Правительство Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Санкт-Петербургский государственный университет»

Кафедра системного программирования

Павлов Владислав Александрович

Организация сбора данных в сети датчиков с нестационарным агрегатором

Бакалаврская работа

Допущена к защите.
Зав. кафедрой:
д. ф.-м. н., профессор Терехов А. Н.

Научный руководитель:
ст. преп. Сартасов С. Ю.

Рецензент:
ООО "Оракл Девелопмент СПБ", Старший руководитель группы Косенчук А. М.

Санкт-Петербург
2016

SAINT-PETERSBURG STATE UNIVERSITY

Software Engineering Chair

Vladislav Pavlov

Organization of data collection in a sensor net with a mobile collector

Bachelor's Thesis

Admitted for defence.
Head of the chair:
professor Andrey Terekhov

Scientific supervisor:
assistant Stanislav Sartasov

Reviewer:
Senior Engineering Manager, Oracle Development SPB, LLC Alexey Kosenchuk

Saint-Petersburg
2016

Оглавление

Введение **4**

1. Постановка задачи **6**

2. Обзор существующих аналогов **8**

2.1. ODK Sensors 8

2.2. ThingSpeak 9

2.3. Axibase Collector 10

2.4. SDCF 11

2.5. Выводы 12

3. Разработка агрегаторского компонента и прототипирование **13**

3.1. Разработка архитектуры 13

3.1.1. Конфигурирование 14

3.1.2. Сбор данных с датчиков 16

3.1.3. Логирование и обработка событий 19

3.1.4. Локальное сохранение данных 21

3.1.5. Выгрузка данных 22

3.2. Выбор программной платформы для реализации агрегаторского компонента фреймворка 24

3.3. Создание прототипа 26

4. Заключение **27**

Список литературы **28**

Введение

В настоящее время «Интернет Вещей» (далее ИВ) стал достаточно распространенной концепцией. Её основная идея заключается в том, что глобальная сеть Интернет стала уже не просто глобальной сетью для общения людей посредством машин, но и средой для устройств, позволяющей им коммуницировать с окружающим миром и между собой. Умные мусорные корзины, которые оповещают санитарные экипажи о своём заполнении, интеллектуальные системы охраны дома и многое другое - всё это стало возможным после появления концепции ИВ.

Сегодня мы наблюдаем огромные объемы данных, которые накапливаются на устройствах, как на сенсорах. Эти данные могут эффективно использоваться. К примеру, датчики, установленные на производстве, могут быть использованы для сбора информации об эксплуатации ресурсов. Полученная с устройств информация может быть полезна для сбора статистики, составления предсказаний, предупреждения опасностей и других целей. Как видно, задачи сбора и анализа данных с устройств являются одними из наиболее приоритетных. Успешное агрегирование и последующая интеллектуальная обработка данных могут позволить построить умную окружающую среду для людей средствами ИВ.

В настоящее время во многих областях ИВ стала актуальна проблема регулярного сбора данных с некоторой периодичностью с датчиков, распределённых по обширной территории. В частности, получил популярность особый вид сбора данных[1], при котором устройство, которое собирает данные (далее агрегатор) перемещается в пространстве, а устройства, получающие данные из внешней среды и копящие их (далее датчики) неподвижны. При таком подходе к сбору данных датчики могут располагаться в местах, где отсутствует выход в глобальню сеть. Из этого можно сделать вывод, что рассматриваемый вид сбора более неприхотлив к окружающей среде, чем, к примеру, подход, при котором имеется ряд стационарных агрегаторов-шлюзов, которые имеют выход в глобальную сеть и периодически собирают данные с датчи-

ков в своей зоне видимости. Также использование данного вида сбора позволяет снизить потенциальную стоимость развертывания решения, так как при таком виде сбора снижаются требования на аппаратные составляющие системы. Ведь теперь в качестве нестационарного сборщика данных может выступать недорогой смартфон, нетбук и любое другое относительно маломощное устройство.

Множество задач, решаемых с помощью использования вида сбора нестационарным агрегатором, обширно. К примеру, сбор нестационарным агрегатором возможен в следующих сценариях:

- Устройство на автомобиле собирает информацию о потенциальных дорожных опасностях с датчиков, установленных вдоль автомагистрали
- Устройство сельского работника (к примеру, квадрокоптер) при попадании в зону видимости миниатюрных датчиков, разбросанных по засеянной зоне, получает оповещения о различных свойствах почвы

Изобилие сфер применения обсуждаемого вида сбора ставит разработчиков систем сбора данных перед одной из следующих проблем:

- Проблема поиска программных средств для быстрого построения систем периодического сбора данных нестационарным агрегатором
- Проблема трудоемкой адаптации существующих инструментов к подходу с нестационарным агрегатором

Поэтому создание некоторого фреймворка для организации сетей с нестационарным сборщиком смогло бы значительно облегчить работу разработчиков и администраторов таких сетей при решении вышеупомянутых проблем.

1. Постановка задачи

Основной целью данной работы является разработка агрегаторской части фреймворка, предназначенного для организации систем периодического сбора данных в сетях датчиков с помощью нестационарного агрегатора. Агрегаторский компонент фреймворка должен обладать следующими свойствами:

- Гибкая архитектура: агрегаторское приложение строится по частям, отдельные части приложения могут быть быстро заменены на новые, агрегационный процесс конфигурируем
- Мультиплатформенность и невысокие аппаратные требования: агрегаторский компонент должен быть спроектирован с учетом возможности разработки систем под большой сегмент устройств, который включает в себя смартфоны и устройства сравнимые с ними по мощности
- Независимость от конкретных протоколов общения между компонентами системы сбора
- Независимость от формата передаваемых данных между компонентами системы сбора
- Относительная простота использования агрегаторского компонента фреймворка для разработчиков

Для достижения цели были поставлены следующие задачи:

- Обзор феймворков аналогичных целевому
- Выбор программной платформы в качестве основы для разработки агрегаторской части фремворка
- Разработка архитектуры агрегаторской части фреймворка, позволяющей разработчикам легко осуществлять создание различно параметризованных нестационарных агрегаторов

- Создание агрегаторского прототипа на основе разработанного фреймворка

Примечание: В данной работе не рассматривалась часть разрабатываемого фреймворка, позволяющая организовывать накопление, обработку и отправку данных на стороне датчиков. Разработка части фреймворка для датчиков велась в параллельной бакалаврской дипломной работе Рагимова Р. В.

2. Обзор существующих аналогов

В сфере Интернета Вещей существует огромное число платформ для организации различных умных систем. В настоящей работе не производился обзор всех возможных аналогов. Рассматривались лишь те существующие системы, которые решали проблему организации сбора данных и позволяли строить различно параметризованные решения.

2.1. ODK Sensors

ODK Sensors[9] является ближайшим аналогом разрабатываемого фреймворка. Данный фреймворк был создан для облегчения работы разработчиков при написании приложений под платформу Android для сбора данных с внутренних или внешних датчиков. Фреймворк не имеет разделения на компоненты для агрегатора и датиков, и вся настройка системы ведётся на стороне сборщика. Как утверждают разработчики фреймворка, некоторые типичные датчики поддерживают только такие низкоуровневые I/O интерфейсы, как I^2C, SPI и другие, что усложняет внешнее конфигурирование таких датчиков. Фреймворк позволяет строить приложения для сбора данных, которые могут варьироваться в следующих направлениях: протокол общения с датчиками, формат собираемых данных, конфигурация датчиков. Фреймворк предоставляет отдельные интерфейсы для разработчиков приложений, разработчиков драйверов датчиков и разработчиков самого фреймворка. Среди основных достоинств фреймворка при решении задачи организации сбора данных нестационарным агрегатором можно выделить:

- Фреймворк допускает отсутствие выхода в Интернет у датчиков
- Гибкая архитектура фреймворка

Среди недостатков фреймворка при решении поставленной задачи можно отметить:

- Фреймворк не предоставляет встроенной поддержки выгрузки данных на отдалённый сервер

- Централизованная конфигурация на стороне сборщика увеличивает трудоемкость настройки некоторых типичных датчиков
- Фреймворк пригоден для разработки только под платформу Android

2.2. ThingSpeak

ThingSpeak[12] является бесплатным онлайн сервисом, позволяющим организовывать сбор данных с датчиков с последующим сохранением в облаке. Данный сервис предоставляет некоторый API для быстрого построения собственного приложения. Данный сервис имеет функционал, позволяющий производить анализ собранных данных, строить их различные визуальные представления с помощью MATLAB, создавать событийные триггеры и другое. ThingSpeak использует концепцию каналов для организации сбора информации с датчиков. Каналы предоставляют интерфейс для чтения и записи и сохраняют всю информацию, которую в них записали устройства. Для того, чтобы начать работу в системе ThingSpeak достаточно зарегистрироваться в системе, создать канал, настроить датчик на отсылку данных в созданный канал и при необходимости настроить визуальное отображение полученных данных. Как только канал зарегистрирован в ThingSpeak, он способен принимать, обрабатывать данные и предоставлять к ним доступ внешним приложениями. Каналы поддерживают манипуляции с данными в форматах JSON, XML и CSV. ThingSpeak использует HTTP протокол для приема данных с датчиков. Для настройки собственного сервера сбора данных требуется Ruby on Rails.
Таким образом, среди достоинств данной системы для решения поставленных задач можно отметить:

- Простота использования
- Предоставление множества настроек сбора и последующей обработки данных

Среди недостатков системы при её использовании для решения поставленной задачи можно отметить:

- Общение в данной системе основано на фиксированном протоколе и не обеспечивает требуемой протоколонезависимости
- Система нацелена на сбор данных в реальном времени
- Трудоёмкий запуск сервера на маломощном смартфоне ввиду системных требований Ruby On Rails

2.3. Axibase Collector

Axibase Collector[2] представляет собой отдельное Java приложение, которое позволяет организовывать сбор информации различной природы с различных внешних источников данных. После сбора данных данные отправляются в Axibase Time Series Database[3]. Различные агрегаторские процессы могут быть заданы с помощью создания специальной Job с определённым типом, характеризующим формат собираемых данных и протокол общения с источниками данных. Возможно задание расписания Job с помощью cron выражений. Axibase Collector требует Java версии 1.7 или выше. Продукт запускается на устройствах с как минимум 1 процессором с тактовой частотой от 2 GHz, оперативной памятью от 1 GB и с такими операционными системами как Ubuntu 14+, RedHat Enterprise Linux 6+, Suse Linux Enterprise 11, Debian 6+, CentOS 6+. Таким образом, среди достоинств данного продукта для решения поставленной задачи стоит отметить:

- Предоставление возможности задания расписания сбора данных
- Подменяемость формата собираемых данных и протокола общения с источниками

Среди недостатков продукта при решении поставленной задачи выделяются:

- Высокие системные требованния: Axibase Collector не сможет запуститься на маломощном смартфоне

- Отстутствие возможности задания обработки данных на самом агрегаторе

- Фиксированное конечное хранилище данных

2.4. SDCF

SDCF[8] – фреймворк для сбора данных с датчиков Android устройства с открытым исходным кодом. Данный фреймворк предоставляет возможность гибкой конфигурирации процесса сбора данных с датчиков разного типа на Android устройствах. Собранные данные сохраняются в локальной базе данных на устройстве. Фреймворк позволяет организовывать периодическую отправку данных на отдаленный сервер в формате XML. Также данный фреймворк может рассылать собираемые данные другим приложениям на Android устройстве. Проект имеет свой портал с доступной Javadoc документацией. Таким образом, среди достоинств данного продукта для решения поставленной задачи стоит отметить:

- Предоставление возможности организации периодического сбора данных с помощью Android устройства

- Гибкая конфигурируемость процесса сбора данных

- Возможность работы с датчиками различного типа

Среди недостатков продукта при решении поставленной задачи выделяются:

- Фиксированная платформа, под которую решения могут быть созданы с помощью фреймворка

- Фиксированный формат отсылаемых данных

- В архитектуре SDCF устройства датчиков и устройство агрегатора совпадают

2.5. Выводы

Существующие инструменты для организации сбора данных недостаточно адаптированы для организации сбора данных с помощью нестационарного сборщика. Приведённые примеры систем иллюстрируют некоторые из наиболее часто встречаемых разработчиками сложностей при попытке организовать систему сбора данных нестационарным агрегатором под свой конкретный случай с помощью существующих продуктов: зависимость от конкретных протоколов общения агрегатора с датчиками, неподходящая архитектура, зависимость от формата передачи данных, недостаточная мультиплатформенность, невозможность запустить систему на «слабых» устройствах и другие. Ввиду вышеперечисленного было решено разработать собственную архитектуру агрегаторской части фреймворка, которая смогла бы сочетать в себе плюсы и избегать минусы вышеперечисленных систем сбора. Целевой фреймворк должен предоставлять инструментарий, ориентированный на создание гибко конфигурируемых мультиплатформенных систем периодического сбора данных с датчиков нестационарным агрегатором. Разработанный фреймворк также мог бы быть полезным при адаптации подхода с использованием нестационарного агрегатора к существующим системам сбора.

3. Разработка агрегаторского компонента и прототипирование

3.1. Разработка архитектуры

При создании агрегаторского компонента фреймворка для организации систем сбора данных в сетях датчиков нестационарным агрегатором предполагалось, что:

- В целевой системе сбора допускается использование нескольких независимых сборщиков
- В целевой системе сбора допускается использование отдалённого сервера для надёжного сохранения информации, собранной агрегаторами
- В целевой системе сбора агрегаторы могут заранее не знать о существовании датчиков, готовых к передаче данных
- Компонент не решает вопросы безопасности передачи данных

Перед разработкой архитектуры агрегаторской части фреймворка требовалось задать требования на архитектуру. Эти требования к компоненту были сформулированы в терминах основных возможностей, которые он должен предоставлять разработчику:

- Возможность конфигурирования стратегии сбора: возможен одновременный запуск нескольких различно параметризованных процессов сбора (сборы по разным протоколам, сборы разной периодичности с разных датчиков и тому подобное)
- Возможность конфигурирования пропускной способности сбора: количество одновременно обрабатываемых датчиков может быть задано разработчиком
- Возможность задания поведения при определённых событиях сбора и добавление собственных: система должна выполнять особые действия, реагируя на различного рода события (например,

каким-либо образом сигнализировать при обнаружении нового датчика)

- Возможность организации и настройки локального долгосрочного сохранения данных на устройстве сборщика

- Возможность быстро подменять тип данных собираемых с датчиков: система не должна быть зависима от данных, которые она собирает с датчиков

- Возможность настройки отправки данных на отдалённый сервер: отправка данных на сервер может производиться по разным протоколам, в разном формате, иметь разное расписание

- Возможность настройки логирования: логирование является важной частью любой системы, поэтому администратор системы должен иметь контроль над политикой логирования

В работе агрегаторского компонента целевого фреймворка было выделено несколько ключевых процессов: конфигурирование, сбор данных с датчиков, логирование и обработка событий, локальное сохранение данных, выгрузка данных на отдалённый сервер. Архитектура компонента была построена таким образом, что каждый процесс протекал максимально независимо от других. Для каждого процесса в архитектуре компонента можно выделить подкомпонент, отвечающий за обеспечение протекания этого процесса.

Далее будет кратко описан каждый подкомпонент. Описание каждого подкомпонента будет снабжаться графическим представлением его архитектуры.

3.1.1. Конфигурирование

Ввиду того факта, что агрегаторский компонент нацелен на организацию различно параметризованных систем сбора данных, подкомопнента конфигурирования системы играет одну из наиболее важных ролей и нуждается в описании в первую очередь. Именно этот подкомпо-

нент предоставляет некоторый интерфейс для разработчика системы и позволяет последнему создавать абсолютно непохожие друг на друга системы, быстро подменяя нужные элементы в ней.
Замечание: далее на всех схемах элементы, которые могут быть легко заменены разработчиком на другие за счёт подкомпонента конфигурирования, будут помечены зелёным цветом.

Общая структура подкомпонента конфигурирования представлена на рисунке 1.

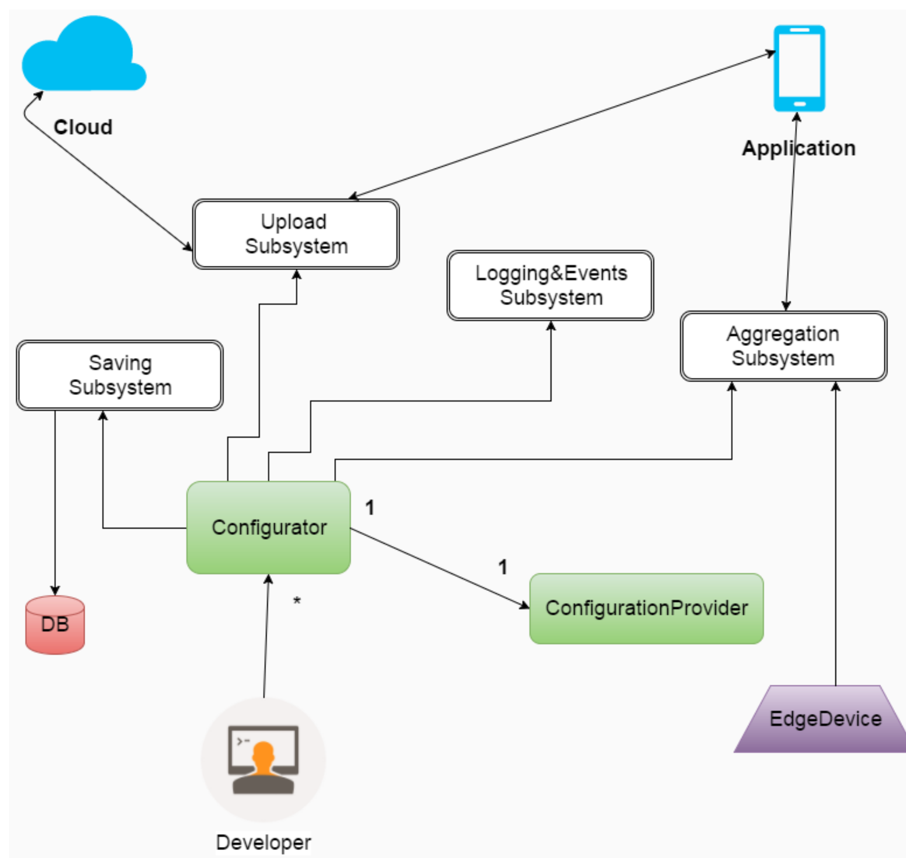

Рис. 1: Архитектура подкомпонента, отвечающего за конфигурирование всей системы

Процедура конфигурирования будущей системы была максимально облегчена для разработчика. Подкомпонент был спроектирован таким образом, что создание экземпляров большинства классов, которые параметризуют систему сбора, контролируется самой системой, а не разработчиком. Разработчик всего лишь создает некоторый объект, реализующий интерфейс $Configurator$ (далее конфигуратор), который будет

строить элементы будущей системы. После создания первоначального класса своего конфигуратора разработчик может быстро подменять построение необходимых элементов, переопределяя методы, их создающие. Если разработчик не хочет создавать своего конфигуратора, определяя все методы интерфейса $Configurator$, он может воспользоваться стандартной реализацией $DefaultPlsosConfigurator$, наследуясь от этого класса и переопределяя нужные методы. О реализациях по умолчанию более детально речь пойдёт при описании других подкомпонент. Конфигуратор будет также предоставлять системе объект, реализующий интерфейс $ConfigurationProvider$ (далее предоставитель) и содержащий все необходимые элементарные настройки для этих построений. Далее элементы могут строиться с помощью конфигуратора с использованием глобальных настроек предоставителя. Если в какой-то момент времени следует поменять глобальные настройки объект предоставителя может быть подменен в конфигураторе. Если такие изменения слишком глобальны для системы, разработчик мождет создать специальный пакет настроек для нужного элемента и построить его. Стоит также заметить, что архитектура подкомпонента довольно гибка, и разработчик может создавать нескольких конфигураторов со своими предоставителями, если того требует система. В архитектуре описываемого подкомпонента использовался шаблон проектирования AbstractFactory[4].

3.1.2. Сбор данных с датчиков

Подкомпонент сбора данных является одним из самых важных подсистем агрегаторского компонента. Ввиду этого факта данный подкомпонент будет описан детально. Разработанная архитектура подкомпонента представлена на рисунке 2.

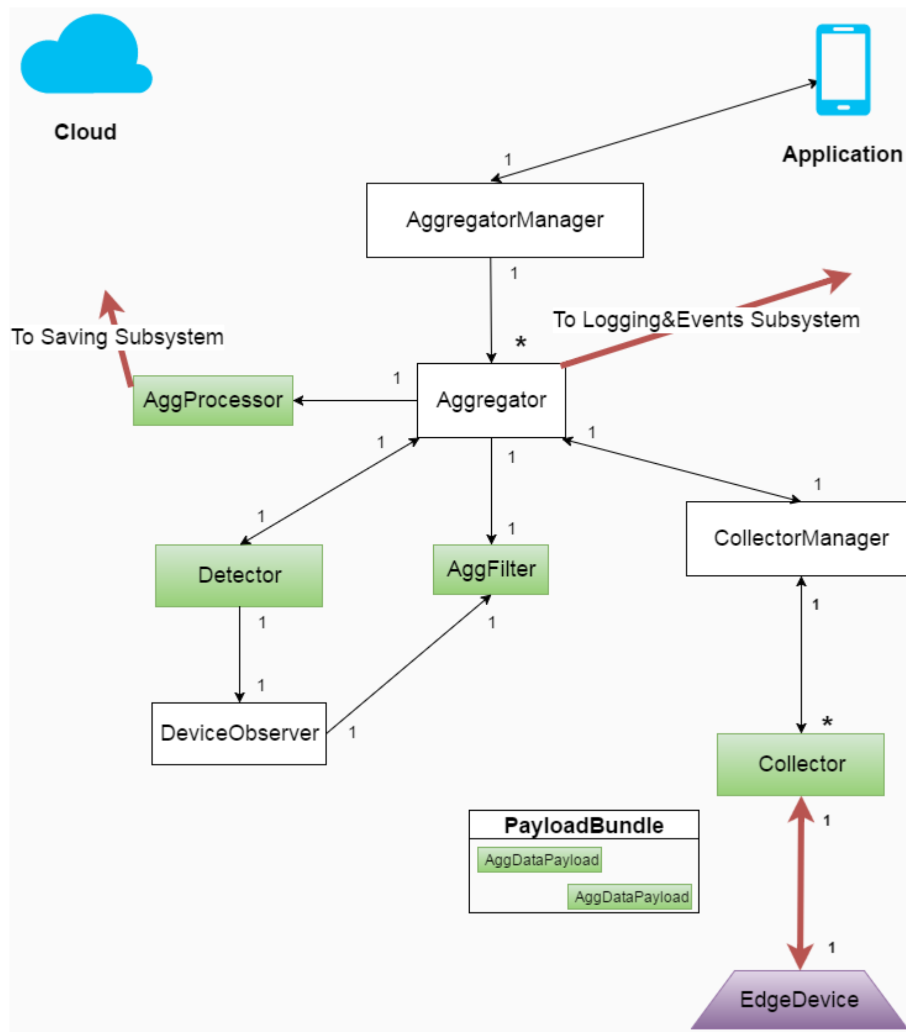

Рис. 2: Архитектура подкомпонента, отвечающего за сбор данных с датчиков

Этот подкомпонент ответственен за обнаружение датчиков, обеспечение сбора данных, временное хранение собранных данных, а также их фильтрацию и обработку. Подкомпонент управляется непосредственно самим приложением, предоставляя последнему некоторый интерфейс через Singleton[4] класс *AggregatorManager*. Данный класс представляет собой некоторый контейнер объектов класса *Aggregator* и управляет их созданием, запуском, остановкой, модификацией. Таким образом, один агрегационный процесс ассоциируется с объектом класса *Aggregator*, который может быть по-разному параметризован. Одними из важнейших параметров являются конфигурации объектов *Collector*, *Detector*, *AggFilter*. Более детально эти классы и их реализации по

умолчанию описаны ниже.

- *Collector*

  Абстрактный класс, отвечающий за взаимодействие с датчиком. Инкапсулирует всю логику отправки и принятия информации с датчиков. В качестве объекта класса *Collector* (далее коллектор) по умолчанию любой конфигуратор будет возвращать объект класса *DefaultDebugCollector*, "собирающий" произвольное число искусственно генерируемых пакетов. Один коллектор ведет общение с одним датчиком. Несколько коллекторов могут быть запущены одновременно для параллельной обработки нескольких датчиков (число регулируется настройкой *threadCount*). Жизненный цикл нескольких коллекторов контролируется объектом класса *CollectorManager*. Реализуя класс *Collector*, разработчик указывает наибольшую часть протоколозависимого поведения его системы. Реализация этого класса сводится к определению нескольких методов: установка соедине ния, идентификация сторон, принятие некоторой "порции" данных, отсылка подтверждения. Именно такая последовательность действий формирует всё общение сборщика с датчиком. Формат данных легко заменяем. Все данные, полученные из внешнего мира датчиком, обёрнуты на стороне агрегатора в объекты класса *AggDataPaylod* (далее объекты-данные). Реализация класса *AggDataPaylod* вводит новый тип данных, собираемый системой. Сами объекты-данные передаются "порциями" (по несколько штук объектом класса *PayloadBundle*) для уменьшения объема передаваемой служебной информации.

- *Detector*

  Абстрактный класс, ответственный за обнаружение датчиков. Поддерживает цикличный опрос (настройка *detectorRefresh*) сети на наличие устройств, а также асинхронное обнаружение. В качестве объекта класса *Detector* (далее детектор) по умолчанию любой конфигуратор будет возвращать объект класса *DefaultDebugDetector*, произвольным образом искусственно "обнаруживающего" одно из

50000 отладочных устройств. Реализация *Detector* является последней протоколозависимой реализацией элемента системы после определения классов коллекторов.

- *AggFilter*
  Интерфейс, который используется для фильтрации обнаруживаемых устройств и входящих с них данных. По умолчанию любой конфигуратор будет возвращать реализацию *AggFilter*, которая пропускает все устройства и данные.

3.1.3. Логирование и обработка событий

Рассматриваемый подкомпонент ответственен за логирование сообщений агрегаторского компонента, а также за обработку событий, которые генерируются объектами класса *Aggregator* (далее объекты-агрегаторы). Разработанная архитектура подкомпонента схематично представлена на рисунке 3.

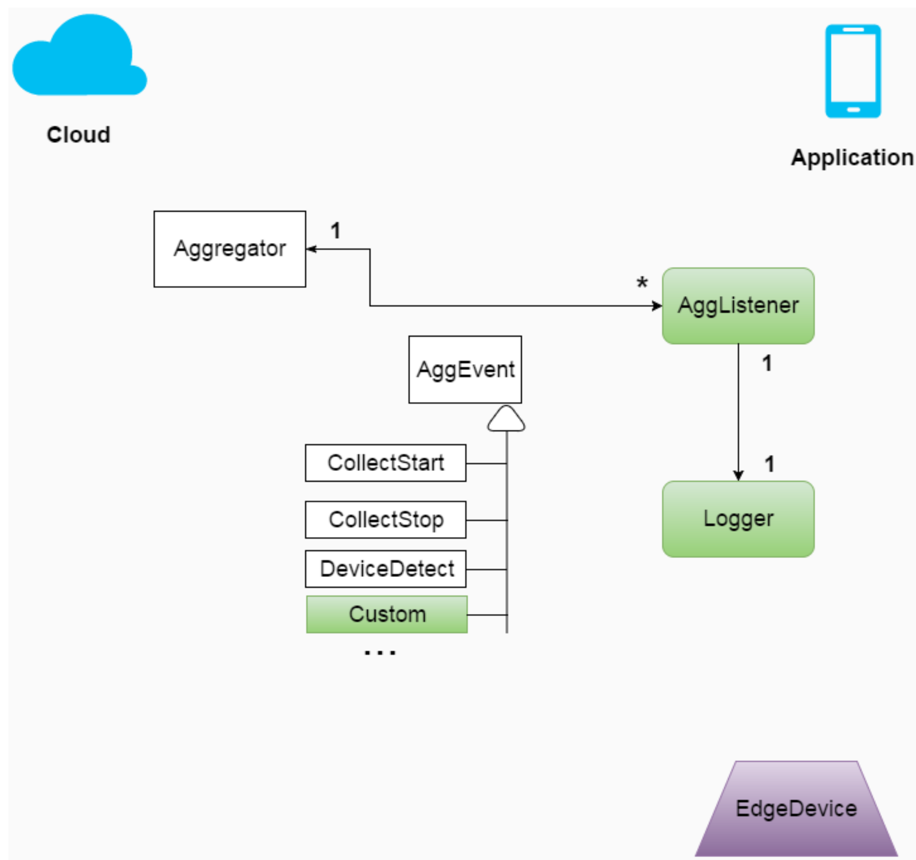

Рис. 3: Архитектура подкомпонента, отвечающего за логирование и обработку событий

Для каждого объекта-агрегатора конфигуратором создаётся объект класса *AggListener* (далее слушатель). По умолчанию любой конфигуратор будет возвращать объект класса *DefaultLoggingAggListener*, который будет писать в логи компонента хронику событий. Слушатель может реагировать на различные события агрегации, которые изначально поддерживаются компонентом: начало сбора с датчика, конец сбора с датчика, агрегатор запущен, агрегатор остановлен, возникла ошибка сбора, обнаружен новый датчик. Реакция на каждое такое событие может быть модифицирована разработчиком. При построении архитектуры данного компонента за основу был взят шаблон проектирования Observer[4]. Разработчик также может добавлять свои события, наследуясь от класса *AggEvent*, и задавать реакцию своих слушателей на них.

Также важным элементом архитектуры является интерфейс *Logger*

(объект класса, реализующего этот интерфейс, далее будет называться логгером). Любой логгер предоставляет функционал записи принятых сообщений в файлы и их чтение. По умолчанию любой конфигуратор возвращает логгера, который работает с системными потоками ввода и вывода.

3.1.4. Локальное сохранение данных

Рассматриваемый подкомпонент ответственен за долгосрочное сохранение данных принятых с датчиков подкомпонентом сбора. Разработанная архитектура подкомпонента схематично представлена на рисунке 4.

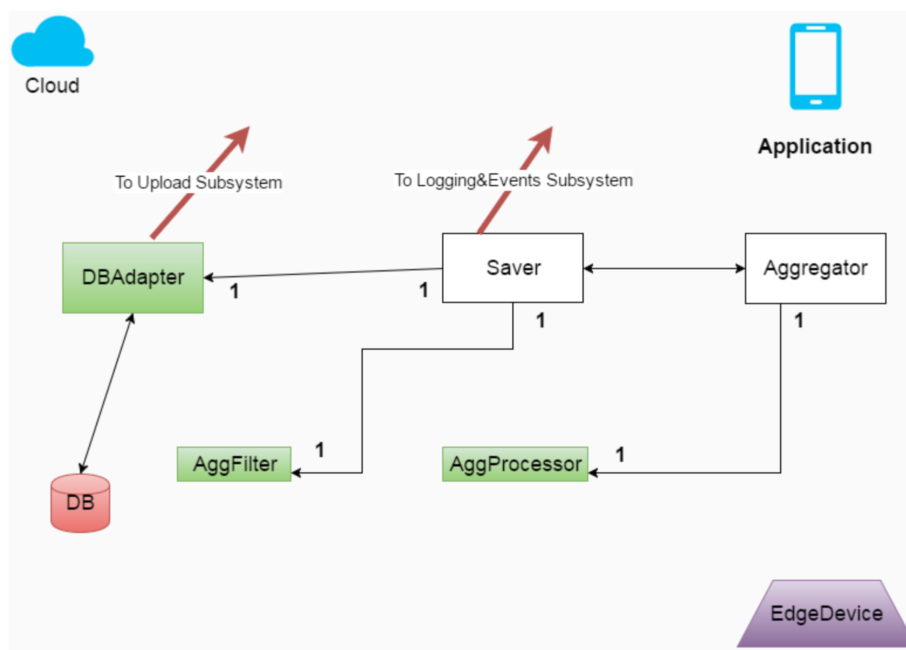

Рис. 4: Архитектура подкомпонента, отвечающего за локальное сохранение данных

Ключевыми классами этого подкомпонента являются $DBAdapter$, $AggFilter$, $AggProcessor$.

- $AggProcessor$
  Интерфейс, который используется для обработки накопленных данных перед их сохранением. Предоставляет метод, который принимает коллекцию данных и возвращает её модификацию. По

умолчанию любой конфигуратор будет возвращать реализацию $AggProcessor$, которая не меняет входную коллекцию.

- $DBAdapter$

  Интерфейс, предоставляющий методы для сохранения и извлечения данных из долгосрочного и надёжного хранилища. Без вмешательства разработчика по умолчанию любой конфигуратор возвратит объект класса $DefaultRAMAdapter$, который будет сохранять данные в оперативной памяти устройства вместо более надёжного места и предоставлять к ним удобный доступ. Объект класса, реализующего интерфейс $DBAdapter$ (далее адаптер) используется объектом класса $Saver$ (далее сохранитель). Данная связь классов объясняется использованием шаблона проектирования Strategy[4]. Для каждого объекта-агрегатора разработчик может задавать своё поведение сохранителя: расписание сохранения, объём данных, при накоплении которого следует произвести сохранение, и так далее.

- $AggFilter$

  Тот же интерфейс, который был описан в 3.1.2. Данный элемент полезен для фильтрации коллекции уже модифицированных с помощью объекта класса $AggProcessor$ данных.

3.1.5. Выгрузка данных

Назначение рассматриваемого подкомпонента – различные манипуляции с локально сохранёнными данными, которые в первую очередь включают в себя извлечение данных из долгосрочного хранилища, отправку на отдалённый сервер. Разработанная архитектура подкомпонента схематично представлена на рисунке 5.

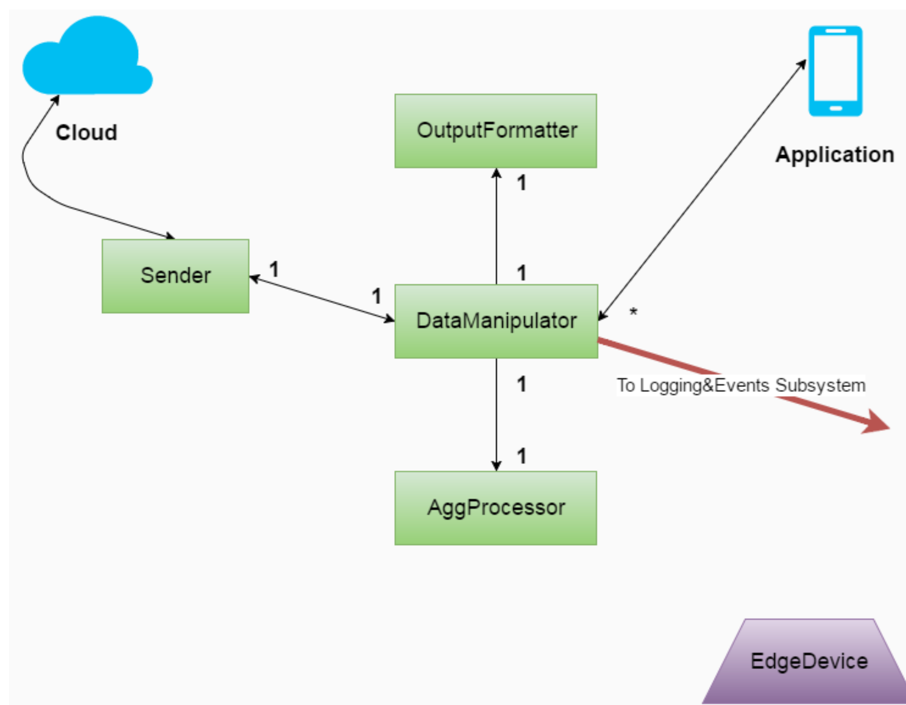

Рис. 5: Архитектура подкомпонента, отвечающего за выгрузку данных

Ключевыми классами этого подкомпонента являются $DataManipulator$, $OutputFormatter$, $Sender$, $AggProcessor$.

- $AggProcessor$

  Тот же интерфейс, который был описан в 3.1.4. Данный элемент может быть полезен для модификации давно хранящейся информации перед отправкой данных на отдалённый сервер.

- $OutputFormatter$

  Абстрактный класс, предоставляющий методы для форматирования подготовленных для выгрузки на отдалённый сервер данных. Объекты класса, реализующего интерфейс $OutputFormatter$ далее будем называть видопреобразователями. Конечный формат данных видопреобразователя можно подменить до и после построения последнего. Любой конфигуратор по умолчанию при запросе на построение видопреобразователя будет строить объект класса $DefaultJSONOutputFormatter$, который будет производить сериализацию данных в формат JSON с помощью программной библиотеки jackson[7].

- *Sender*

  Интерфейс, предоставляющий функционал для работы с конкретным отдалённым сервером. Объект класса, реализующего интерфейс *Sender* (далее отправитель) производит аутентификацию на отдалённом сервере (используя настройки предоставителя), организует отправку форматированных видопреобразователем данных на сервер и её прерывание. Любой конфигуратор по умолчанию при запросе на построение отпарвителя будет строить объект класса *DefaultDropboxSender*, который будет проходить аутентификацию на сервере Dropbox[5] с помощью настроек предоставителя и загружать данные в облако с помощью его API.

- *DataManipulator*

  Класс, который предоставляет интерфейс для извлечения долгосрочно сохранённых данных и организацию их отправки на отдалённый сервер с последующей очисткой памяти при необходимости. Объект класса *DataManipulator*, по умолчанию возвращаемый любым конфигуратором, далее будет именоваться манипулятором. Любой манипулятор имеет в распоряжении некоторый адаптер к долгосрочному хранилищу и отправителя, что позволяет ему предоставлять следующий функционал: извлечение из хранилища данных, удовлетворяющих каким-либо условиям, контроль транзакционности отправки данных, удаление отправленных данных. Отправка данных манипулятором была спроектирована с использованием шаблона проектирования Strategy.

3.2. Выбор программной платформы для реализации агрегаторского компонента фреймворка

Основными характеристиками программной платформы при ее выборе для разработки агрегаторского компонента были:

1. Кроссплатформенность, возможность написания приложений под мобильные устройства

2. Наличие возможности построения приложений с гибкой архитектурой

3. Распространенность среди разработчиков и простота развертывания

Первая характеристика платформы важна для создания мультиплатформенной системы. Нестационарный агрегатор может представлять из себя приложение, запущенное на смартфоне – поэтому от платформы требуется совместимость с мобильными устройствами. Вторая характеристика отвечает за модульность целевого фреймворка, которая нужна для предоставления возможности быстрого построения целевых систем сбора данных. Третья характеристика показывает, насколько легко будет потенциальному разработчику разобраться в архитектуре и начать непосредственное использование итогового фреймворка для простого построения своей системы сбора. В качестве платформы, чьи вышеперечисленные характеристики были приемлемыми для построения агрегаторского компонента, была выбрана Java платформа[10].

Во-первых, приложения под эту платформу пишутся на языке программирования Java, который известен как одни из лучших языков для написания высокопроизводительных кроссплатформенных приложений. Java приложение сможет запуститься на любом устройстве, на котором работает виртуальная машина Java, то есть пропадает необходимость потенциально трудоемкой перекомпиляции написанного приложения под разные конечные платформы.

Во-вторых, язык программирования Java использует парадигму объектно-ориентированного программирования, что позволяет строить приложения с гибкой архитектурой при правильном проектировании(например, при использовании паттернов проектирования[4]).

В-третьих, язык Java занимает первое место в рейтинге TIOBE[11] и имеет рейтинг равный 20.846 % на апрель 2016 года, что иллюстрирует популярность Java платформы среди разработчиков. К дополнительным достоинствам языка программирования Java можно отнести и то, что он является распространённым и богатым по предоставляемым

возможностям инструментом для создания приложений под мобильную платформу Android. Операционная система Android на настоящий момент не перестает быть доминантной среди платформ современных смартфонов[6]. Данный факт приводит к тому, что выбор Java платформы способствует созданию фреймворка, который позволял бы организовывать системы сбора данных, где в качестве агрегаторов могут выступать большинство существующих мобильных Android устройств.

Таким образом, представленная в 3.1 архитектура была создана на вышеописанной платформе. Апробирование полученного компонента описано ниже.

3.3. Создание прототипа

С целью апробации созданной агрегаторской части фреймворка было решено реализовать тестовое приложение на основе разработанного компонента под платформу Android ввиду её популярности (см. 3.2).

Тестовое приложение сборщика было реализовано под Android платформу версии 4.0 и выше. Запуск приложения производился на устройствах, имеющих разные версии установленной Android OS и храрктеристки мощности. Этими устройствами были: смартфон Explay N1 (Android 4.2.2), смартфон Asus Laser 2 Zenfone ZE 500KL (Android 5.0.2). Приложение было протестировано совместно с приложением, построенным на основе компонента фреймворка для датчиков. Часть фреймворка и тестовое приложение для датчиков разрабатывались в параллельной дипломной работе Рагимова Р.В.

Приложение сборщика предоставляло минимальный пользовательский интерфейс для настройки системы во время её работы. Приложение производило автоматический сбор данных с датчиков на Raspberry Pi по Bluetooth в формате JSON. Для пользователя было доступно задание расписания сохранения данных в локальную базу данных SQLite. Пользователь имел возможность просматривать логи, генерируемые системой. Система позволяла пользователю инициировать выгрузку сохранённых данных в облако Dropbox в формате JSON.

4. Заключение

В результате проделанной работы были получены следующие результаты:

- Сформированы общие требования на систему периодического сбора нестационарным агрегатором
- Проведен обзор фреймворков для организации систем периодического сбора нестационарным агрегатором
- Выбрана подходящая платформа для разработки фреймворка для организации периодического сбора нестационарным агрегатором
- Разработана агрегаторская часть целевого фреймворка и описаны её основные элементы
- С помощью агрегаторского компонента фреймворка создан прототип сборщика под платформу Android

Список литературы

[1] Abd Elwahab Boualouache Omar Nouali Samira Moussaoui, Derder Abdessamed. A BLE-based data collection system for IoT // New Technologies of Information and Communication (NTIC), 2015 First International Conference on New Technologies of Information and Communication. — 2015.

[2] Axibase Collector. — URL: http://axibase.com/products/axibase-time-series-database/writing-data/collector/.

[3] Axibase Time Series Database. — URL: http://axibase.com/products/axibase-time-series-database/.

[4] Design Patterns: Elements of Reusable Object-Oriented Software / Erich Gamma, Richard Helm, Ralph Johnson, John M. Vlissides. — 1994.

[5] Dropbox. — URL: https://www.dropbox.com.

[6] Global market share held by smartphone operating systems from 2009 to 2015. — URL: http://www.statista.com/statistics/263453/global-market-share-held-by-smartphone-operating-systems/.

[7] Jackson project. — URL: http://wiki.fasterxml.com/JacksonHome.

[8] Martin Atzmueller Katy Hilgenberg. Towards capturing social interactions with SDCF: an extensible framework for mobile sensing and ubiquitous data collection // MSM '13 Proceedings of the 4th International Workshop on Modeling Social Media. — 2013.

[9] Open Data Kit Sensors. — URL: https://opendatakit.org/use/sensors/.

[10] Oracle. Java Software. — URL: https://www.oracle.com/java/.

[11] TIOBE Index. — URL: http://www.tiobe.com/tiobe_index.

[12] ThingSpeak. ThingSpeak. — 2016. — URL: https://thingspeak.com (online; accessed: 14.05.2016).