*Архитектура CLI*

*Компоненты*

*1) Backend CLI*

От сервера ожидаем следующую функциональность и поведение:

* Динамическая конфигурация сервера при запуске: конфигурируем известные команды, уровень доступа для каждой команды, а также машину состояний CLI. При конфигурации машины состояний CLI мы конфигурируем состояния, устанавливаем начальное состояние и конфигурируем таблицу переходов. Во время конфигурации состояний мы устанавливаем, какие команды доступны в каждом из состояний (в каждом из режимов работы CLI). Во время конфигурации переходов мы устанавливаем, вызовы каких команд приводят к смене состояния.
* Аутентификация и авторизация клиента на сервере. Автоматический выход (logout) по таймауту.
* Одновременная работа нескольких клиентов. Возможность одновременного выполнения той или иной команды определяется реализацией самой команды.
* После аутентификации клиента, создается клиентская сессия. Основой клиентской сессии является машина состояний CLI. При выходе клиента из системы клиентская сессия уничтожается.
* Взаимодействие с клиентом (с клиентами): прием запроса на выполнение команды от клиента, нахождение подходящей команды, проверка на возможность выполнения найденной команды в данный момент времени (при данном состоянии клиентской сессии), выполнение команды, возврат результата на сторону клиента, сигнал об окончании выполнения команды.
* Команды на backend'е — это команды для взаимодействия с железом и/или изменения состояния backend'а (в основном, для изменения состояния клиентской сессии).
* Если выполнение команды завершилось успешно, то необходимо известить об этом машину состояний CLI. Если выполнение команды завершилось с ошибкой, то машину состояний CLI мы об этом не извещаем.
* Для каждого клиента, текущая функциональность сервера определяется текущим состоянием машины состояний CLI (текущим режимом работы).
* Отправка списка всех поддерживаемых команд, вместе с дополнительной информацией о каждой команде, на сторону клиента.

*2) Frontend CLI*

От клиента ожидаем следующую функциональность и поведение:

* Аутентификация и авторизация на сервере.
* Два режима работы клиента: режим ввода (когда пользователь может ввести команду на клиенте) и режим выполнения.
* По умолчанию – клиент находится в режиме ввода. Когда пользователь вводит команду и запускает ее на выполнение (нажимая клавишу Enter), клиент переключается в режим выполнения. В режиме выполнения у пользователя нет возможности для ввода. После окончания выполнения команды, клиент переключается обратно в режим ввода.
* Выполнение команды происходит следующим образом: прием ввода пользователя на клиенте, разбор ввода пользователя на цепочку команд, выполнение цепочки команд, вывод результата выполнения цепочки команд на клиенте. Команды могут быть двух типов: frontend команды (это команды, которые выполняются непосредственно на frontend'е), и backend команды (выполнение которых происходит на backend'е).
* Команды, выполняющиеся на frontend'е могут выполнятся как последовательно, так и параллельно (это зависит от реализации). Команды, выполняющиеся на backend'е, всегда выполняются последовательно (отосительно клиента). Другими словами, команды для backend'а являются точками синхронизации выполнения команд.
* Команды на frontend'е — это команды для обработки результата выполнения предыдущих команд и/или изменения состояния frontend'а.
* Редактирование ввода пользователя (in bash-like style).
* Автодополнение ввода для всех известных на сервере (на backend'е) команд.
* Помощь по каждой из известных команд.
* История команд.
* Формирование внешнего вида в зависимости от пользователя и текущего режима работы.

*3) Proxy (MITM)*

От proxy ожидаем следующую функциональность и поведение:

* Proxy работает как на стороне сервера, так и на стороне клиента.
* Proxy инкапсулирует реальный протокол взаимодействия как от клиента, так и от сервера.
* Клиент и сервер взаимодействуют друг с другом через стандартный механизм взаимодействия между процессами в языке Erlang. При этом в действительности, и клиент и сервер взаимодействуют с процессами Proxy, а не между собой. Пусть, например, процесс A на стороне клиента считает, что взаимодействует с процессом B на стороне сервера. В действительности же, процесс A взаимодействует с процессом Proxy на стороне клиента X, а процесс B взаимодействует с процессом Proxy на стороне сервера Y. При этом процессы X и Y взаимодействуют друг с другом через реальный протокол взаимодействия, создавая видимость, что процессы A и B взаимодействуют друг с другом напрямую.

*Протокол взаимодействия*

*1) Аутентификация пользователя на сервере*

Запрос на сервер:

Клиент: **{login,***LoginName***,** *Password***}**, где *LoginName* - имя пользователя, *Password* - пароль

Ответ с сервера (один из перечисленных ниже вариантов):

* Сервер (если пользователь аутентифицирован): **{login\_success,** *IsAdmin, CliMode, SessionPid, GreetingMessage***}**, где *IsAdmin* – флаг, определяющий является ли пользователь администратором, *CliMode* – строковое представление текущего режима работы (текущее состояние машины состояний CLI), *SessionPid –* идентификатор процесса (Pid), созданный для обработки запросов от данного клиента (идентификатор клиентской сессии)*,* *GreetingMessage* – строка с приветствием.
* Сервер (если пользователь не аутентифицирован): **{login\_fail,** *Reason***}**, где *Reason* - причина, по которой пользователь не смог пройти аутентификацию.

*2) Информация обо всех доступных командах*

Запрос на сервер:

Клиент: **{commands\_info}**.

Ответ с сервера (один из перечисленных ниже вариантов):

* Сервер (когда информация о командах получена успешно): **{commands\_info\_result, [{command\_info,** *CommandName, CommandBody, CommandHelp***}, …]}**, где *CommandName* – имя команды, *CommandBody* – тело команды, *CommandHelp* – справочная информация о команде.
* Сервер (когда происходит ошибка при получении информации о командах): **{commands\_info\_fail,** *Reason***}**, где *Reason* – причина ошибки.

*3) Выполнение команды CLI*

Запрос на сервер:

Клиент: **{command,** *CommandText***}**, где *CommandText* - текст команды, введенный на клиенте.

Ответ с сервера:

* Вывод команды (цепочки команд): **{command\_output,** *CommandOutput***}**, где *CommandOutput* - вывод команды (цепочки команд).
* Вывод ошибок команды (цепочки команд): **{command\_error,** *CommandError***}**, где *CommandError* - вывод ошибок команды (цепочки команд).
* Завершение выполнения команды: **{command\_end,** *CommandCompletionCode, CliMode***}**, где *CommandCompletionCode* - код завершения команды (цепочки команд), *CliMode –* строковое представление текущего режима работы (текущее состояние машины состояний CLI).
* Ошибка сервера при выполнении команды (цепочки команд): **{command\_fail,** *FailReason, CliMode***}**, где *FailReason* — причина/описание ошибки сервера при выполнении команды, *CliMode –* строковое представление текущего режима работы (текущее состояние машины состояний CLI).

В ответ на запрос с сервера приходит одно или несколько сообщений **{command\_output,** *CommandOutput***}** и **{command\_error,** *CommandError***}**, причем эти сообщения могут идти в любом порядке и любом количестве. В конце ответа всегда приходит сообщение **{command\_end,** *CommandCompletionCode, CliMode***}**. Если во время выполнения команды произошла ошибка на сервере, то с сервера придет сообщение **{command\_fail,** *FailReason, CliMode***}** и выполнение команды прекратится. Если при этом были сгенерированы одно или несколько сообщений **{command\_output,** *CommandOutput***}** и **{command\_error,** *CommandError***}**, то они будут отправлены клиенту.

*4) Выход пользователя с сервера*

Запрос на сервер:

Клиент: **{logout}**.

Ответ с сервера:

Сервер: **{logout\_success}**.

*Машина состояний (конечный автомат) CLI*

На данный момент конечный автомат имеет следующий вид:

*1) Состояния (режимы работы CLI)*

* Фундаментальный режим работы
* Режим глобальной настройки
* Режим настройки интерфейсов
* Режим настройки группы интерфейсов
* Режим настройки VLAN

2) События, приводящие к смене состояния (команды)

* Команда ***configure terminal***: фундаментальный режим работы -> режим глобальной настройки.
* Команда ***interface***: режим глобальной настройки -> режим настройки интерфейсов.
* Команда ***interface range***: режим глобальной настройки -> режим настройки группы интерфейсов.
* Команда ***vlan***: режим глобальной настройки -> режим настройки VLAN.
* Команда ***end***: режим глобальной настройки, режим настройки интерфейсов, режим настройки группы интерфейсов, режим настройки VLAN -> фундаментальный режим работы.
* Команда ***exit***: режим настройки интерфейсов, режим настройки группы интерфейсов, режим настройки VLAN -> режим глобальной настройки; режим глобальной настройки -> фундаментальный режим работы.

Все остальные команды состояние машины состояний CLI не меняют. В машину состояний CLI уведомление о выполнении команды, попадает только после ее успешного выполнения. Если команда выполняется не успешно, то машина состояний CLI уведомления не получает.

Машина состояний создается при создании сессии для каждого клиента после его аутентификации. При завершении работы клиента, сессия связанная с ним (и, соответственно, машина состояний CLI) уничтожаются.

Структура (состояния и переходы) машины состояний будет меняться. Так, например, настройка некоторых протоколов (BGP, OSPF, MSTP) требуют добавления одного или более режима CLI. Поэтому необходимо реализовать загрузку конфигурации конечного автомата из внешнего источника, вместо жесткой реализации в теле конечного автомата.

*Работа с реальным "железом"*

При реализации CLI (а точнее, backend'а CLI) напрямую с "железом" мы работать не будем: для CLI будет предоставлен некоторый API.

*Некоторые архитектурные решения*

*Формат конфигурационных файлов*

Все конфигурационные файлы будут хранить данные в виде выражений языка Erlang. С одной стороны это сильно облегчит и ускорит разбор таких файлов, а с другой стороны особо не усложнит синтаксис таких конфигурационных файлов.

Реализация машины состояний CLI

Машину состояний CLI мы будем реализовывать на базе OTP (модуль gen\_fsm). В будущем мы будем использовать более легковесное решение для реализации машины состояний, не использующее отдельный процесс.

*Реализация команд и потоков (для перенаправления стандартного ввода, стандартного вывода и стандартного вывода для ошибок)*

От команд мы ожидаем стандартного Unix-подобного поведения: каждая команда имеет доступ к потокам ввода, вывода и вывода ошибок; мы можем для каждой команды перенаправлять эти потоки и мы можем объединять несколько команд в одну цепочку. Кроме того, каждая команда при завершении работы возвращает некоторый код возврата, показывающий успешно или нет выполнилась данная команда. На данном этапе мы не реализуем обработку цепочек команд, но учитываем их появление в реализуемой архитектуре.

*Реализовывать команды и потоки мы будем следующим образом*

Команды могут быть двух типов: frontend команды (это команды, которые выполняются непосредственно на frontend'е), и backend команды (выполнение которых происходит на backend'е).

*Для команда на frontend'е справедливо следующее:*

* Команды в процессе своей работы могут как не выходить за пределы frontend'а (например, команда помощи ?), так и взаимодействовать с backend'ом (например, команда login). Для вызова backend команд на frontend'е используется специальная команда-обертка.
* Каждая команда представляет отдельный процесс Erlang.
* Очередь сообщений каждой команды является потоком ввода. В эту очередь сообщений попадает ввод с клиента, либо вывод из предыдущей команды (а также вывод ошибок, если есть соответствующее перенаправление). Часть команд потоком ввода не пользуется.
* Каждая команда имеет "ссылки" на две другие команды (знает PID'ы соответствующих процессов): какой команде отсылать вывод и какой команде отсылать ошибки. Эти две "ссылки" являются потоками вывода и вывода ошибок для данной команды.
* Каждая цепочка команд завершается специальной командой-приемником,. Эта команда принимает вывод ошибок от других команд, если нет перенаправления вывода ошибок в этих командах. Также эта команда принимает вывод от предпоследней команды из цепочки команд. Эта специальная команда может, как накопить весь вывод и только потом вывести его на консоль, так и выводить на консоль весь вывод команд по мере поступления.
* Команда может, как накопить весь вывод и потом отослать следующей за ней команде, так и отсылать весь вывод по мере его появления. Это поведение зависит как от деталей реализации, так и от состояния на frontend'е.
* Команда, когда завершает свою работу, сообщает об этом контексту выполнения вместе с кодом возврата (как в реальных \*nix-системах). Если код возврата свидетельствует о нормальном завершении выполнения команды (если код возврата равен 0), то контекст выполнения начинает выполнение следующей команды из цепочки (либо завершает ее выполнение, если это последняя команда). Если код возврата свидетельствует о ненормальном завершении выполнения команды (если код возврата не равен 0), то контекст выполнения заканчивает выполнение цепочки и выполняет последнюю команду из этой цепочки (которая является специальной командой, см. выше).
* Связи между командами (перенаправление потоков вывода) и специальную команду в конце цепочки команд создает модуль разбора команд.
* На frontend'е, команды могут выполнятся как последовательно, так и параллельно (это зависит от реализации). Любой запрос на backend всегда выполняются последовательно (отосительно клиента). Другими словами, любое обращение на backend является точками синхронизации выполнения команд.
* Каждая команда (модуль, реализующий каждую команду) содержит информацию о ней самой: имя команды и некоторое описание команды. Имя команды используется для поиска команды, описание команды - при запросах помощи на frontend'е.
* Команда, обрабатывающая запросы помощи, поиск информации ведет как среди frontend команд, так и среди информации, полученной с backend'а. При поиске, информация из frontend команд имеет приоритет над информацией, полученной с backend'а (возможно, что в будущем мы будет комбинировать информацию из frontend команд и полученную из backend'а).

*Для команда на backend'е справедливо следующее:*

* Реализация каждой команды может как использовать отдельный процесс, так и выполнятся в процессе контекста клиента на backend'е.
* Команда "ввод с клиента" получает в качестве одного из параметров.
* Для команд используется специальный приемник данных, отсылающий (и, возможно, накапливающий) как обычный вывод, так и вывод ошибок обратно на frontend (в frontend обертку для backend команды).
* Команда может, как накопить весь вывод и потом отослать его на frontend, так и отсылать весь вывод по мере его появления. Это поведение зависит как от деталей реализации, так и от состояния на backend'е.
* Команды всегда выполняются последовательно относительно одной и той же клиентской сессии.
* Каждая команда (модуль, реализующий каждую команду) содержит информацию о ней самой: имя команды и некоторое описание команды. Имя команды используется для поиска команды, описание команды - при запросах помощи на клиенте.

*Реализация модуля разбора команд на backend'е*

В данной версии модуль разбора команд просто идентифицирует команду по ее имени. Для этого он последовательно выделяет из строки с командой одну или несколько подстрок (разделителем подстрок является символ пробела), после чего по этим подстрокам находит команду (у нас есть команды, состоящие из нескольких строк). Команда позволяет определить точку входа (на самом деле мы определяем только имя модуля, т.к. имя функции будет жестко задано), после чего создает процесс для этой команды. После этого модуль разбора команд создает специальный приемник данных (см. выше) для взаимодействия с клиентом (с frontend'ом).

*Реализация модуля разбора команд на frontend'е*

На данный момент реализация совпадает с реализацией модуля разбора команд на backend'е. Есть одно исключение из этого совпадения: если модуль разбора не находит подходящую команду среди frontend команд, то будет произведен поиск среди backend команд. Если поиск среди backend команд будет успешен, то будет создана специальная команда-обертка для вызова найденной команды на backend'е. В ближайшем будущем появится поддержка цепочки команд, перенаправления ввода и вывода и, соответственно, модуль разбора команд на frontend'е значительно усложниться.

*Реализация контекста выполнения команд на backend'е*

Контекст выполнения команд управляет выполнением на backend'е запроса (команды) от frontend'а. Перед запуском команды на выполнение контекст выполнения проверяет, доступна ли данная команда в данном состоянии машины состояний CLI, а также хватает ли прав у текущего пользователя на выполнение этой команды. Если все проверки выполняются успешно, то контекст запускает команду на выполнение, после чего ожидает результат ее выполнения (по коду возврата). В случае успешного выполнения контекст уведомляет об этом машину состояний CLI; если выполнение команды закончилось с ошибкой, то состояние машины состояний CLI меняться не будет. После этого (вне зависимости от того, успешно или не успешно закончилось выполнение команды) контекст уведомляет специальный приемник данных об окончании выполнения команды (а специальный приемник данных уведомляет об этом frontend).

Реализация контекста выполнения команд на frontend'е

Контекст выполнения команд содержит цепочку команд, полученную от модуля разбора команд. На данный момент все команды у нас выполняются синхронно, поэтому в данный момент времени у нас выполняется только одна команда (в зависимости от настроек frontend'а, одновременно с командой может работать и специальный приемник данных). Поэтому контекст выполнения содержит "ссылку" (PID соответствующего процесса) на текущую команду.

Перед запуском команды на выполнение контекст выполнения проверяет, доступна ли данная команда в данном состоянии машины состояний CLI, а также хватает ли прав у текущего пользователя на выполнение этой команды. Если все проверки выполняются успешно, то контекст запускает команду на выполнение, после чего ожидает результат ее выполнения (по коду возврата). В случае успешного выполнения контекст уведомляет об этом машину состояний CLI, после чего запускает следующую команду (опять же с проверками перед запуском) и так далее, пока не будут выполнены все команды. Если выполнение одной из команд завершится с ошибкой, то контекст выполнения все последующие команды выполнять не будет за одним исключением: последнюю команду он выполняет всегда, чтобы отправить весь накопившейся вывод на сторону клиента.

*Взаимодействие клиента с севером до входа в систему и после входа в систему.*

Клиент до входа в систему (до вызова команды **login**) взаимодействует со специальным процессом сервера (backend'а). Этот процесс известен под определенным именем и обрабатывает запросы от клиентов, которые не вошли в систему. Соответственно, этот процесс обрабатывает (знает) только команду **login**. Если вход в систему был успешен, то создается процесс для работы с данным клиентом (создается клиентская сессия). Идентификатор этого процесса возвращается как результат выполнения команды **login**. Когда клиент вызывает команду **logout**, то этот процесс (клиентская сессия) уничтожается. Если клиент с активной клиентской сессией обратится к специальному процессу сервера (backend'а), то доступна ему будет только команда **login** (т.е., специальный процесс сервера не отслеживает существующие клиентские сессии).