драйвер радара*** – описано в тексте задания.
2. ***Live Service*** – сервис текущего состояния наблюдаемых объектов (ТСНО). У него несколько функций:
   * Получает ТСНО от драйвера радара;
   * Транслирует ТСНО по TCP/IP всем подключенным клиентам (TCP-JSON streaming). Без запроса со стороны клиентов, как радиоэфир.
   * Сохраняет ТСНО, периодически посылая SQL-запросы в Хранилище.
3. ***Хранилище*** – база данных с историей перемещения объектов (ИПО).
4. ***History Service*** – сервис ИПО. По запросам клиентов предоставляет данные из Хранилища по TCP/IP по протоколу JSON-RPC (для простоты). Очевидно, что внешние системы не должны иметь прямой доступ к Хранилищу, поэтому делаем данный сервис.
5. ***Radar UI application*** – приложение, тоже входящее в нашу систему, отображающее на экране ТСНО и ИПО.
6. ***Внешние системы*** – внешние акторы, которые получают от нас ТСНО и ИПО и распоряжаются этой информацией по собственному усмотрению.

Live Service

Данный компонент системы реализован в коде, поэтому приведу лишь диаграмму классов (в слегка упрощённом виде).

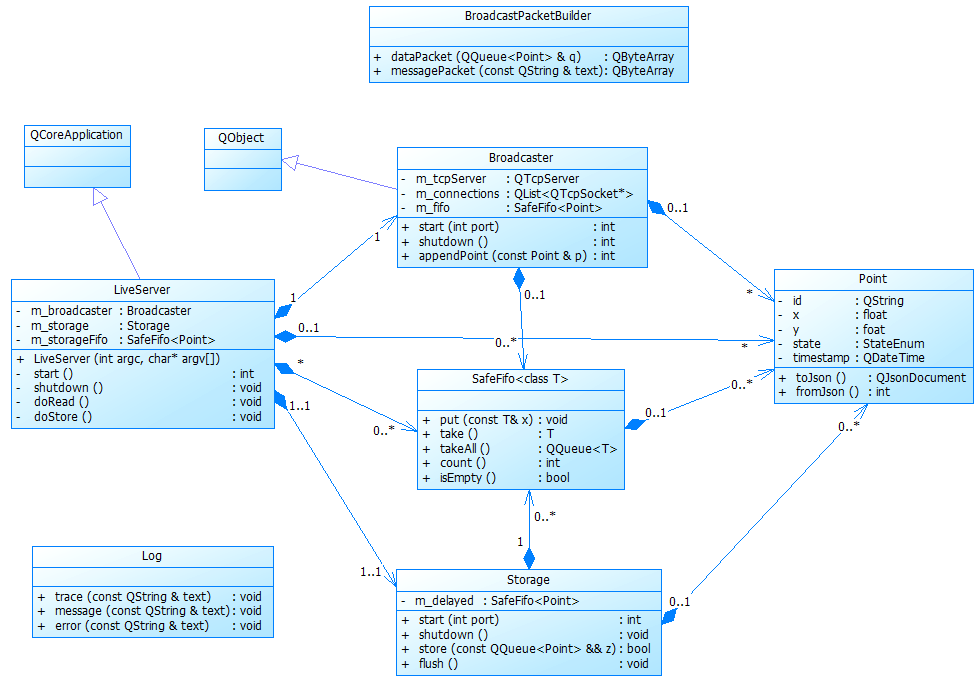


Рис.2 Диаграмма классов модуля Live Service.

Хранилище

Предлагаемая схема данных для БД приведена ниже. Таблица *objects* – наблюдаемые объекты, *points* – их состояния.

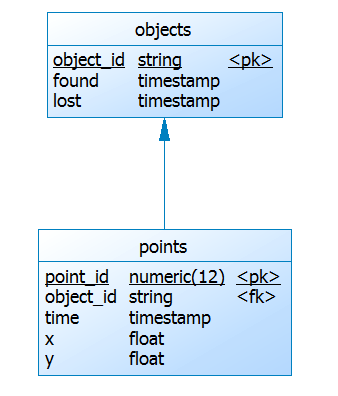


Рис. 3 Схема данных Хранилища.

History Service

Это сервис, обслуживающий клиентов по простому протоколу - JSON-RPC. На входе принимает фильтр, по которому надо возвратить данные. Структура JSON примерно такая:

{

"id":"abcd123",

"state":"lost",

"tsFrom":"dd.mm.yyy hh:mm:ss.zzz",

"tsTo":"dd.mm.yyy hh:mm:ss.zzz",

"rect":{"x1":0,"y1":0, "x2":0.5,"y2":0.7},

"pageNo": 1,

"pageSize": 50,

}

Все поля опциональны, поэтому надо подстраховаться, чтобы клиент не смог запросить миллиард записей и подвесить сервер. Варианты возможны различные:

* Можно сделать пейджинг (для этого и введены параметры pageNo и pageSize – номер и размер страницы соответственно).
* Сделать логические ограничения по параметрам.
* Просто ограничивать макс. кол-во выдаваемых записей в select’е.

Диаграмма классов проста и приведена ниже.

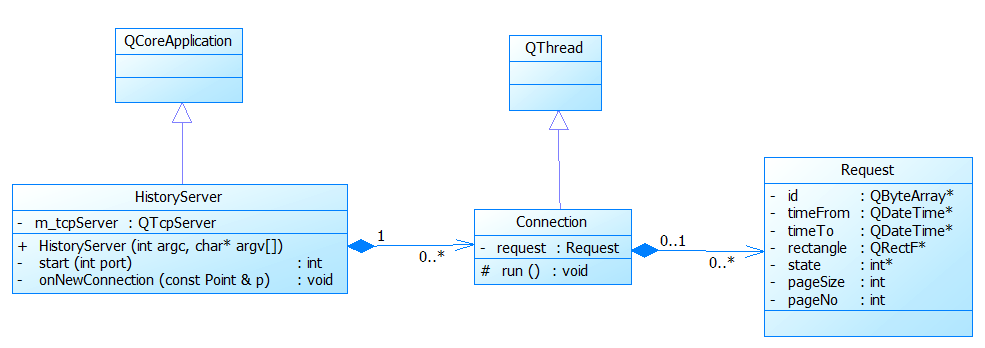


Рис.4 Диаграмма классов History Service.

Предполагается, т.к. клиентов всего несколько, tcp-соединения закрывать только по запросу клиентов. Т.е. после отправки ответа на запрос мы ждём следующего запроса и т.д. Данный цикл изображен на блок-схеме. Выходим из цикла лишь при разрыве соединения или при закрытии приложения History Service.

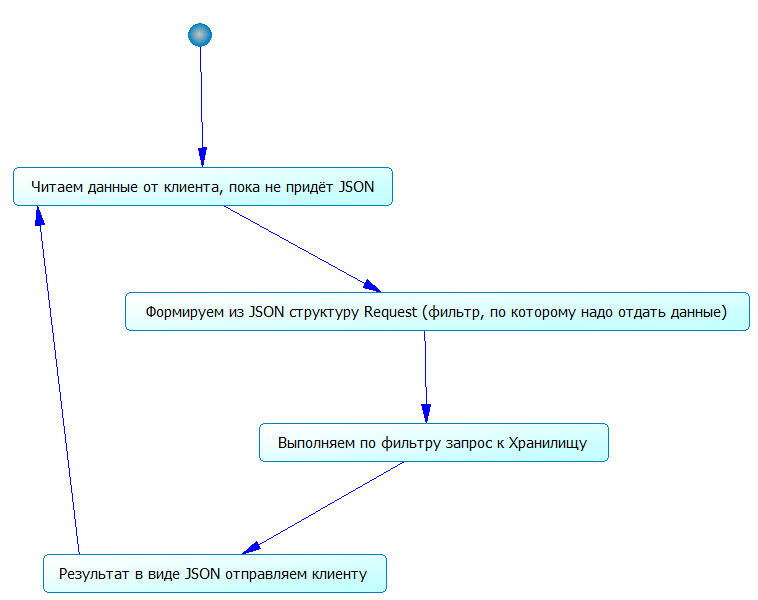


Рис.5 Упрощенная блок-схема метода Cоnnection::run().

Radar UI application

Вообще говоря, чтобы спроектировать GUI, надо знать контекст её использования. За его неимением предлагаю простой вариант (что первое приходит в голову). Опишу словами. Это одно окно (если речь о десктопном приложении) или веб-страница, где содержатся:

* *График* - прямоугольник с нашей координатной системой, где изображаются объекты (возможно, с фазовыми траекториями).
* Под графиком *полоса прокрутки* (как в медиа-плеере, с кнопками «Play/Pause», «Настройка» и т.д. + отображением времени).
* В свободном месте отображаем фильтр. В нём можно выбрать режим: это или «Трансляция» (тогда можно выбрать длину периода до текущего момента) или «Архивные данные» (тогда более расширенный фильтр).

В общем, интерфейс похож немного на Youtube-овский. По ситуации запрашиваем данные с нужного сервиса (или с обоих сразу). Диаграмма классов и алгоритмы вполне тривиальны для Qt-шного UI-приложения, поэтому не привожу здесь.

**Примечания.**

1. Для минимизации трафика вместо JSON в протоколах обмена с серверами можно использовать более компактные формы (MessagePack, например).