Санкт-Петербургский Государственный Университет

Математико-механический факультет

Кафедра системного программирования

Разработка вспомогательной библиотеки для системы автоматического тестирования

Курсовая работа студента 344 группы

Рагимова Руслана Вугаровича

Научный руководитель: Васильев Игорь Борисович,

Руководитель группы перспективных разработок RAIDIX

Санкт-Петербург

2015

Оглавление

Введение 3

1. Постановка задачи 5

2. Существующие фреймворки для тестирования ПО 6

1.1 TETware 6

1.2 DejaGnu 7

1.3 STAF 7

1.4 Вывод 8

3. Описание инструментария 9

3.1 Фреймворк STAF 9

3.2 Язык программирования 10

4. Организация тестирования 11

4.1 Параметризация тестов 11

4.2 Логирование 11

4.3 Возврат результатов 12

5. Реализация библиотеки тестирования 13

5.1 STAFAdapter 14

5.2 RDAdapter 14

5.3 Adapter 14

5.4 Machines 15

5.5 Transports 15

6. Архитектура 16

7. Апробация 17

7.1 Тестирование обнаружения SDC 17

Заключение 18

Литература 19

Введение

Тестирование является неотъемлемой частью разработки любого программного продукта. На начальном этапе разработки очень часто бывает достаточно ручного тестирования. Но по мере развития и усложнения продукта ручное тестирование становится неэффективным, а зачастую и замедляет процесс выпуска продукта. Поэтому неизбежно встаёт вопрос по автоматизации процесса тестирования.

Одним из плюсов автоматического тестирования является строгое исполнение всех шагов тестового сценария. При усложнении системы ручное тестирование оказывается малоэффективным, поскольку в силу человеческого фактора не всегда оказывается возможным перевоспроизвести обнаруженную ошибку. С развитием программного продукта возникает потребность в частых рефакторингах кода, что влечёт необходимость в перетестировании. К тому же автоматическое тестирование может проходить без участия человека, например, в ночное время. Это является важным фактором, если при тестировании используется дорогостоящее оборудование. Однако если система подразумевает пользовательский интерфейс, то в некотором объёме всё же имеет смысл ручное тестирование, учитывающее человеческий фактор.

В конкретной работе будет рассмотрено решение по автоматическому тестированию многоконтроллерной распределённой системы хранения данных (СХД). Рассмотрим пример двухконтроллерной СХД.

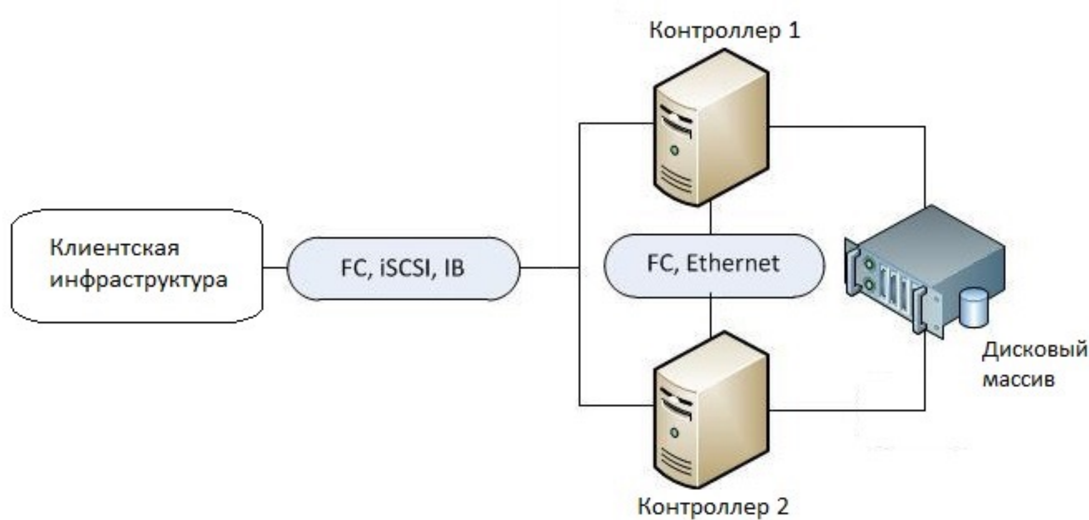

Рис. 1 Двухконтроллерная СХД

Системы следующих поколений вовлекут использование большого количества контроллеров в кластере хранения данных. Таким образом, тестирование таких систем в ручном режиме становится непригодным.

На данном этапе мы говорим о системе автоматического тестирования, где серверы и клиенты на Linux. Конфигурирование сущностей производится через средства Command line interface (CLI), не через Graphical user interface (GUI). Система должна повторять те действия, которые производятся при ручном тестировании. Поэтому в нашей системе выделяется ровно одна отдельная машина, на которой запускается тестовый скрипт. Этот тестовый скрипт будет выполнять действия на всех сущностях тестируемой системы. Поэтому необходимо внедрить «агентов» на каждую из вовлечённых в тестирование систем.

Был поставлен вопрос о написании ПО, обеспечивающего функции «агентов», с нуля или попытаться найти существующее среди ПО с открытым кодом. Таким образом, был рассмотрен ряд фреймворков, описанных в разделе обзора. По итогам рассмотрения был выбран фреймворк STAF.

1. Постановка задачи

Целью моей курсовой работы является разработка вспомогательной библиотеки для тестирования многоконтроллерной системы хранения данных, работающей под управлением ПО Raidix в ОС Linux. Для достижения данной цели мною были сформулированы несколько задач:

1. Выбрать язык программирования для написания библиотеки.
2. Проработать организацию тестовых скриптов (параметризация тестов, логирование, возврат результатов, …)
3. Разработать вспомогательные библиотеки и поместить в них код, многократно используемый в скриптах, организовав простой и удобный API для написания тестов:
    1. Управление транспортом iSCSI (по возможности InfiniBand, FibreChannel)
    2. Взаимодействие с СХД
    3. Логирование
    4. Работа с RAID
    5. Работа с блочными устройствами

2. Существующие фреймворки для тестирования ПО

Поскольку системы не налагают каких-либо ограничений на выбор фреймворка, логично обратить внимание на фреймворки, предоставляющие наибольший функционал. Также важна многоплатформенность.

1.1 TETware[1]

TETware - инструментарий для тестирования ПО, предоставляющий фреймворк для не распределённого (запуск тестов на локальной или удалённой машине) и распределённого (запуск различных частей теста на разных удалённых машинах) тестирования.

Возможность распределённого тестирования важна в нашем случае, т.к система хранения данных - прежде всего, сервис, у которого может быть более одного клиента.

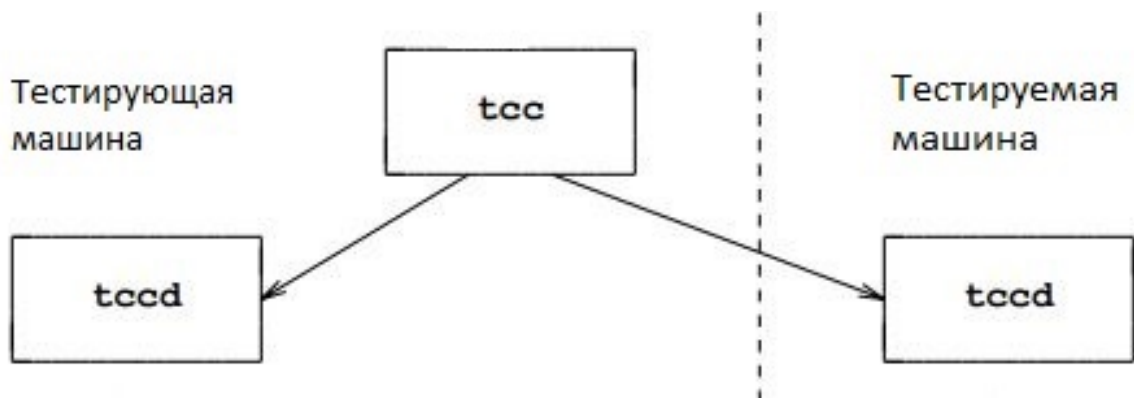

Рис. 2 Схема взаимодействия сущностей

Предполагается взаимодействие только между контроллером (tcc[2]) и «агентом» (tccd[3]), см. Рис. 2.

Имеется API для языков (* - только не распределённое тестирование):

- C
- Shell
- Java
- C++*
- Perl*
- Python*

Поддерживаемые платформы:

- UNIX
- Linux
- Windows 32-bit

---

[1] http://tetworks.opengroup.org

[2] tcc – Distributed TETware Test Case Controller

[3] tccd – Test Case Controller daemon

Поддержка Windows только до версии XP, что уже не актуально.

TETware используется в The X Test Suite.

К сожалению, на данный момент обновления не выпускаются.

1.2 DejaGnu[4]

Создатели DejaGnu позиционируют его как «фреймворк для тестирования других программ». Написан на Expect. Его цель - предоставить единый фронтенд для всех тестов. По сути, это библиотека процедур Tcl, позволяющая на её основе создавать системы для тестирования конкретного ПО.

Исполнительный движок тестов - утилита runtest.

Поддерживает POSIX - системы и Windows.

Используется для тестирования таких программ, как GCC, GDB.

Фреймворк весьма примитивен по своим возможностям.

Поддерживается по сей день.

1.3 STAF[5]

STAF - Software Testing Automation Framework. Мультиплатформенный фреймворк, в основе которого лежит идея многократно используемых компонентов, называемых сервисами (например, сервис логирования, сервис процессов).

STAF обеспечивает базу, необходимую для создания высокоуровневых решений по автоматизированному тестированию систем. К ряду встроенных сервисов имеется возможность добавить собственные сервисы, написанные на одном из поддерживаемых языков.

Исполнительным движком STAF служит STAX - eXecutive engine, построенный на базе трёх технологий: STAF, XML и Python. Позволяет полностью автоматизировать развёртывание, исполнение и анализ результатов тестовых наборов.

---

[4] http://www.gnu.org/software/dejagnu/

[5] http://staf.sourceforge.net/index.php

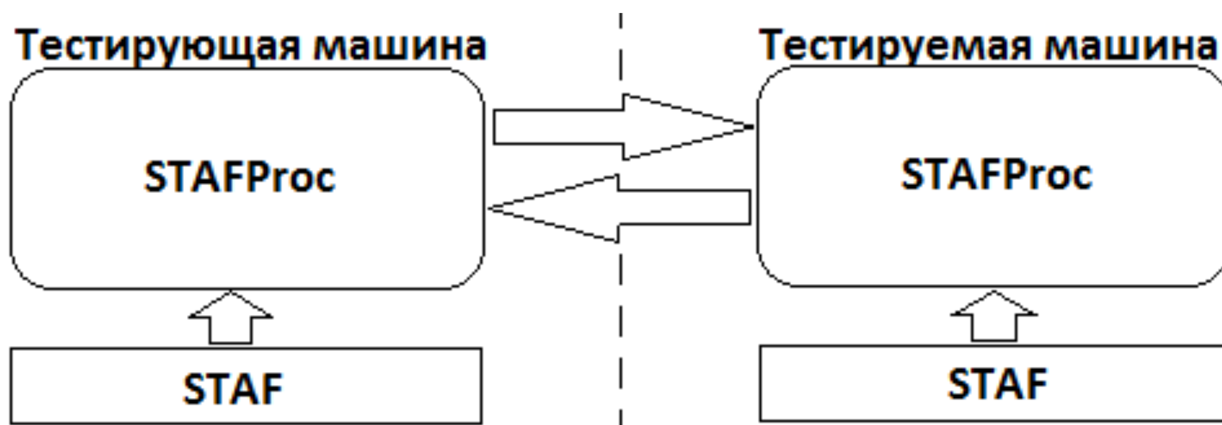

Рис. 3 Схема взаимодействия сущностей

На схеме (Рис. 3) видно, что сущности равноправны. На каждой сущности есть STAFProc - «агент» и STAF - клиент. Таким образом, любые две сущности могут взаимодействовать между собой без посредников.

Имеется API для работы с C/C++/Java/Rexx/Perl/Python/Tcl.

Поддерживаемые платформы:

- Linux
- Mac OS
- UNIX
- Windows 32/64

STAF используется при тестировании Xerox, HP, Intel, IBM, VMWare, Linux.

На данный момент разработка фреймворка активно продолжается.

1.4 Вывод

Среди рассмотренных фреймворков наиболее мощным и универсальным оказался STAF. Он поддерживает наибольшее количество платформ и до сих пор разрабатывается. Ко всему прочему, стоит обратить внимание на то, что это фреймворк используется многими крупными компаниями, что ещё раз подтверждает его превосходство.

3. Описание инструментария

3.1 Фреймворк STAF

STAF будет обеспечивать функции «агента» на каждой из сущностей, вовлечённых в тестирование. Для этого на всех машинах требуется запустить процесс STAFProc, который будет ждать входящих подключений. Этот процесс служит для обращения к сервисам, каждый из которых имеет свой функционал. Есть два типа сервисов: внутренние и внешние. Внешние сервисы должны быть зарегистрированы перед использованием. Есть возможность добавления своих внешних сервисов.

На тестирующей машине к STAFProc будет подключаться клиент STAF и сам Test Script через API STAF. На остальных машинах STAFProc ждёт подключений от тестирующей машины. Рассмотрим пример использования STAF для взаимодействия сущностей.

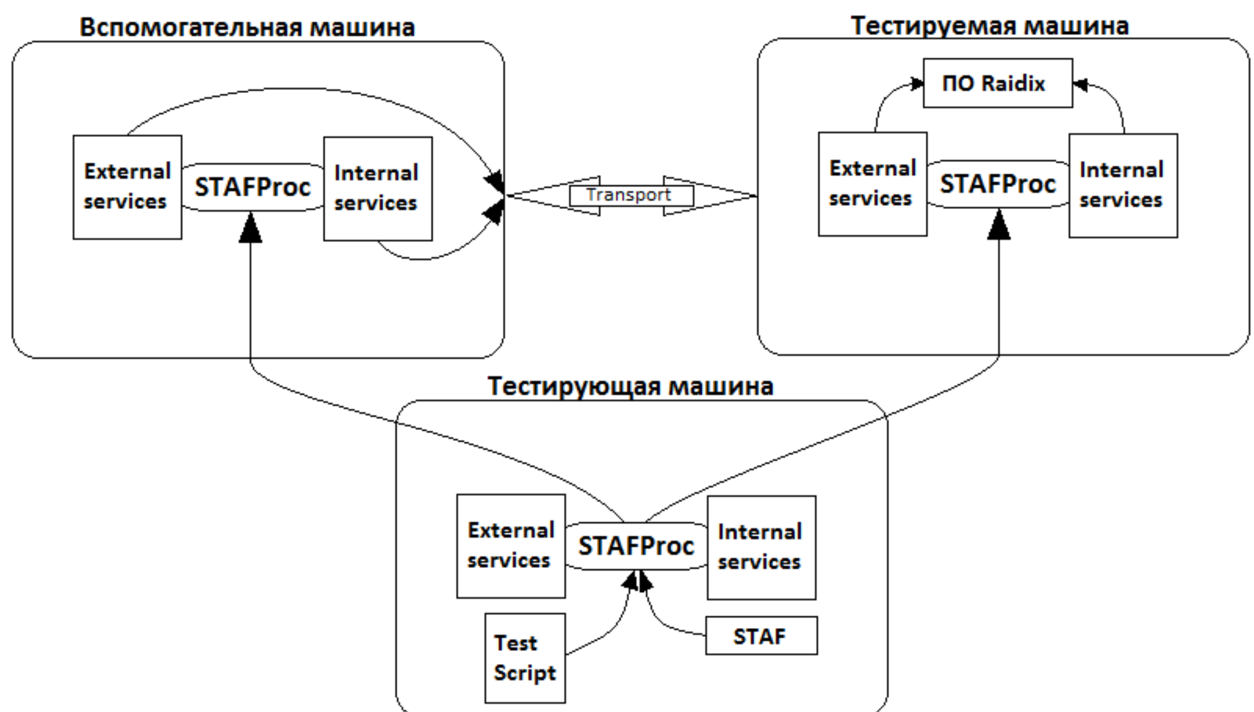

Рис. 4 Пример использования STAF

3.2 Язык программирования

Необходимо выбрать язык программирования, на котором будет реализована библиотека. На нём же и будут разрабатываться тестовые скрипты, использующие API этой библиотеки. Таким образом, нужен простой высокоуровневый язык, пользующийся популярностью среди программистов, т.к наборы тестов необходимо пополнять/изменять по ходу разработки. STAF поддерживает C/C++/Java/Rexx/Perl/Python/Tcl.

С не поддерживает ООП. C++/Java не поддерживают динамическую типизацию, что усложняет процесс разработки. Язык Rexx не является популярным.

Среди Perl/Python/Tcl был выбран Python как наиболее простой в изучении и достаточно популярный язык. Он изначально поддерживает многие удобные структуры данных (списки, словари, кортежи), функции работы со строками, имеет в своём арсенале огромное количество библиотек. Всё это позволит сделать написание тестов наиболее простым и быстрым.

4. Организация тестирования

Введём некоторые термины. На основе библиотеки, реализованной в рамках данной курсовой работы, будут создаваться так называемые Test Script'ы (тестовые скрипты/тестовые сценарии) - тесты, имеющие набор входных параметров.

Каждый Test Script имеет пролог и эпилог. Пролог включает в себя подготовку системы к проведению тестовых действий, а эпилог возвращает систему в исходное состояние.

Test Suite (набор тестов) - набор Test Script'ов, направленных на тестирование определённого функционала или свойств системы.

4.1 Параметризация тестов

Среди параметров каждого теста есть статические параметры, одинаковые для всех Test Script'ов (это, в основном, конфигурация оборудования), и динамические (для каждого Test Script'а свои). Исходя из этого было решено передавать динамические параметры через командную строку при вызове тестового скрипта, а статические параметры через переменные окружения (т.к при создании нового процесса в Linux, окружение копируется, и новый процесс имеет те же переменные окружения, что и старый).

Конфигурация оборудования содержит информацию об используемом транспорте для подключения клиентов к СХД, IP-адрес Management-интерфейса (для непосредственного управления СХД) и другую информацию.

4.2 Логирование

Фреймворк предоставляет сервис логирования. Логирование будет осуществляться локально на машине, исполняющей тестовый скрипт. Есть различные уровни логов (Fatal, Error, Info, …). В данном случае будут использоваться следующие уровни:

- Fatal - ошибки, при возникновении которых Test Script завершает работу
- Error - ошибки, не ведущие к завершению работы. Могут быть ожидаемы
- Info - информация о текущем шаге тестового сценария
- User1 - пользовательский уровень. Будет использоваться для логирования команд, отправляемых на исполнение
- User2 - для логирования результатов исполнения отправленных команд

4.3 Возврат результатов

Test Script завершает свою работу с кодом:

- 0 - тест пройден
- 2 - произошла ошибка, в результате которой скрипт прервал работу
- 16 - скрипт завершился успешно, но задача, поставленная перед тестируемой системой, не выполнена

5. Реализация библиотеки тестирования

На данном этапе реализации библиотеки будет достаточно следующих сервисов STAF: PROCESS и LOG. Сервис PROCESS в нашем случае используется для запуска команд на удалённых машинах, а сервис LOG для локального логирования. Ниже приведена схема взаимодействия тестирующего скрипта с ПО на всех машинах, вовлечённых в тестирование.

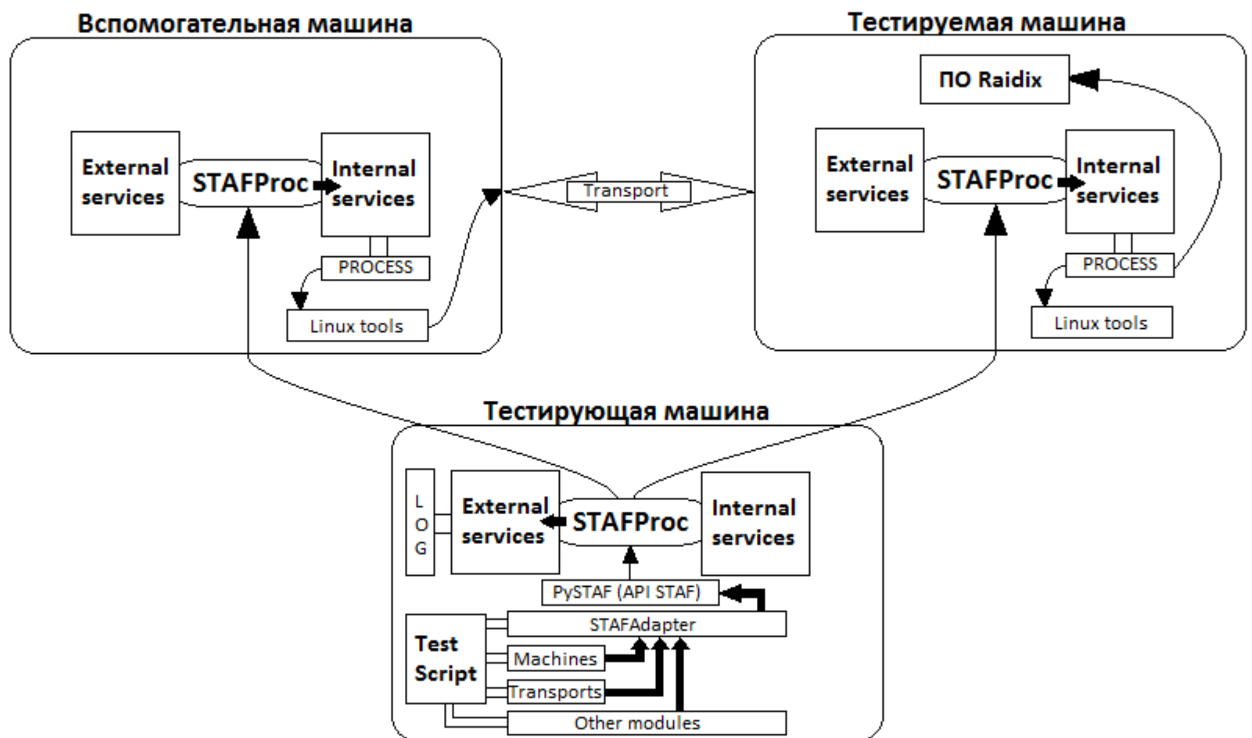

Рис. 5 Взаимодействие сущностей

STAFAdapter, Machines, Transports - модули, подключаемые к Test Script’у. Будут рассмотрены далее.

5.1 STAFAdapter

Класс, реализующий необходимые функции для взаимодействия с фреймворком STAF через API (модуль PySTAF). Основной функционал:

- Исполнение команды на удалённой машине, используя STAF сервис «PROCESS», и получение результата исполнения.
- Логирование сообщений через сервис «LOG»

5.2 RDAdapter

Класс, скрывающий специфику работы с конкретной СХД.

Основной функционал:

- Создание/удаление RAID
- Чтение/запись из/в RAID
- Очистка кэш-памяти RAID
- Получение объёма RAID
- Информация об SDC[6]
- Установка режима работы SDC

5.3 Adapter

Модуль для работы с входными параметрами теста. Есть набор необязательных параметров, которые определятся во время исполнения случайным образом или при помощи заданных умолчаний для этих параметров, исходя из значений других параметров. Например, исходя из количества дисков в системе, можно определять уровень RAID (не единственным образом, конечно же).

---

[6] SDC - Silent Data Corruption (скрытое повреждение данных)

5.4 Machines

Этот модуль включает в себя классы SCSIDevice, RAID, Machine, NUT (Node Under Test), AUX (Auxiliary[7]).

Класс SCSIDevice осуществляет работу с дисковыми устройствами SCSI.

Класс RAID использует RDAdapter для взаимодействия с СХД. Этот класс задаёт удобную и понятную абстракцию над RDAdapter, дополняя его функционал следующими пунктами:

- Инициализация (из /dev/zero) RAID
- Получение номера первого/последнего сектора, используемого RAID на каждом из физических дисков, его составляющих

Классы NUT и AUX наследуются от класса Machine и отвечают за работу с тестируемыми и вспомогательными узлами, соответственно.

5.5 Transports

Данный модуль содержит несколько классов, инкапсулирующих функции по работе с транспортами между СХД и клиентом.

Класс Transport конструируется по экземплярам классов NUT и AUX, определяет тип транспорта, который можно использовать между NUT - AUX, и создаёт экземпляр класса, соответствующего данному транспорту. Имеет методы Connect/Disconnect.

Класс ISCSI скрывает особенности конфигурирования транспорта iSCSI, а именно реализует методы discovery, login и logout.

[7] Auxiliary - вспомогательный узел. Является клиентом по отношению к СХД.

6. Архитектура

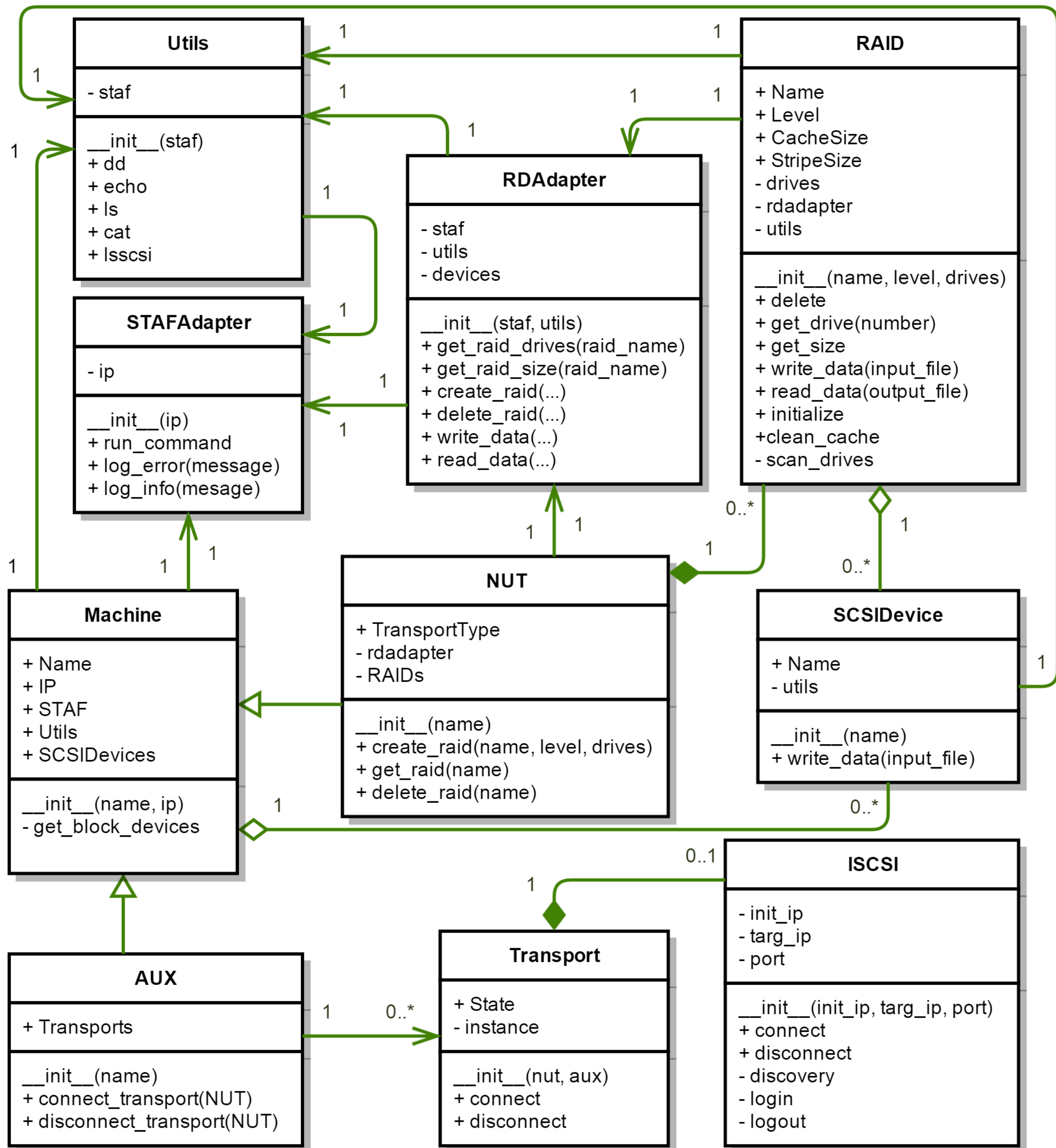

Рис. 6 UML-диаграмма классов

7. Апробация

7.1 Тестирование обнаружения SDC

Чтобы протестировать в простейшем случае функции по обнаружению SDC, достаточно двух машин: тестирующая и тестируемая. В данном случае нет необходимости во вспомогательной машине.

Сначала тестовый скрипт получает входные параметры и на их основе конструирует экземпляр класса NUT:

```
nut0 = NUT(nut_get()[0])
```

Далее создаётся RAID с именем, заданным по умолчанию в модуле Defaults:

```
raid = nut0.create_raid(Defaults.RAIDName, get_raid_level(), get_raid_drives())
```

Для тестирования SDC необходимо произвести инициализацию RAID:

```
raid.initialize(count=10)  # Инициализируем первые 10 секторов
```

Затем необходимо включить функцию SDC в режим Detection:

```
raid.set_sdc_mode('detection')  # Переводим функцию SDC в режим Detection
```

Записываем в RAID один сектор случайных данных:

```
raid.write_data(input_file='/dev/urandom', count=1)
```

Теперь портим данные, записав сектор случайных байтов в начало одного из физических дисков, на котором располагается RAID массив:

```
raid.get_drive(get_sdc_drives()[0]).write_data(input_file='/dev/urandom',
                                               seek=int(raid.get_start_sector()),
raid.clean_cache()  # Очищаем кэш-память СХД     count=1)
```

Данные испорчены, теперь пробуем их прочитать и проверяем SDC Info:

```
try:
    # Чтение из испорченной области
    raid.read_data(output_file='/dev/null', count=1)
except RAIDException:
    # Если область испорчена, то чтение не выполнится,
    # и возникнет исключение в классе RAID
    # Проверяем, не пуст ли список обнаруженных SDC
    if raid.get_sdc_info() is not None:
        sys.exit(0)  # Тест успешно пройден
    else:
        sys.exit(16)  # Тест завершён, но функция не сработала
```

На этом тест заканчивается. Следует обратить внимание, что в коде теста нет необходимости заботиться о логировании. Всё это скрыто внутри библиотеки, и каждое действие автоматически вносится в лог.

Заключение

В рамках данной курсовой работы были достигнуты следующие результаты:

- Выбран фреймворк, на основе которого будет строиться система автоматического тестирования
- Проработана организация тестирования
- Разработана вспомогательная библиотека, состоящая из набора модулей на языке Python, для системы автоматического тестирования. Обладает следующей функциональностью:
  1. Управление транспортом iSCSI
  2. Взаимодействие с СХД и вспомогательными машинами
  3. Логирование
  4. Основные функции по работе с RAID
  5. Функции по работе с блочными устройствами
- Проведена апробация получившейся библиотеки на примере конкретного теста.

Литература

[1] Python 2.7 documentation, URL: https://docs.python.org/2.7/

[2] Linux Open-iSCSI, URL: http://www.open-iscsi.org/docs/README

[3] STAF V3 User's Guide, URL: http://staf.sourceforge.net/current/STAFUG.htm

[4] Python User's Guide for STAF Version 3, URL: http://staf.sourceforge.net/current/STAFPython.htm

[5] James Rumbaugh, Ivar Jacobson, Grady Booch. «The Unified Modeling Language Reference Manual Second Edition», 2004

[6] Jon Tate, Pall Beck. «Introduction to Storage Area Networks and System Networking», 2012

[7] С.В. Синицын, Н.Ю. Налютин. «Верификация программного обеспечения. Курс лекций», 2006