AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA W KRAKOWIE

Wydział Informatyki, Elektroniki i Telekomunikacji
Katedra Informatyki

DOKUMENTACJA TECHNICZNA

METODY OPTYMALIZACJI WYKORZYSTYWANIA ZASOBÓW STORAGE'OWYCH Z UWZGLĘDNIENIEM WYMAGAŃ I PROFILU UŻYTKOWNIKA

MICHAŁ LISZCZ, WOJCIECH BASZCZYK

OPIEKUN:
dr inż. Włodzimierz Funika

Kraków 2015

OŚWIADCZENIE AUTORA PRACY

OŚWIADCZAM, ŚWIADOMY(-A) ODPOWIEDZIALNOŚCI KARNEJ ZA POŚWIADCZENIE NIEPRAWDY, ŻE NINIEJSZY PROJEKT WYKONAŁEM(-AM) OSOBIŚCIE I SAMODZIELNIE W ZAKRESIE OPISANYM W DALSZEJ CZĘŚCI DOKUMENTU I ŻE NIE KORZYSTAŁEM(-AM) ZE ŹRÓDEŁ INNYCH NIŻ WYMIENIONE W DALSZEJ CZĘŚCI DOKUMENTU.

. . . . . .

PODPIS

Spis treści

1. Wprowadzenie

Niniejszy tekst stanowi dokumentację techniczną opisującą szczegóły architektoniczne i implementacyjne systemu projektowanego w ramach pracy inżynierskiej.

Tematem projektu jest stworzenie systemu przechowującego i udostępniającego pliki w rozproszonej infrastrukturze. System można uruchomić na zespole heterogenicznych węzłów tworzących klaster. Wszystkie informacje o akcjach podejmowanych przez użytkowników są gromadzone, a na ich podstawie użytkownicy zostają dopasowani do różnych klas odpowiadających ich wymaganiom.

1.1. Opis problemu

Współcześnie bardzo wzrosło zapotrzebowanie na usługi przechowywania plików. Na rynku istnieje kilka rozwiązań pozwalających na przechowywanie plików. Są to między innymi Amazon S3, Riak Cloud Storage czy Ceph.

Celem projektu był stworzenie o podobnej funkcjonalności, pozwalającej użytkownikom tworzyć, czytać, zapisywać i usuwać pliki. Użytkownicy nie muszą znać dokładnego miejsca fizycznego położenia swoich danych. System powinien tak lokować pliki w poszczególnych magazynach danych (storage), aby zasoby systemowe były równomiernie i efektywnie wykorzystane.

1.2. Opis produktu

Dostarczony produkt to rozproszona aplikacja Erlang/OTP. Każdy magazyn (storage) to samodzielna instancja komunikująca się z pozostałymi magazynami. System przeznaczony jest do uruchomienia na klastrze złożonym z kilku - kilkunastu maszyn (węzłów) połączonych w sieć lokalną LAN. Możliwa jest również komunikacja poprzez sieć Internet.

Użytkownik może wykonywać operacje CRUD (Create, Read, Update, Delete) na swoich plikach korzystając z trzech interfejsów:

- biblioteki klienckiej (napisanej w Erlangu)
- RESTful API dostępnego każdym węźle przez protokół HTTP
- manualnie, korzystając z udostępnionego w przeglądarce GUI

Własne pliki można udostępniać innym użytkownikom w trybie tylko do odczytu.

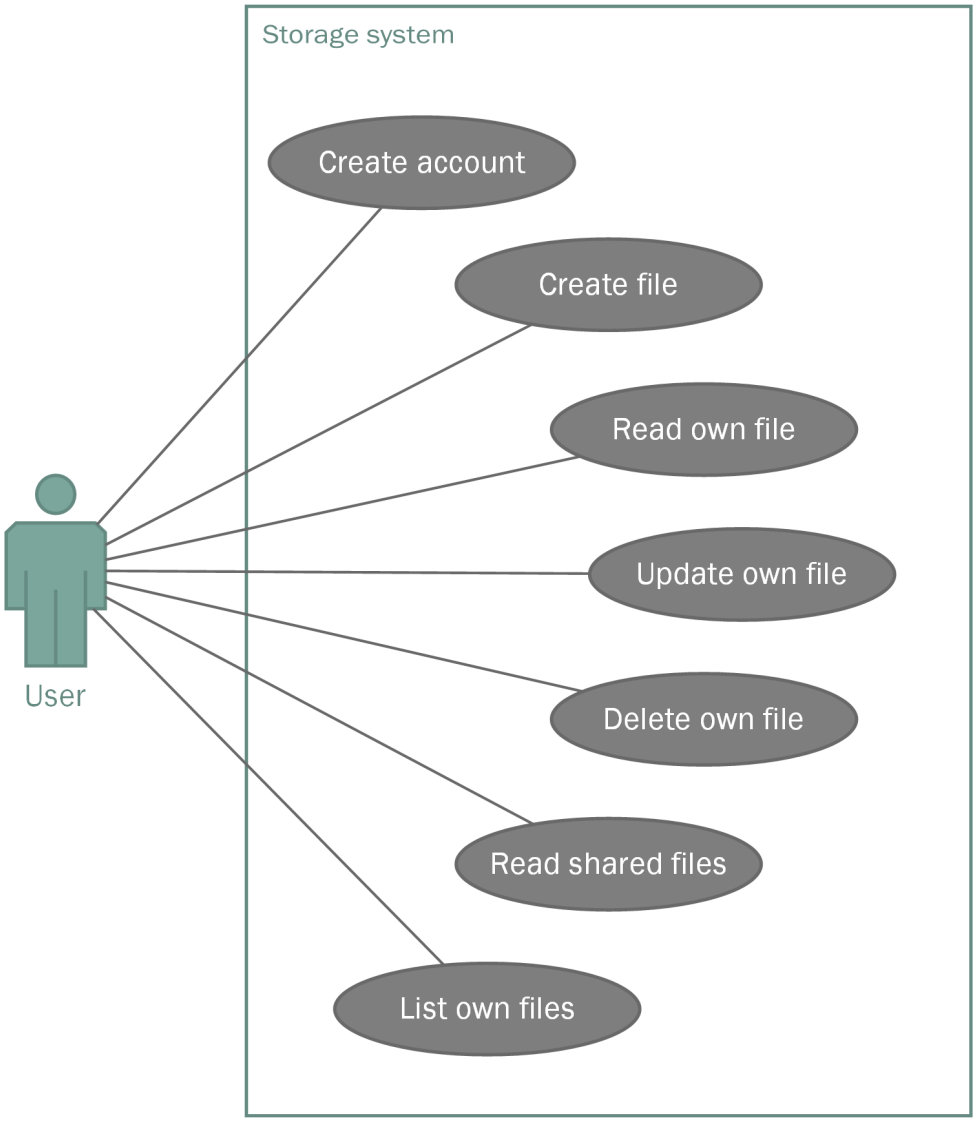
Aplikacja monitoruje wszystkie akcje, jakie podejmują użytkownicy. Są one zapisywane w bazie danych (aktualnie jest to SQLite 3). Przy ich pomocy system ustala priorytety dla poszczególnych zapytań podczas fazy schedulingu.

W razie potrzeby do systemu można dodać nowy węzeł. Nie zakłóca to jego pracy i nie wymaga zatrzymywania systemu. W przypadku wyłączenia jednego z węzłów system nadal funkcjonuje poprawnie a dane z odłączonego węzła są niedostępne do czasu jego ponownego podłączenia.

1.3. Wymagania funkcjonalne

Wymagania funkcjonalne zostały zebrane w postaci diagramu przypadków użycia przedstawionego na Rys. 1. W systemie występuje tylko jeden typ użytkownika, którego akcje są bezpośrednio obsługiwane przez system. Jest to użytkownik końcowy, który ma możliwość wykonywania podstawowych operacji na plikach.

Innym typem użytkownika jest administrator. System nie oferuje jednak funkcjonalności przydatnej w pracy administratora. Konfiguracja dobywa się poprzez pliki konfiguracyjne a za zarządzanie węzłem odpowiada mechanizm supervisorów.



Rysunek 1: Diagram przypadków użycia użytkownika systemu. Oprócz podstawowych operacji CRUD użytkownik może wylistować własne pliki i założyć konto w systemie.

1.4. Inne wymagania

Pozostałe wymagania dotyczą nie funkcjonalności systemu, a jego pożądanych cech:

- system zdecentralizowany

- rozwiązanie wieloplatformowe
- system powinien gromadzić informacje o akcjach użytkowników
- zachowanie użytkownika powinno mieć wpływ na priorytet jego zapytań

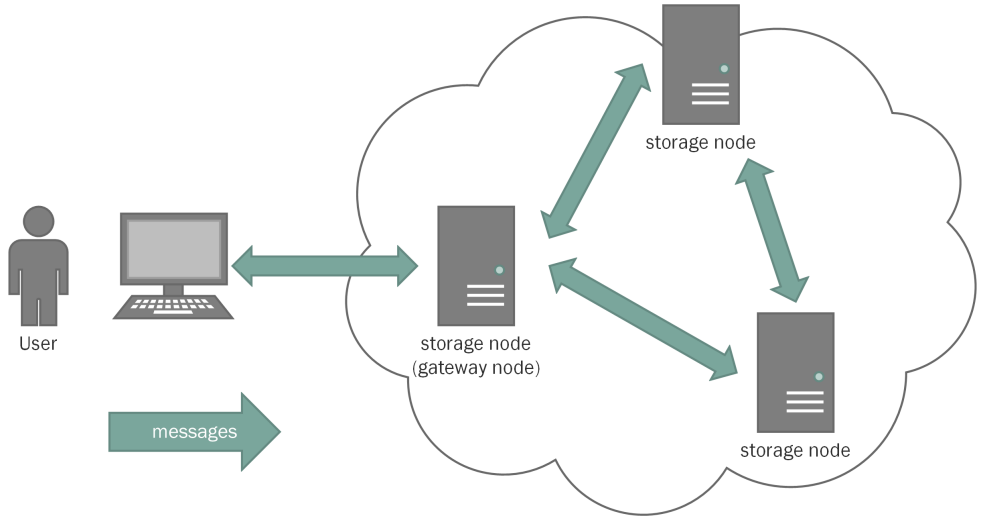
2. Architektura

Rozdział ten opisuje architekturę systemu, zaczynając od najbardziej ogólnego spojrzenia na całość i komunikaty wymieniane pomiędzy poszczególnymi węzłami, kończąc na szczegółowych opisach poszczególnych modułów.

2.1. Wstęp

Najmniejsza możliwa konfiguracja systemu to pojedynczy, samodzielnie działający węzeł. Węzeł jest aplikacją Erlang/OTP (z reguły spakowaną razem ze środowiskiem uruchomieniowym przy pomocy reltool'a). W kwestii instalacji i uruchomienia obowiązują więc standardowe procedury.

Węzły są połączone między sobą (znają swoje adresy) w sieć tworzącą graf pełny. Przedstawia to Rys. 2. Cała komunikacja między nimi oparta jest wyłącznie na komunikatach języka Erlang. Kiedy nowy węzeł dołączany jest do systemu, pobiera informacje o pozostałych węzłach od jednego z nich, a następnie rozgłasza komunikat o swoim dołączeniu.



Rysunek 2: System to zdecentralizowana sieć komunikujących się ze sobą węzłów. Komunikaty, w zależności od typu, są rozgłaszane po całym systemie lub kierowane bezpośrednio do odpowiedniego węzła.

Użytkownik może wykonywać swoje zapytania na dowolnym z węzłów. Zawsze jednak otrzyma dostęp do wszystkich swoich plików, jakie przechowuje w systemie. Wybór węzła dostępowego (gateway node) nie powinien mieć wpływu na wydajność. W związku z tym, że każdy plik może być przechowywany w innym węźle, procedura obsługi zapytań pomija sprawdzanie czy poszukiwany plik znajduje się w aktualnym węźle dostępowym.

2.2. Struktury

Spośród wszystkich wymienianych między procesami i węzłami wiadomości, dla trzech z nich zostały zdefiniowane właściwe struktury (rekordy w języku Erlang). Są to struktura użytkownika (User), metadanych pliku (File), zapytania (Request), oraz akcji użytkownika (Action).

Pozostałe wiadomości są zwykłymi krotkami zbudowanymi z typów prymitywnych oraz trzech poniżej przedstawianych struktur.

Są to również jedyne obiekty persystowane w bazie danych przy pomocy odpowiednich DAO.

Definicje rekordów znajdują się w pliku storage/include/shared.hrl.

2.2.1. User

Definicja rekordu:

```
-record(user, {
    name        :: nonempty_string(),
    secret      :: nonempty_string(),
    create_time :: integer(),
}).
```

Struktura User reprezentuje użytkownika końcowego systemu. Pola:

- name – unikalna nazwa użytkownika / login
- secret – prywatny klucz użytkownika używany przy autentykacji
- create_time – data (timestamp) utworzenia konta użytkownika

2.2.2. File

Definicja rekordu:

```
-record(file, {
    owner       :: nonempty_string(),
    vpath       :: nonempty_string(),
    bytes       :: integer(),
    access_mode :: integer(),
    create_time :: integer()
}).
```

Struktura File gromadzi metadane dotyczące pliku w systemie. Pola:

- owner – identyfikator użytkownika, właściciela pliku
- vpath – ścieżka do pliku (UNIX-style). Razem z owner tworzą klucz
- główny listy plików
- bytes – rozmiar pliku (w bajtach)
- access_mode – flagi dostępu do pliku
- create_time – data (timestamp) utworzenia pliku

2.2.3. Request

Definicja rekordu:

```
-record(request, {
    type                  :: 'create' | 'read' | 'update'
                           | 'delete' | 'list' | 'find',
    user                  :: nonempty_string(),
    addr = {none, none}   :: { nonempty_string(),
                               nonempty_string() },
    hmac = none           :: string(),
    data = none           :: binary(),
    opts = none           :: term()
}).
```

Struktura Request reprezentuje żądanie operacji na pliku. Jest to również podstawowy typ wiadomości w systemie. Przekazywane jest zarówno pomiędzy węzłami jak i wewnątrz węzłów między poszczególnymi modułami. Pola:

- type – atom reprezentujący jeden z typów operacji
- user – identyfikator użytkownika wykonującego zapytanie
- addr - krotka reprezentująca adres żądanego pliku, w postaci {owner, vpath}
- hmac – suma kontrolna HMAC (więcej w rozdziale dotyczącym autentykacji)
- data – binarne dane (w przypadku tworzenia / aktualizacji pliku)
- opts – obecnie nie używane

2.2.4. Action

Definicja rekordu:

```
-record(action, {
    user_id          :: nonempty_string(),
    file_id          ,
    weight           :: integer(),
    action_time      :: integer(),
    action_type      :: nonempty_string()
}).
```

Struktura Action tworzona jest przy przetwarzaniu zapytania Request a następnie zapisywana w bazie danych. Lista takich struktur daje opis historii operacji danego użytkownika. Pola:

- user_id – identyfikator użytkownika
- file_id – adres pliku

- weight – waga przypisana żądaniu podczas schedulingu
- action_time – data (timestamp) obsługi żądania
- action_type – identycznie jak request#type, tylko w postaci string()

2.3. Storage

Pojedynczy węzeł systemu jest aplikacją Erlang/OTP. Składa się z jednego, głównego supervisora zarządzającego pięcioma gen_serverami: storage_http_srv, storage_auth_srv, storage_dist_srv, storage_core_srv oraz storage_uuid_srv. storage_http_srv jest opcjonalny a jego obecność nie jest wymagana do poprawnej pracy całego systemu. Supervisor pracuje w polityce one-for-one, restartując poszczególne komponenty w razie awarii. Struktura serwerów przedstawiona jest na Rys. 3.

Każdy z gen_serverów ma jasno określone zadania. Poszczególne serwery (moduły) ściśle współpracują między sobą i wymieniają wiadomości w celu obsługi zapytania. Na podstawie przepływu informacji między nimi można wyróżnić warstwową hierarchię przedstawioną na Rys. 4. Zapytanie przekazywane jest między kolejnymi modułami:

1. storage_http_srv (opcjonalnie)
2. storage_auth_srv
3. storage_dist_srv
4. storage_core_srv

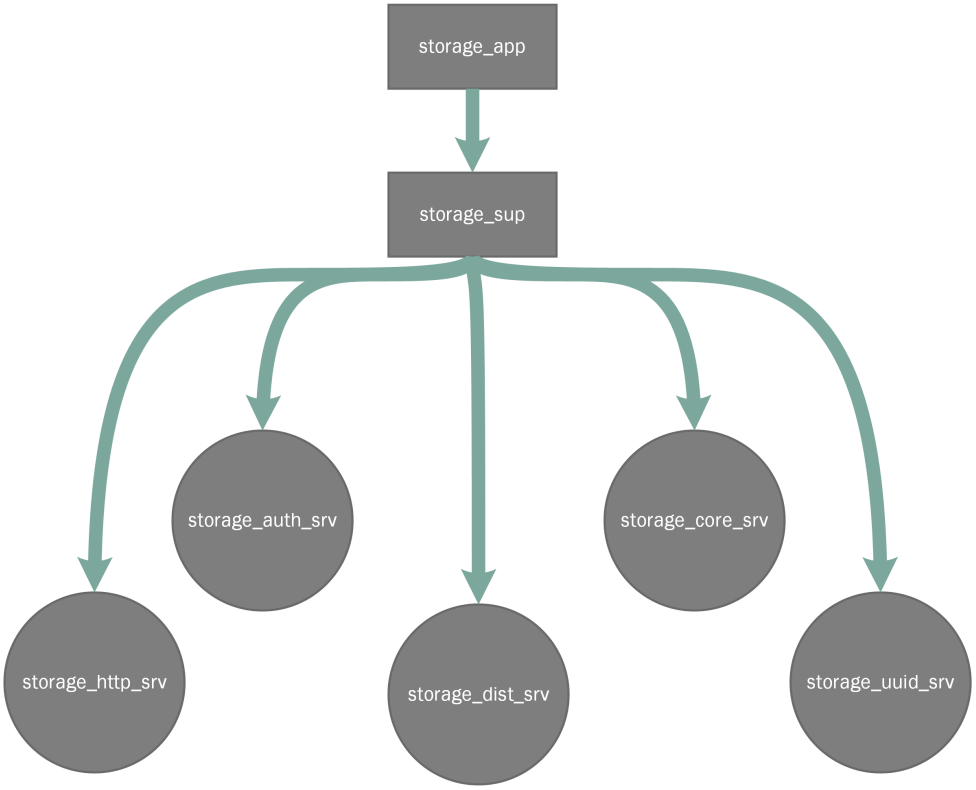
Użycie modułu storage_http_srv jest opcjonalne – użytkownik może również skorzystać z biblioteki klienckiej (napisanej w Erlangu) i pominąć wysyłanie zapytania przez protokół HTTP.

Szczegółowy opis kolejnych faz obsługi zapytania znajduje się w podrozdziale dotyczącym komunikacji wysokopoziomowej.

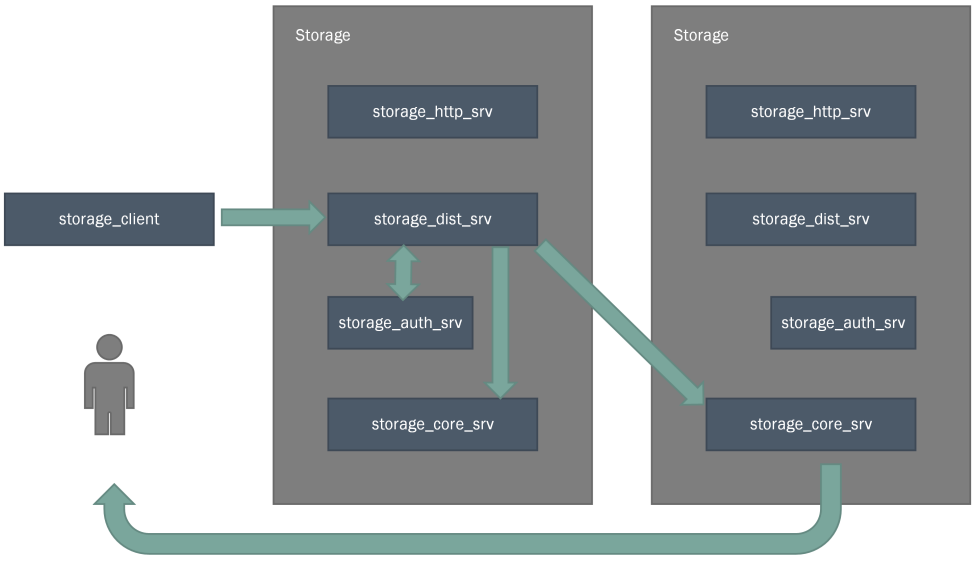
2.3.1. Komunikacja wysokopoziomowa

Komunikacja pomiędzy węzłami oparta jest w całości na przesyłaniu struktur Request. W zależności od typu zapytania, w procedurze obsługi występują nieznaczne różnice. Można jednak przedstawić to w postaci listy modułów, gdzie kolejno trafia zapytanie:

1. (opcjonalnie) storage_http_srv – zapytanie jest parsowane, tworzona jest struktura Request
2. storage_dist_srv – przyjmuje strukturę Request, przekazuje do autentykacji
3. storage_auth_srv – autentykuje i sprawdza spójność przekazanego zapytania
4. storage_dist_srv
   (a) zapytania create / update – wyszukuje odpowiedni węzeł i przekazuje strukturę Request do działającego na nim storage_core_srv



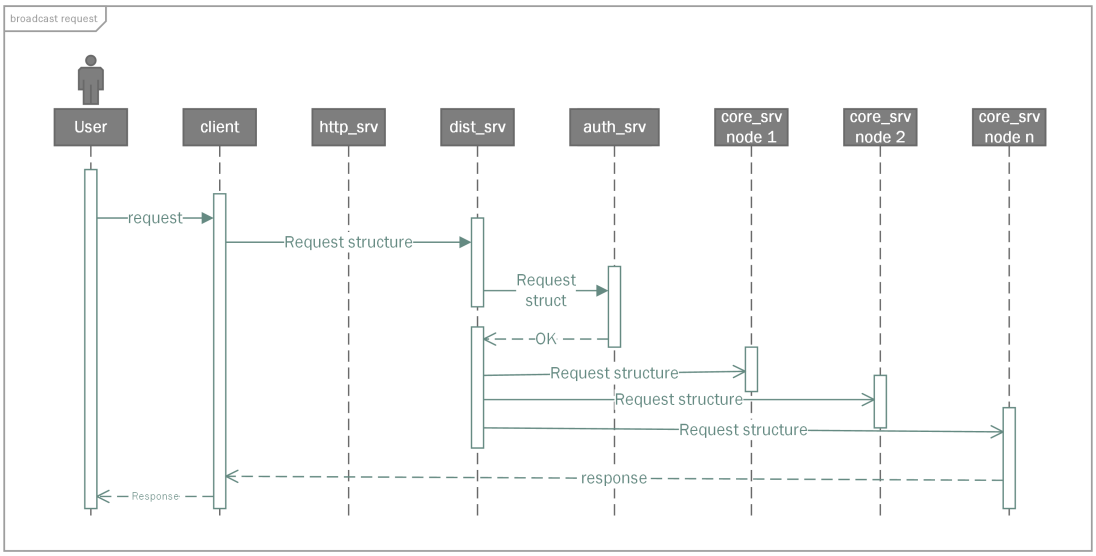
Rysunek 3: Supervision tree. Aplikacja zbudowana jest z pięciu gen_serverów. Każdy jest osobnym procesem. Ponadto storage_core_srv zarządza pulą procesów wykonawczych.



Rysunek 4: Przepływ zapytania w systemie. Zapytanie jest rozgłaszane do wszystkich węzłów. Zielone linie oznaczają przepływ zapytania.

(b) pozostałe zapytania – rozgłasza strukturę Request do serwerów storge_core_srv na wszystkich węzłach w systemie

5. storage_core_srv – dokonuje obsługi zapytania lub odrzuca je, jeżeli dotyczy pliku który nie znajduje się w danym węźle. Odpowiedź kierowana jest prosto do użytkownika



Rysunek 5: Zapytanie rozgłoszeniowe z wykorzystaniem biblioteki klienckiej zakończone powodzeniem.

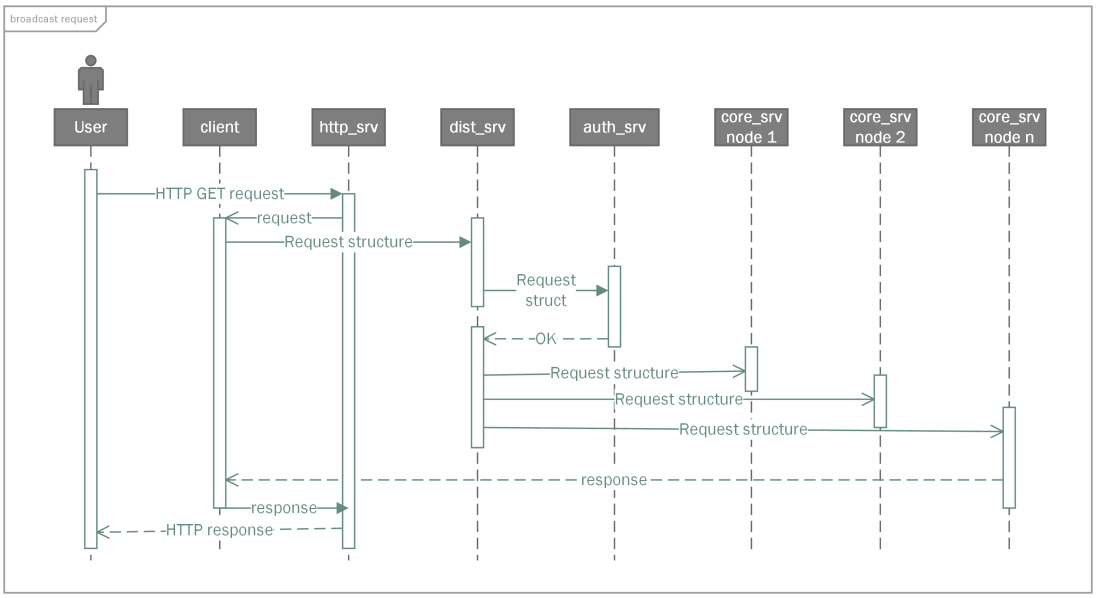
Rys. 5 pokazuje diagram sekwencji obsługi zapytań rozgłoszeniowych z wykorzystaniem biblioteki klienckiej. Zapytania tego typu to zapytania read, delete oraz find. Rys. 6 pokazuje te same akcje przy wykorzystaniu modułu HTTP. Widać, że moduł HTTP wewnętrznie korzysta z biblioteki klienckiej. Rysunki zakładają, że autentykacja przebiegła pomyślnie a jeden z węzłów zawierał żądany plik. Użytkownikowi odpowiada tylko jeden węzeł - ten, który obsłużył zapytanie (przechowywał poszukiwany plik). Pozostałe węzły ignorują komunikat.

Zapytania typu create i update nie są rozgłaszane. Najbardziej odpowiedni na przyjęcie nowych danych węzeł jest wybierany spośród wszystkich węzłów w systemie. Wybór ten dokonywany jest w węźle, do którego początkowo trafia zapytanie (gateway node). Jedyną różnicą w stosunku do Rys. 5 oraz Rys. 6 byłoby zaznaczenie pojedynczego modułu core_srv, działającego na docelowym węźle.

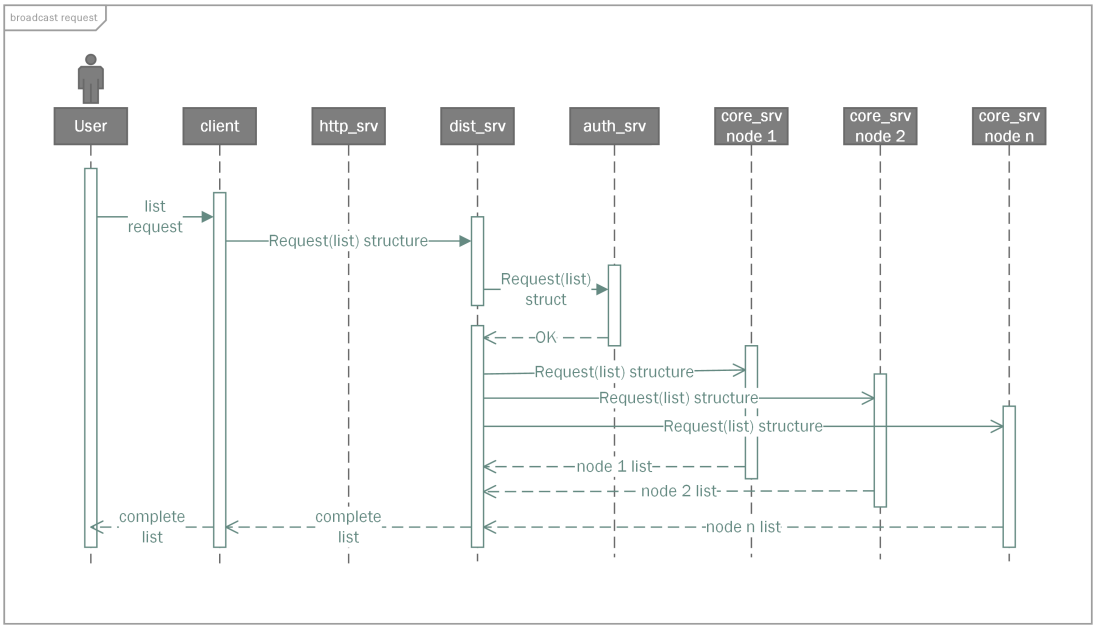
W przypadku zapytania o listę wszystkich plików (zapytanie list), przeszukane muszą zostać wszystkie węzły. Odpowiedź nie wraca jednak z każdego z nich bezpośrednio do klienta, lecz do modułu dist_srv, odpowiedzialnego za połączenie wszystkich list w jedną listę wynikową. Sytuację tę przedstawia diagram na Rys. 7.

2.3.2. Obsługa błędnych zapytań

Istnieją sytuacje, kiedy żądanie przesłane do systemu kończy się niepowodzeniem – kiedy nie udało się znaleźć pliku który użytkownik chciał odczytać lub w systemie nie ma miejsca na stworzenie nowego pliku. Można rozróżnić tutaj dwa scenariusze: system od razu sygnalizuje błąd oraz system 'zawiesza się', w oczekiwaniu na odpowiedź jednego z węzłów.

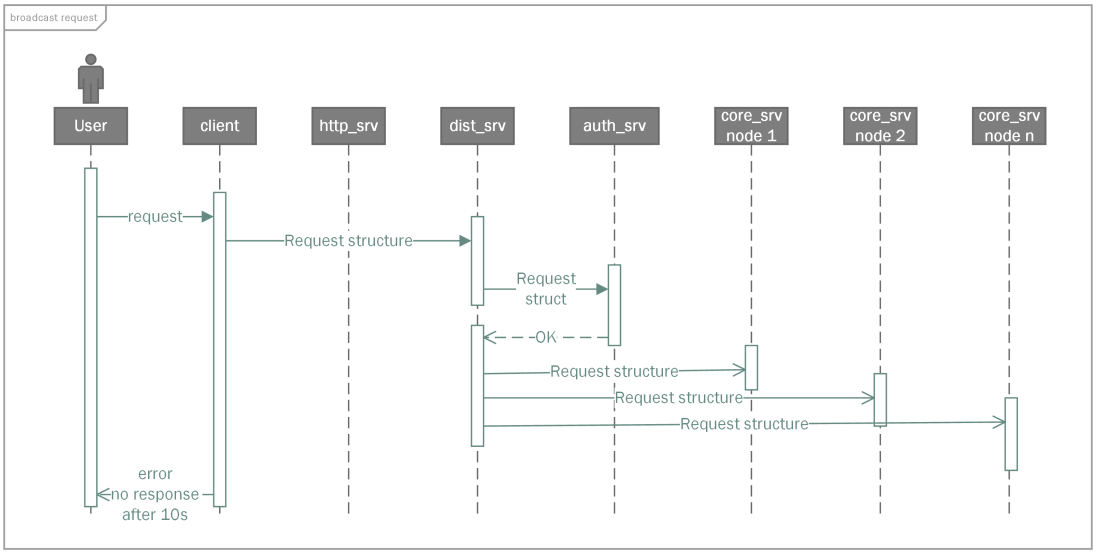


Rysunek 6: Zapytanie rozgłoszeniowe z wykorzystaniem modułu HTTP. Użytkownik wysyła zapytanie metodą GET, które tłumaczone jest na odpowiednią strukturę Request.



Rysunek 7: Obsługa zapytania o listę wszystkich plików użytkownika.

Pierwszy ma miejsce w przypadki zapytań create i update, kiedy moduł storage_dist_srv zwraca do biblioteki klienckiej błąd o niemożliwości zapisania pliku. Wszystkie pozostałe zapytania (zapytania rozgłoszeniowe) mają ustalony pewien czas (domyślnie 10 sekund), po którym jeżeli biblioteka kliencka nie otrzyma odpowiedzi, zgłasza błąd. Taki scenariusz przedstawia Rys. 8.



Rysunek 8: Zapytanie rozgłoszeniowe zakończone niepowodzeniem. Jeżeli biblioteka kliencka nie otrzyma przez pewien czas odpowiedzi z żadnego z węzłów, zakłada się że poszukiwany plik nie istnieje (węzeł który nie posiada szukanego pliku ignoruje zapytania).

2.3.3. Moduł HTTP

Struktura:

- storage/src/http/storage_http_srv.erl – gen_server
- storage/src/http/http_utils.erl – funkcje pomocnicze

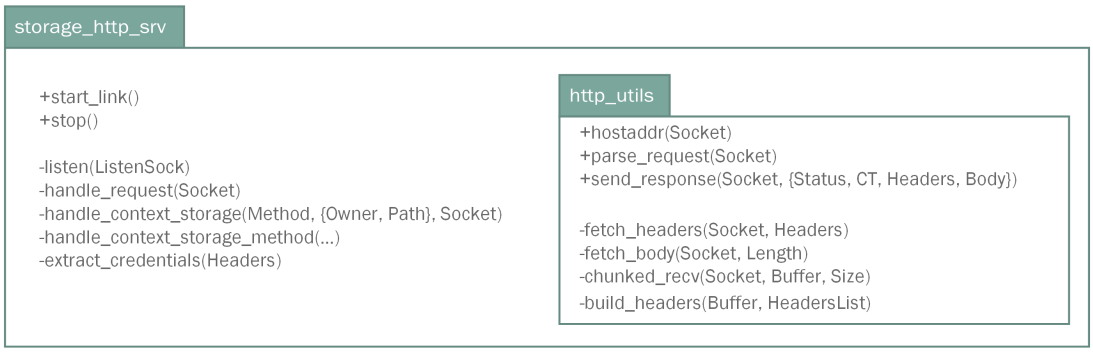
Moduł służy do przetwarzania zapytań HTTP i generowania odpowiedzi. Potrafi również serwować statyczne pliki HTML – przykładowo graficzny menedżer plików.

Nie definiuje żadnego publicznego API oraz ignoruje wszystkie komunikaty pochodzące z handle_call i handle_cast. Jedyne dwie funkcje to standardowe start_link() oraz stop(). Cała komunikacja z modułem dobywa się poprzez socket akceptujący połączenia na porcie określonym w pliku konfiguracyjnym.

Moduł http_utils to zbiór funkcji pomocniczych, odpowiedzialnych za parsowanie zapytań i konstruowanie odpowiedzi HTTP. Oba moduły przedstawiaRys. 9.

Budowa przykładowego zapytania, jakie można wysłać do modułu (standardowy *HTTP Request*):

```
GET storage/user02/path/to/my/file.dat HTTP/1.1
Host: ds-01.storage.example.com:9001
Authorization: HMAC user01:6c51adc384572536d9c8a9dbcfbebf590942771f
```



Rysunek 9: Struktura modułu HTTP.

Zostanie przetłumaczone na poniższą strukturę Request:

```
-record(request, {
    type        = 'read',
    user        = "user01"
    addr        = {"user02", "␣path/to/my/file.dat},
␣␣␣␣hmac␣␣␣␣␣␣␣␣=␣"6c51adc384572536d9c8a9dbcfbebf590942771f",
␣␣␣␣data␣␣␣␣␣␣␣␣=␣none,
␣␣␣␣opts␣␣␣␣␣␣␣␣=␣none
}).
```

Jeżeli plik zostanie znaleziony a użytkownik będzie mógł go odczytać, serwer zwróci odpowiedź HTTP 200 OK, a w treści znajdzie się zawartość binarna pliku.

Diagram sekwencji obsługi przykładowego zapytania typu GET (odczyt pliku) przedstawia Rys. 10.

2.3.4. Moduł autentykacji

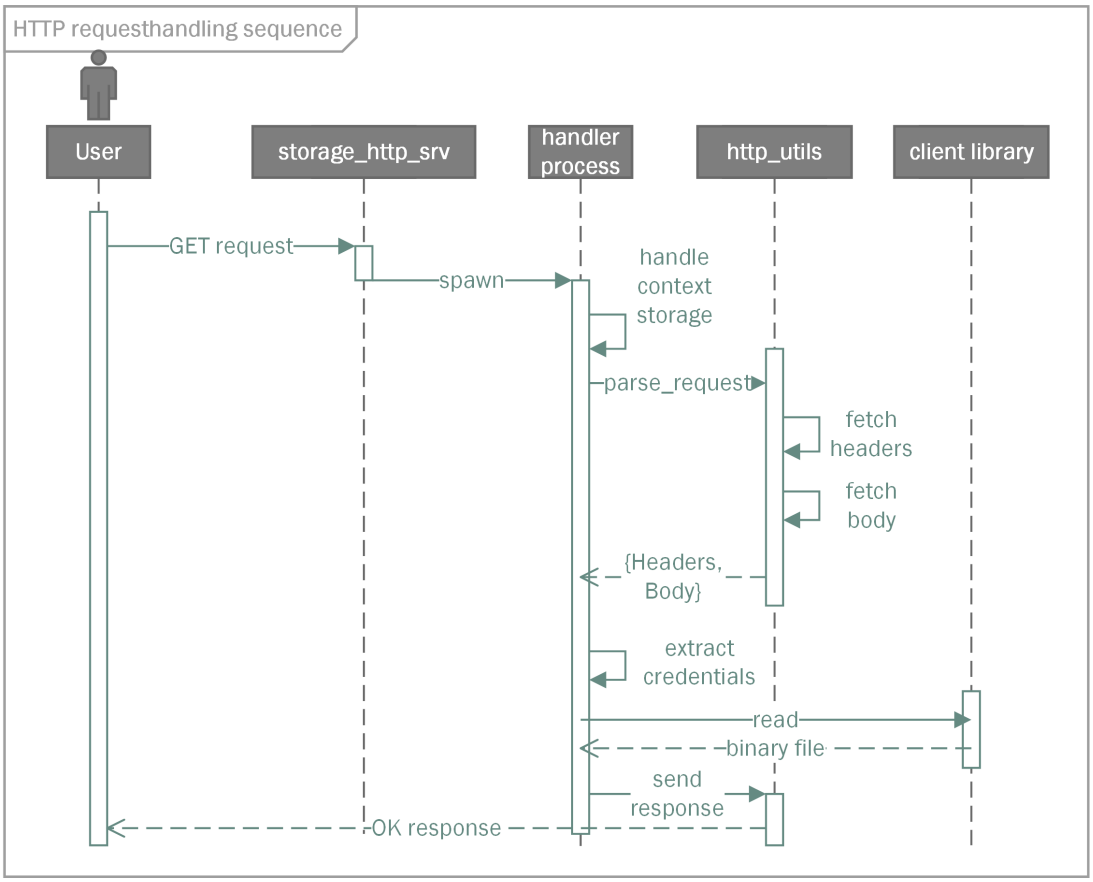
Struktura:

- storage/src/auth/storage_auth_srv.erl – gen_server
- storage/src/auth/db_users.erl – DAO dla struktur User

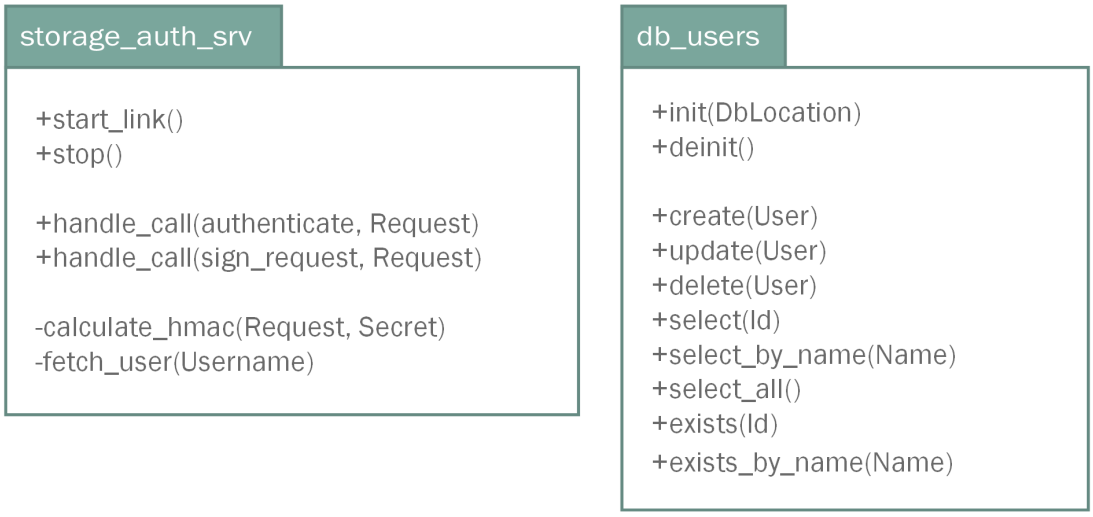
Moduł autentykacji odpowiada za obliczanie i weryfikację sumy kontrolnej HMAC przekazywanych zapytań. Moduł db_users jest modułem dostępowym do encji reprezentujących użytkownika (struktury User), przechowywanych w bazie danych. Suma HMAC opisana jest dokładniej w kolejnym punkcie.

Serwer storage_auth_srv przechowuje tablicę ets zawierającą krotki Username, Secret, Expires, pełniącą rolę pamięci podręcznej. Znajdują się w niej użytkownicy wraz z ich kluczami prywatnymi. Opis aktualizacji tej tablicy znajduje się w rozdziale Synchronizacja użytkowników.

Suma HMAC: keyed-Hash Message Authentication Code to funkcja skrótu z dodatkowo wmieszanym kluczem prywatnym. Wynikowy kod zależy nie tylko od danych z których jest



Rysunek 10: Przetwarzanie zapytania HTTP w module storage_http_srv. Dla każdego zapytania uruchamiany jest osobny wątek odpowiedzialny za jego obsługę, który parsuje zapytanie, wykonuje żądaną akcję przy użyciu biblioteki klienckiej a rezultat zwraca z powrotem przy użyciu protokołu HTTP.



Rysunek 11: Moduły składające się na moduł autentykacji. storage_auth_srv pozwala na porównanie sumy kontrolnej otrzymanego zapytania z wartością, którą sam wylicza. db_users oferuje podstawowe operacje na bazie użytkowników, z wykorzystaniem struktury User.

obliczany, ale również od wykorzystanego hasła. Hasło (klucz prywatny) znane jest tylko użytkownikowi oraz systemowi. Nie jest przesyłane między nimi, co zmniejsza szanse na ujawnienie klucza.

Wykorzystanie HMAC zapewnia ochronę integralności przesyłanych zapytań (modyfikacja parametrów zapytania wymaga przeliczenia sumy HMAC) oraz autentyczności danych (wyliczenie sumy wymaga znajomości klucza prywatnego).

Każdy użytkownik posiada "hasło". Może on je ustalić dowolnie ze swoimi preferencjami. Kluczem prywatnym staje się suma SHA1 obliczona z tego hasła, i tylko ta wartość przechowywana jest w bazie danych.

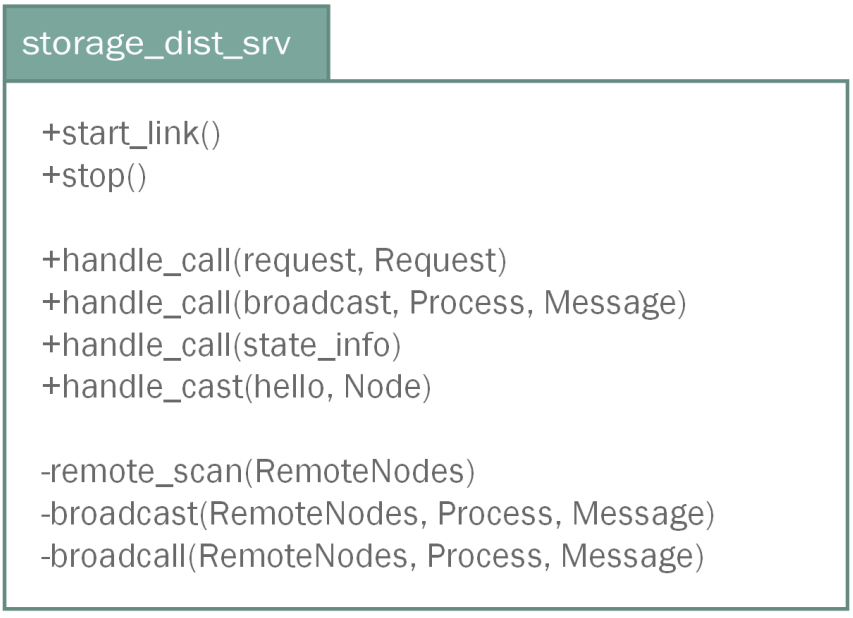
2.3.5. Moduł komunikacyjny

Struktura:

- storage/src/dist/storage_dist_srv.erl – gen_server

Jest to moduł odpowiedzialny za komunikację między węzłami systemu. Zapytania generowane przez bibliotekę kliencką trafiają właśnie tutaj. Są autentykowane a następnie przesyłane dalej zgodnie z polityką obsługi danego typu zapytania.

gen_server w swoim stanie przechowuje zbiór adresów wszystkich węzłów w systemie (rezultat wywołania node() na każdym węźle). Standardowy protokół tworzenia i aktualizacji tego zbioru opisuje rozdział Dołączanie węzłów (synchronizacja stanu).

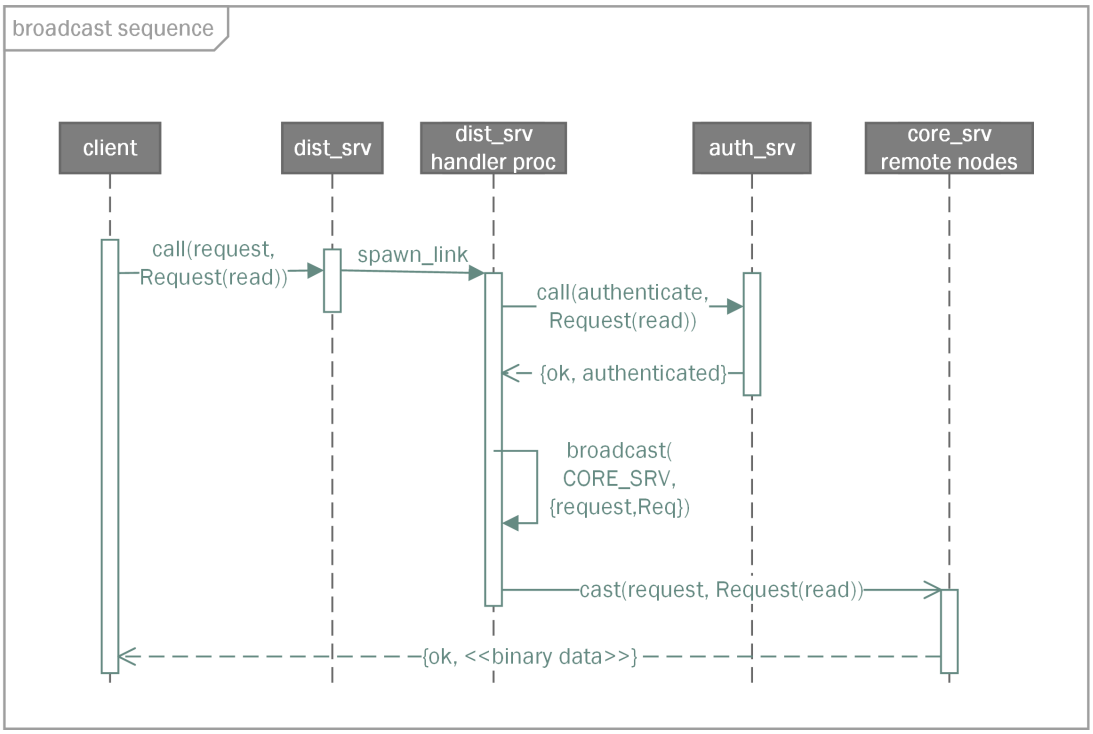


Rysunek 12: Publiczny interfejs modułu storage_dist_srv. Podstawowa metoda to handle_call(request, Request), która jest zdalnie wywoływana przez API klienckie z odpowiednią strukturą zapytania jako argument.

Obsługa zapytań read, delete, find Wszystkie te zapytania obsługiwane są w identyczny sposób. Przykładowy ciąg sekwencji przy obsłudze zapytania typu read (rozsyłanego do wszystkich węzłów) przedstawia Rys. 13.

Źródłem zapytania jest proces używający biblioteki klienckiej (client), wywołujący synchroniczne zapytanie na module storage_dist_srv. Od razu tworzony jest nowy proces odpowiedzialny za obsługę tego żądania. Główny proces gen_servera zwraca z handle_call status noreply i kontynuuje działanie. Wygenerowanie odpowiedzi i przesłanie jej do użytkownika będzie należało do jednego z modułów storage_core_srv, gdzie przesłana jest odpowiednia struktura Request.

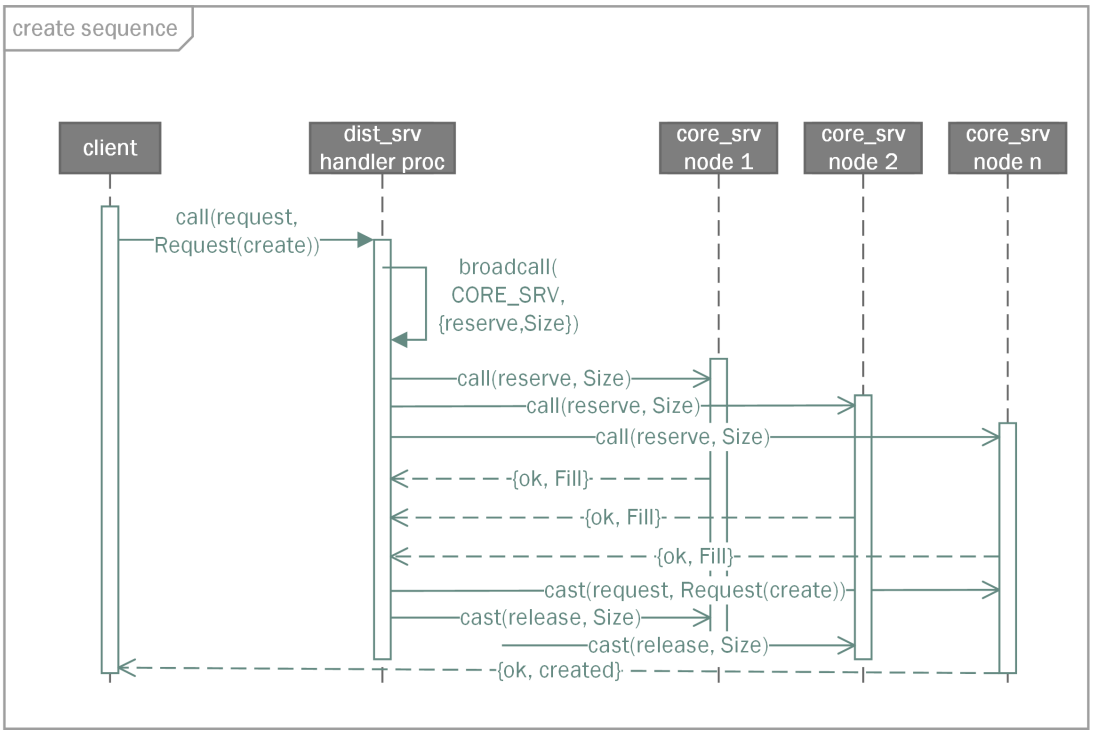
Proces obsługi zapytania autentykuje je przy użyciu modułu storage_auth_srv zaraz po jego otrzymaniu.



Rysunek 13: Sekwencja wywołań przy obsłudze zapytania read w module storage_dist_srv.

Obsługa zapytań create Zapytanie create różni się od obsługi zapytań read, delete czy find. Wymaga zlokalizowania węzła o najmniejszym zapełnieniu, który może pomieścić dany plik. Do modułów storage_core_srv na wszystkich węzłach w systemie wysyłana jest wiadomość reserve, Size, gdzie Size jest rozmiarem pliku. Jeżeli dany węzeł może przyjąć plik, odpowiada komunikatem ok, Fill, gdzie Fill jest jego procentowym zapełnieniem. Do niego następnie przesyłane jest właściwe żądanie użytkownika. Do pozostałych węzłów trafia komunikat release, Size, mówiący, że mogą zwolnić zarezerwowane zasoby dyskowe. Sekwencja wywołań przedstawiona jest na Rys. 14.

Obsługa zapytań update Zapytanie update przekazywane jest bezpośrednio do węzła który przechowuje aktualizowany plik. Implementacja wykorzystuje dodatkowe zapytanie find, w celu zlokalizowania tego węzła. Sekwencja jest więc bardzo podobna do tej przedstawionej na Rys. 13.



Rysunek 14: Obsługa zapytania create. Następuje rezerwacja miejsca na wszystkich węzłach, wybierany jest najbardziej odpowiedni do przechowania pliku, a zarezerwowane miejsce jest zwalniane.

2.3.6. Moduł wykonawczy

Struktura:

- storage/src/core/storage_core_srv.erl – gen_server
- storage/src/core/db_files.erl – DAO dla struktur File
- storage/src/core/db_actions.erl – DAO dla struktur Action
- storage/src/core/core.erl – implementacja operacji plikowych
- storage/src/core/files.erl – funkcje I/O
- storage/src/core/scheduler.erl – scheduler i wątki wykonawcze

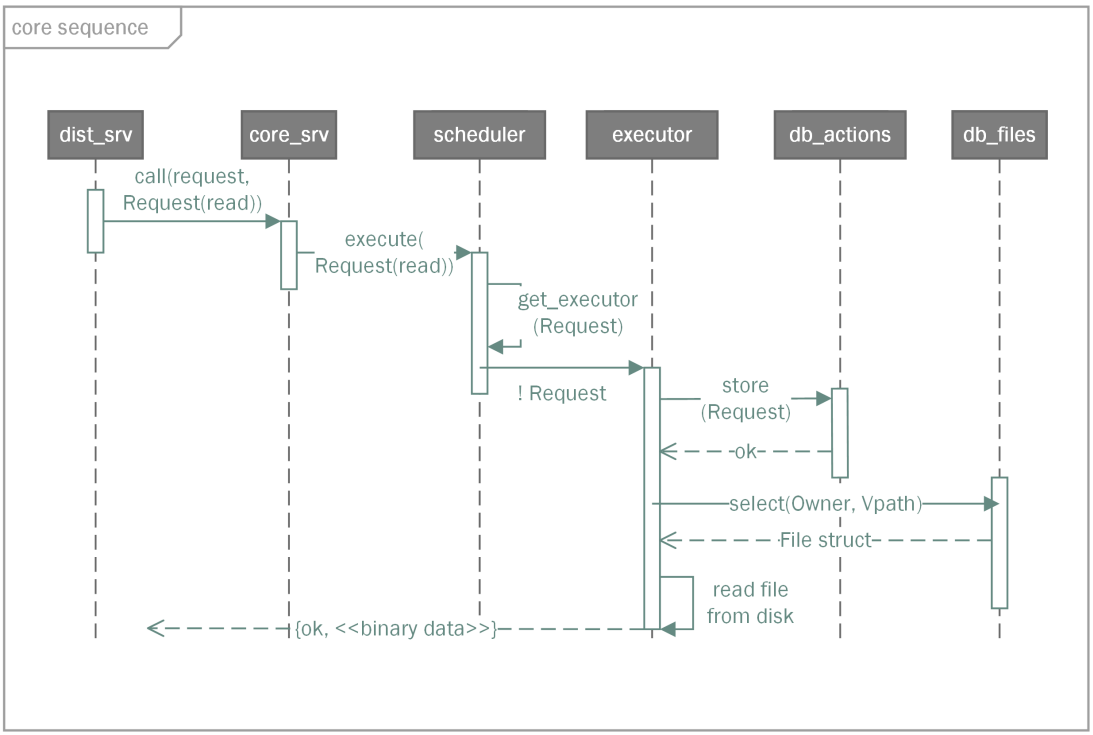
Moduł core to główny moduł systemu, odpowiedzialny za realizację wszystkich zleconych operacji. Komunikuje się bezpośrednio z bazą danych i systemem plików. Tutaj trafiają wszystkie zapytania w finalnej fazie obsługi. Odpowiedzi kierowane są zwykle wprost do użytkownika.

Składowe moduł i zawartość każdego z nich przedstawia diagram pokazany na Rys. 15.

Każdy węzeł dysponuje określoną ilością miejsca na przechowywane pliki. Kiedy tworzone jest nowy plik, dostępne miejsce maleje. W punkcie 1.3.5 Moduł komunikacyjny przedstawiono zasady działania wywołań call(reserve, Size) i call(release, Size). Trzecia z funkcji gen_servera, call(request, Request) odpowiedzialna jest za obsługę zapytania i przyjmuje w argumencie

Rysunek 15: Podmoduły składowe modułu core.

strukturę Request. Zapytanie jest natychmiast przekazywane do modułu schedulera, który wyszukuje odpowiedni wątek wykonawczy i przekazuje mu zapytanie do obsługi. Wątek ten jest również odpowiedzialny za umieszczenie w bazie informacji o wykonanej akcji. Ogólny diagram sekwencji przedstawia Rys. 16. Rozważany przypadek to zapytanie read. Wszystkie inne wyglądają jednak tak samo. Scheduler został ukazany jako jeden, atomowy obiekt. Dalej opisany jest w szczegółach.



Rysunek 16: Obsługa żądania odczytu pliku w module wykonawczym.

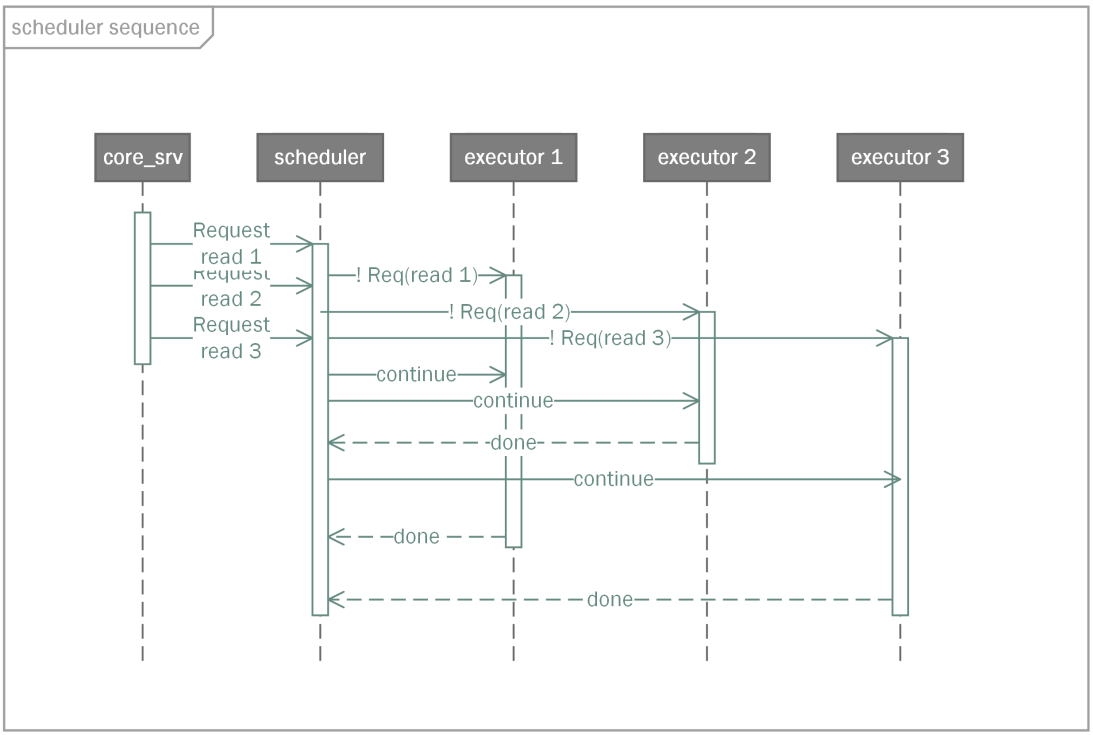
Scheduler Scheduler jest modułem szeregującym zapytania trafiające do danego węzła. Zarządza on pulą wątków wykonawczych (executor), odpowiedzialnych za bezpośrednią obsługę zadań. Każdy plik (każda ścieżka) ma przydzielony jeden wątek wykonawczy. Zapytania dotyczące konkretnego pliku zawsze więc są obsługiwane przez jeden wątek. Wątek wykonawczy tworzony jest po próbie dostępu do określonego pliku i ulega zniszczeniu po określonym czasie bezczynności (domyślnie 180 sekund).

Po otrzymaniu zapytania o konkretny plik, scheduler znajduje skojarzony z nim wątek wykonawczy i umieszcza zadanie (strukturę Request) w kolejce wiadomości tego procesu (message queue języka Erlang).

Scheduler umieszcza wszystkie przychodzące akcje w kolejce priorytetowej. Zapisuje je w postaci krotek {Priority, Executor}, gdzie Priority to priorytet obliczony dla danego zapytania natomiast Executor to identyfikator wątku wykonawczego gdzie trafiło to zapytanie.

Przed rozpoczęciem przetwarzania kolejnego zadania, wątek wykonawczy czeka na pozwolenie od schedulera. Po każdym zakończonym zadaniu informuje scheduler o zakończeniu jego obsługi.

Scheduler uruchamia N (domyślnie 4) pierwszych wątków wykonawczych z kolejki priorytetowej. Jeżeli jakiś wątek skończy przetwarzać jakieś zapytanie, scheduler wybiera z kolejki w jego miejsce wątek o najwyższym priorytecie.



Rysunek 17: Sekwencja uruchamiania wątków wykonawczych (maksymalnie dwa jednoczesne wątki). Pojawiają się trzy zapytania o trzy różne pliki. Każde z nich trafi zatem do innego wątku wykonawczego. Zadania są kolejkowane w wątkach zgodnie z kolejnością przybycia (kolejność w obrębie jednego wątku wykonawczego musi zostać zachowana). Wątki nie rozpoczynają pracy od razu – scheduler wysyła do każdego z nich komunikat continue. Dwa wątki zostały uruchomione jednocześnie. Trzeci czekał na zakończenie pracy przez jeden z nich.

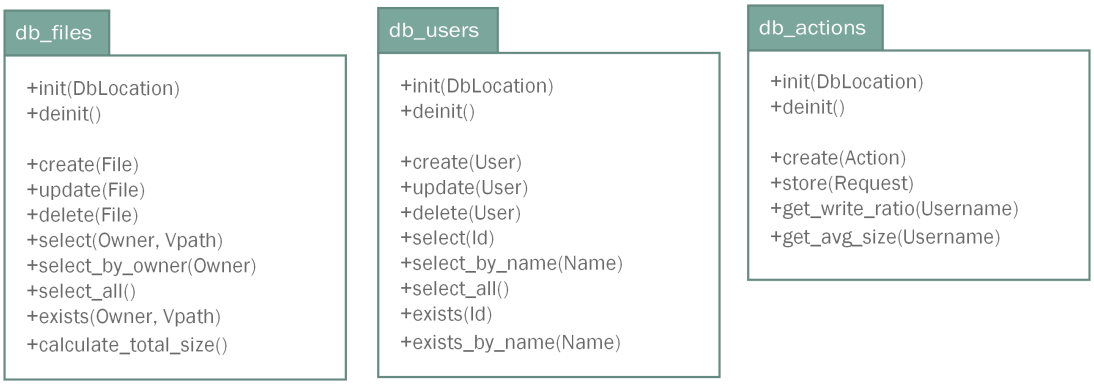
2.3.7. Baza danych

System korzysta z bazy danych SQLite3. W celu komunikacji z bazą danych z poziomu Erlanga wybrano erlang-sqlite3 (`https://github.com/alexeyr/erlang-sqlite3`). Persystowane są trzy rodzaje struktur: User, File oraz Action. Opisuje je rozdział 1.2 Struktury. Dla każdej z nich zdefiniowany jest odpowiedni moduł DAO: db_users, db_files oraz db_actions. Wszystkie zostały przedstawione w poprzednich rozdziałach przy omawianiu modułów nadrzędnych.

Moduły DAO znajdują się w plikach:

- storage/src/core/db_files.erl
- storage/src/auth/db_users.erl
- storage/src/core/db_actions.erl

W przypadku chęci zmiany bazy danych, zmian wystarczy dokonać tylko w tych plikach (oraz dostarczyć odpowiedni sterownik). Jeden z prototypów działał z wykorzystaniem bazy MySQL.



Rysunek 18: Moduły DAO pozwalają zapisywać obiekty w bazie danych i wyszukiwać je na podstawie identyfikatora.

Rys. 19 przedstawia diagram ERD bazy danych. Struktura tabel jest bardzo prosta jednak pozwala zrealizować wszystkie wymagania projektu.

Baza może pracować w standardowo, zapisując rekordy na dysku oraz w konfiguracji in-memory. Druga opcja przydatna może być w testach wydajnościowych. W celu jej aktywacji należy skompilować projekt z odpowiednią flagą. Więcej na temat kompilacji mówi rozdział 3.3 Kompilacja.

Rysunek 19: Baza danych - diagram ERD

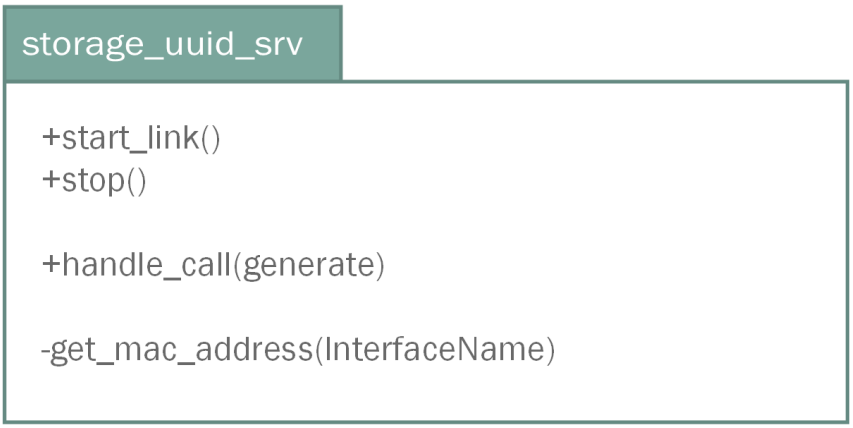
2.3.8. Generator UUID

Struktura:

- storage/src/shared/storage_uuid_srv.erl – gen_server

Jest to wielowątkowy moduł służący do generowania unikalnych, 128-bitowych identyfikatorów. Identyfikatory te są wykorzystywane jako nazwy fizycznych plików przechowywanych w systemie dyskowym. Właściwymi identyfikatorami plików i użytkowników w systemie pozostają jednak zwykłe napisy, w postaci czytelnej dla człowieka.

Strukturę modułu przedstawia diagram z Rys. 20. Jedyna publiczna funkcja służy do generowania identyfikatora. Dla każdego wywołania uruchamiany jest osobny wątek. Identyfikator zwracany jest w postaci typu binary().



Rysunek 20: Interfejs modułu storage_uuid_srv.

Identyfikator składa się z trzech części – grup bitów, z których każda ma odpowiednią interpretację:

`« 48bit timestamp, 48bit adres MAC, 32bit losowa wartość »`

48-bitowy znacznik czasowy przechowuje czas wygenerowania identyfikatora (UNIX time) z dokładnością do milisekund. Adres MAC pobierany jest z określonego w pliku konfiguracyjnym interfejsu sieciowego (domyślnie eth0). Zastosowanie składnika w postaci MAC gwarantuje, że generowane identyfikatory są unikalne w obrębie całego systemu. Ostatni składnik identyfikatora to 32 losowe bity. Daje to 4 294 967 296 możliwych do wygenerowania unikalnych identyfikatorów w ciągu milisekundy.

2.3.9. Logger

Struktura:

- storage/src/shared/log.erl – moduł

Moduł loggera. Wypisuje komunikaty na standardowe wyjście. Węzeł zbudowany w trybie release nie posiada uruchomionej konsoli. Wtedy logi trafiają w domyślne miejsce: log/erlang.log.1

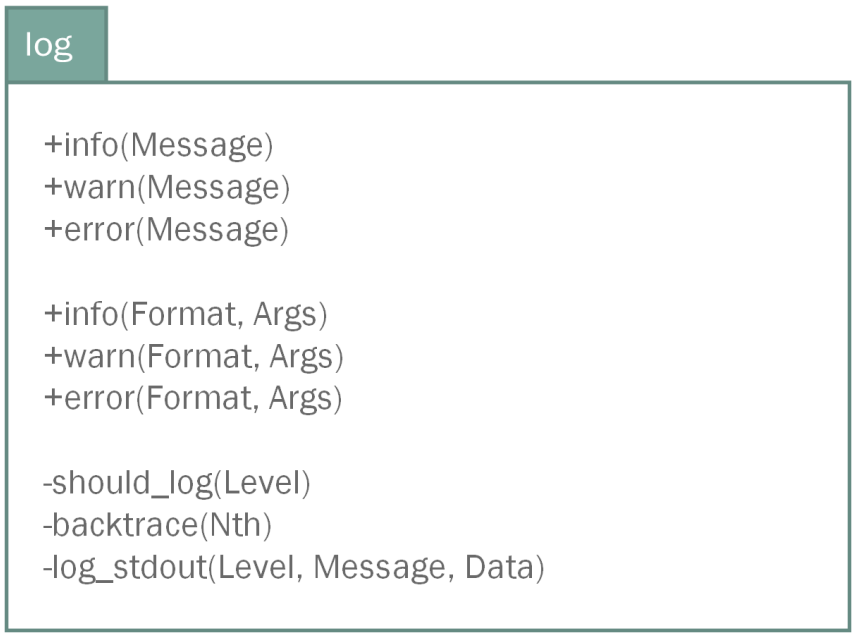
Interfejs loggera przedstawia Rys. 21. Dostępne są trzy funkcje logujące informacje: log:info, log:warn oraz log:error. Pozwalają logować informacje różnych typów. O tym, czy dana funkcja wypisze komunikat na ekran decyduje poziom logowania ustawiony w konfiguracji aplikacji. Szczegóły przedstawia rozdział 3.5 Pliki konfiguracyjne. Możliwe poziomy logowania:

- info: wszystkie funkcje wypisują komunkaty. Z użyciem tej funkcji wypisywane są informacje o zdarzeniach aktualnie zachodzących w systemie.
- warn: wywołania log:info są ignorowane. . Przy użyciu tej funkcji wypisywane są przykładowo informacje o niepoprawnej sumie kontrolnej

- error: wywołania log:info oraz log:warn są ignorowane. Zarezerwowana jest dla krytycznych błędów.
- none: wszystkie funkcje są ignorowane, nic nie jest wypisywane

Interfejs tych trzech funkcji jest identyczny. Każa występuje w dwóch wersjach:

- log:xxxx(Message) – wypisuje wiadomość (typu string())
- log:xxxx(Format, [Args]) – wypisuje sformatowany napis (string()) z wstawionymi argumentami. Obowiązuje standardowe formatowanie jak we wbudowanym module io.



Rysunek 21: Struktura modułu loggera.

Wypisany komunikat ma strukturę:

```
[level] HH:MM:SS (module:function/arity): message
```

Przykładowo:

```
[info] 18:51:33 (storage_dist_srv:handle_cast/2):
            'ds3@michal-pc' has joined the cluster!
```

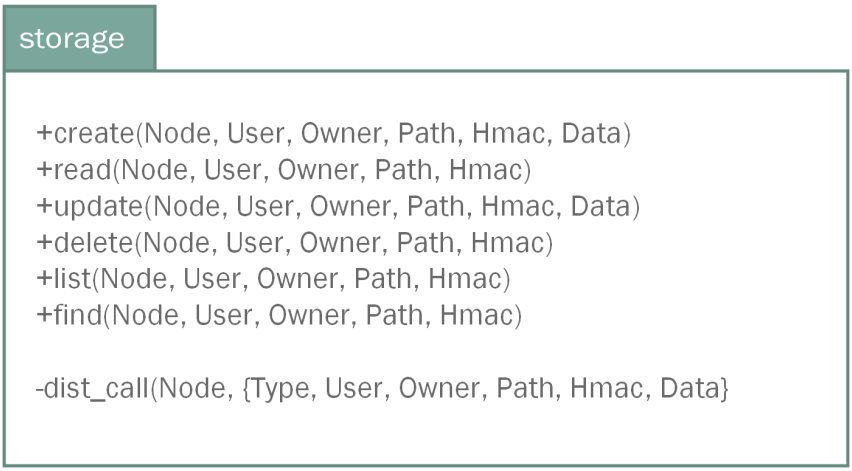
Określenie lokalizacji (funkcji, z której nastąpiło logowanie) odbywa się automatycznie, poprzez symulowane rzucenie wyjątku, złapanie go, a następnie zbadanie stosu wywołań. Może do negatywnie wpływać na wydajność. Logowanie można więc całkowicie zablokować odpowiednią opcją kompilacji.

2.3.10. Biblioteka kliencka

Struktura:

- storage/src/client/storage.erl – moduł

Biblioteka kliencka jest zwykłym modułem języka Erlang. Publiczny interfejs oferuje sześć funkcji, odpowiadających oferowanej przez system funkcjonalności tworzenia (storage:create), czytania (storage:read), aktualizowania (storage:update), usuwania (storage:delete), listowania (storage:list) oraz wyszukiwania plików (storage:find). Moduł przedstawiony jest na Rys. 22.



Rysunek 22: Moduł biblioteki klienckiej.

Wszystkie funkcje mają identyczną sygnaturę. Dodatkowo, storage:create oraz storage:update w ostatnim argumencie przyjmują dodatkowy argument – zapisywany plik w postaci danych binarnych. Kolejne argumenty to:

- Node – adres węzła docelowego (gateway node). Zapytanie trafi do modułu storage_dist_srv na wskazanym węźle.
- User – identyfikator (nazwa) użytkownika wykonującego zapytanie.
- Owner – identyfikator właściciela pliku, do którego odnosi się zapytanie.
- Path – ścieżka do pliku (zgodna z konwencją adresowania w systemie). Musi być unikatowa.
- Hmac – suma kontrolna HMAC-SHA1, obliczona z całego, skonkatenowanego ciała zapytania, zaszyfrowana sumą SHA1 obliczoną z hasła użytkownika (kluczem prywatnym)
- Data – dane binarne

Sposób działania wszystkich funkcji jest identyczny. Konstruują odpowiednią strukturę Request, wysyłają ją do wskazanego węzła (proces storage_dist_srv) a następnie oczekują na odpowiedź.

Odpowiedzi mają postać:

- ok, SuccessResponse
- error, Reason

W przypadku pozytywnej odpowiedzi, SuccessResponse do binarna zawartość odczytanego pliku w przypadku zapytania read, lista adresów wszystkich plików w przypadku zapytania list czy też adres węzła przechowującego dany plik w przypadku zapytania find.

Reason to zawsze atom, opisujący przyczynę błędu, przykładowo not_found.

2.4. GUI

TODO

3. Implementacja

Rozdział ten opisuje działanie pojedynczego węzła bez zagłębiania się w szczegóły jego architektury. Węzeł jest samodzielną aplikacją. Może więc działać bez obecności innych węzłów. Sam stanowi wtedy jedyny punkt w systemie i przesyła zapytania sam do siebie. Taka konfiguracja, choć mało praktyczna, jest możliwa.

3.1. Cykl życia instancji

Administrator ma możliwość swobodnego włączania i wyłączania nowych węzłów. Po uruchomieniu węzła (aplikacji), uruchamiany jest supervisor. Uruchamia on kolejno wszystkie gen_servery:, w kolejnośći:

1. storage_uuid_srv
2. storage_core_srv
3. storage_auth_srv
4. storage_dist_srv
5. storage_http_srv

Każdy uruchomiony jest w trybie one-for-one – w przypadku awarii jednego z nich, tylko on zostanie uruchomiony ponownie. Maksymalna liczba prób ponownego uruchomienia dla każdego gen_servera wynosi 5. Po przekroczeniu tej liczby, uznaje się, że awaria jest krytyczna.

Kolejność uruchamiania może być dowolna – procedury inicjalizacyjne poszczególnych serwerów nie zależą od dostępności innych. Należy jednak mieć na uwadze, że uruchomienie serwera storage_dist_srv spowoduje, że system będzie mógł przyjmować komunikaty od użytkowników, więc storage_core_srv powinien już działać.

Wszystkie zmiany w metadanych plików są persystowane w bazie danych. Po wyłączeniu / awarii węzła i ponownym jego włączeniu, zapisane dane zostają odtworzone. Jeżeli jakieś zapytanie było w kolejce wątku wykonawczego który został zatrzymany, zapytanie zostaje zgubione i użytkownik musi wykonać je ponownie.

Jeżeli węzeł nie przeprowadza aktualnie żadnych operacji, może zostać bezpiecznie wyłączony.

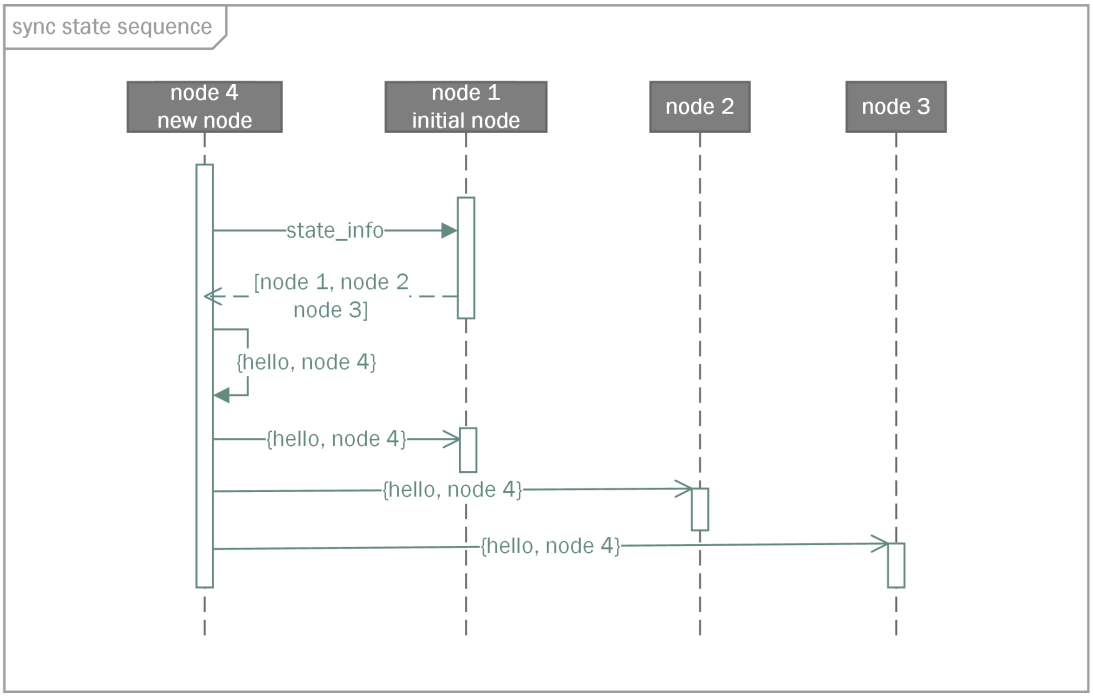
3.2. Wewnętrzne protokoły

Istnieją procedury wymagające współpracy wielu węzłów, bardziej skomplikowane niż przykładowo tworzenie pliku w systemie. Jedna z nich to procedura dołączania nowego węzła do systemu, w wyniku której każdy węzeł w systemie dysponuje aktualnym jego obrazem i jest poinformowany o dołączeniu nowego węzła. Druga z procedur ma miejsce, kiedy dochodzi do autentykacji zapytania. Dane użytkowników są przechowywane w różnych węzłach i mogą ulegać zmianom w czasie, dlatego należy dokonać takiego wyszukiwania w optymalny sposób ograniczając liczbę przesyłanych wiadomości.

3.2.1. Dołączanie węzłów (synchronizacja stanu)

Każdy węzeł na starcie pobiera z pliku konfiguracyjnego lokalizację (adres) węzła początkowego (initial node). Jest to węzeł działający w klastrze, do którego dołączany jest nowy węzeł. Może to być również adres własny jeżeli węzeł ma działać samodzielnie / być pierwszym węzłem w klastrze.

Nowy węzeł w trakcie uruchamiania rozpoczyna procedurę *remote scan* (storage_dist_srv:remote_scan/1). Wykonywana jest w trakcie inicjalizacji modułu storage_dist_srv. Rys. 23 pokazuje przebieg sekwencji wywołań i komunikatów tej procedury.



Rysunek 23: Synchronizacja listy węzłów. Uproszczony wariant z jednym initial node.

Przyjmuje ona zbiór adresów węzłów (domyślnie znajduje się tam tylko initial node). Ze zbioru usuwany jest aktualny węzeł (zbiór może przez to stać się pusty). Następnie do każdego serwera storage_dist_srv działającego na każdym z węzłów ze zbioru wysyłany jest komunikat (atom) state_info. Odpowiedzią na ten komunikat jest zwrócenie pytającemu informacji o znanych sobie węzłach w postaci listy. Domyślny węzeł łączy te listy w jedną, z której następnie usuwa duplikaty.

Ten nowy zbiór staje się stanem nowego węzła i będzie udostępniany w wywołaniach state_info kierowanych do niego. Ostatnim krokiem jest rozesłanie do wszystkich poznanych węzłów komunikatu {hello, node()} z własnym adresem. Każdy z nich uaktualni wtedy listę znanych sobie węzłów o nowy węzeł.

3.2.2. Synchronizacja użytkowników

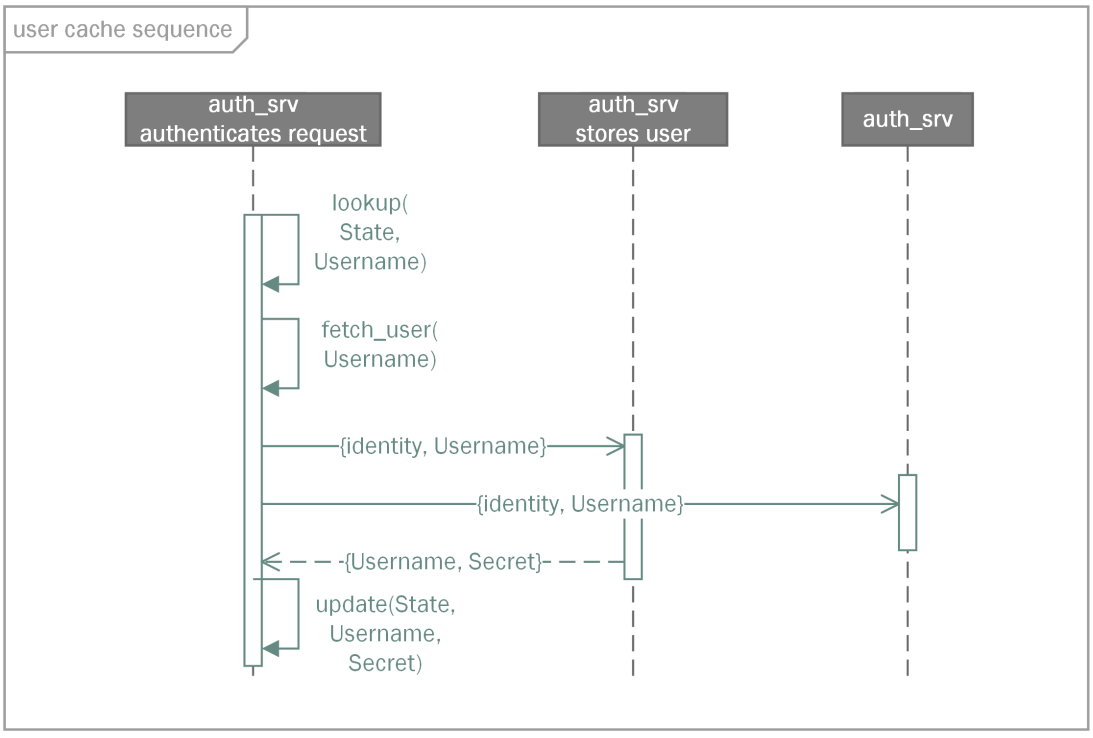
Użytkownicy przechowywani są w węźle, na którym zostało utworzone ich konto (dodane do bazy danych). Ponieważ użytkownik może wykonywać zapytania wysyłając je do dowolnego węzła, należy zapewnić możliwość uzyskania dostępu do danych użytkownika bez informacji o tym, na którym węźle znajdują się informacje o jego koncie.

Informacją niezbędną do autentykacji zapytania jest klucz prywatny użytkownika. Moduł storage_auth_srv działający na każdym węźle zarządza listą krotek {username, secret, expires}. Taka struktura wystarcza żeby dokonać autentykacji użytkownika.

Kiedy storage_auth_srv autentykuje zapytanie, szuka użytkownika w opisanej wyżej liście. Jeżeli go znajdzie, wykorzystuje skojarzony klucz prywatny do wyliczenia sumy kontrolnej. Jeżeli nie ma takiego użykownika, wysyła zapytanie rozgłoszeniowe do wszystkich innych modułów storage_auth_srv. Odpowiada na nie węzeł, który znajdzie w swojej bazie danych szukanego użytkownika. Po pewnym czasie lista zapełni się użytkownikami i nie będzie trzeba sięgać do innych węzłów w celu autentykacji zapytania.

Dane użytkownika mogą się zmienić (może na przykład zmienić hasło). By w systemie nie zalegały nieaktualne dane, każdy rekord ma dodatkowo pole expires, które ustawiane jest domyślnie na 10 minut. Przedawnione wpisy są traktowane na równi z brakiem wpisów w liście użytkowników.

Jeżeli walidacja przy pomocy przechowywanego w liście klucza nie powiedzie się, istnieje przypuszczenie, że użytkownik zmienił swoje hasło i podpisał wiadomość nowym kluczem. W takim wypadku komunikat z poszukiwaniem użytkownika jest jeszcze raz rozgłaszany w systemie.



Rysunek 24: Wyszukiwanie użytkownika w systemie. Użytkownik nie zostaje znaleziony w lokalnym cache. Rozgłaszany jest komunikat identity, Username. Odpowiedni węzeł zwraca informacje na temat użytkownka lista cache ulega aktualizacji.

4. Technologie

Niniejszy rozdział opisuje szczegóły techniczne organizacji projektu, kompilacji kodu źródłowego i uruchamiania zbudowanej aplikacji.

4.1. Struktura projektu

W głównym katalogu repozytorium znajduje się wiele folderów i plików.
Katalogi:

- benchmark/ - testy wydajnościowe, napisane w Erlangu wykonujące dużą ilość zapytań w systemie i mierzące czas obsługi. Testy uruchamiane są poprzez skrypty bash.
- clustertool/ - dwa skrypty: make_cluster.sh pozwalający na wygenerowanie klastra o dowolnej konfiguracji oraz cluster.sh służący do zarządzania klastrem / węzłem działającym na lokalnej maszynie
- curl/ - implementacja metod PUT (put.sh), POST (post.sh) i DELETE (delete.sh) przy pomocy programu curl. Sumy HMAC wyliczane są przez openssl-client
- erlang-sqlite/ - sterownik bazy danych
- storage/ - właściwy kod źródłowy aplikacji Erlang/OTP

Pliki:

- rebar – skrypt używany do budowania aplikacji
- Makefile – makefile będący nakładką na rebara. Interesujące cele: all, release
- env_server.sh – skrypt uruchamiający interpreter języka Erlang i ładujący wszystkie skompilowane moduły. Aplikację można uruchomić wpisując:

  application:start(sqlite3). application:start(storage).
- env_test.sh – skrypt uruchamiający interpreter z załadowaną bibiloteką kliencką
- node.config – plik z konfiguracją budowanego węzła. Więcej szczegółów w rozdziale 4.5 Pliki konfiguracyjne
- node.config.template – plik z ustawieniami referencyjnymi
- setup-rhel.sh – skrypt konfigurujący repozytorium i środowisko w systemach RedHat. Można również zobaczyć jak należy ustawiać parametry do komunikacji węzłów w sieci Internet.
- test.config(.template) – odpowiednik node.config dla benchmarków

Właściwa aplikacja Źródła aplikacji znajdują się w katalogu storage/. Struktura podkatalogów przedstawia się następująco:

- include/shared.hrl – definice rekordów i makr
- priv/manager.html – web-gui
- rel/ - konfiguracja reltoola i miejsce generowania releasa
- src/ - pliki źródłowe
  - auth/ – moduł autentykacji
  - client/ - moduł biblioteki klienckiej
  - core/ - moduł wykonawczy
  - dist/ - moduł komunikacyjny
  - http/ - moduł HTTP
  - shared/ - współdzielone moduły
- test/accept.sh – testy akceptacyjne

4.2. rebar

rebar jest narzędziem wykorzystywanym do budowania aplikacji. Jest to samodzielny skrypt, konfigurowalny przez pliki rebar.config (zawierające struktury języka Erlang) znajdujące się w katalogu głównym i podkatalogach.

W głównym katalogu znajduje się plik rebar.config o następującej treści:
sub_dirs, [śtorage", "benchmark"] .
Oznacza to że w te dwa katalogi zostaną przeszukane podczas kompilacji projektu.

4.3. Kompilacja

Plik storage/rebar.config ma następującą zawartość:

```
{sub_dirs, ["rel"]}.

{erl_opts, [

    % compile-time options (no'opt' disables option, e.g. nolog)
    {d, log},           % enable logger
    {d, auth},          % authenticate requests
    {d, fileio},        % read / write real files
    {d, actionlog},     % track and log user actions
    {d, persistentdb}, % disk / memory db
    {d, noprofile},     % run with fprof profiler

    {i, "include"},
```

```
    {src_dirs, [
        "src",
        "src/shared",
        "src/core",
        "src/client",
        "src/dist",
        "src/http"
    ]}
]}.
```

Pierwsza linia mówi, że należy przeszukać podkatalog rel/. Znajduje się tam konfiguracja innego narzędzia, reltoola, uruchamianego poleceniem ./rebar generate, tworzącego samodzielny release.

Następnie definiowany jest szereg flag, które blokują bądź odblokowują pewne funkcje systemu (najczęściej zdefiniowane w postaci makr zależnych od wyżej wymienionych symboli). Dostępne opcje:

- log / nolog – włączenie / wyłączenie loggera
- auth / noauth – włączenie / wyłączenie autentykacji zapytań (przydatne do testów kiedy nie mamy możliwości wyliczenia sumy HMAC)
- fileio / nofileio – wykonywane / pomijane operacje na fizycznych plikach (zapisy są ignorowane, przy odczycie zwracana jest pusta zawartość)
- actionlog / noactionlog – włączenie / wyłączenie logowania zachodzących akcji do bazy danych
- persistentdb / nopersistentdb – baza zapisywana na dysku / działająca w trybie in-memory.
- profile / noprofile – urchom razem z profilerem

Ostatnia sekcja definiuje położenie plików źródłowych.

4.4. Testy akceptacyjne

Testy akceptacyjne zawarte są w skrypcie storage/test/accept.sh. Testowany jest następujący scenariusz:

- POST - utworzenie pliku o rozmiarze 128 MB (z losową zawartością)
- GET – pobranie utworzonego pliku, porównanie bajt po bajcie
- PUT – aktualizacja pliku losowymi danymi o rozmiarze 256 MB
- GET - pobranie utworzonego pliku, porównanie bajt po bajcie
- DELETE – usunięcie utworzonego pliku, oczekiwana odpowiedź: 202 Accepted

- GET – pobranie usuniętego pliku, oczekiwana odpowiedź: 404 Not Found

Test komunikuje się z systemem poprzez protokół HTTP. Należy znać więc adres na jakim nasłuchuje moduł storage_http_srv. Wtedy test można uruchomić tak:

./accept.sh localhost:8090

Skrypt do uruchomienia wymaga narzędzi curl (komunikacja HTTP) i openssl-client (suma HMAC, generowanie losowych plików).

4.5. Pliki konfiguracyjne

Podstawowym plikiem konfiguracyjnym jest node.config umieszczony w głównym katalogu projektu. Przykładowa zawartośc:

```
{app_log_level, info}.

{core_work_dir, "/home/michal/Documents/inz/work_dir"}.
{core_storage_quota,      1073741824}.
{core_memory_quota,       134217728}.

{dist_initial_node, "ds@michal-pc"}.

{uuid_interface_name, "eth0"}.

{http_port, 8090}.
```

Dostępne parametry to:

- app_log_level – poziom szczegółowości komunikatów loggera. Dostępne wartości: info, warn, error oraz none
- core_work_dir – fizyczna lokalizacja gdzie będą przechowywane pliki użytkowników i baza danych
- core_storage_quota – dostępne zasoby dyskowe, w bajtach (maksymalna pojemność węzła)
- core_memory_quita – dostępna pamięć ram, w bajtach (przy przekroczeniu tej wartości zostawał uruchomiony gabage collector, obecnie nieużywane)
- dist_initial_node – adres początkowego węzła, który posłuży do inicjalizacji aktualnego. Procedura opisana jest w punkcie 3.2.1 Dołączanie węzłów.
- uuid_interface_name – nazwa interfejsu sieciowego z którego pobrany zostanie adres MAC wykorzystywany przez generator identyfikatorów
- http_port – port na jakim nasłuchuje moduł HTTP

W czasie generowania releasa, plik ustawień zostanie dołączony do wynikowej paczki.

4.6. Zarządzanie klastrem

Klaster to kilka połączonych węzłów współpracujących ze sobą. Klaster można wygenerować przy użyciu skryptu clustertool/make_cluster.sh. Wytworzy on we wskazanym folderze podfoldery ds1, ds2, ..., z odpowiednio ustawionymi initial node oraz portami. Następnie można przenieść je na docelowe maszyny i uruchomić przy użyciu narzędzia clustertool/cluster.sh. Procedury te opisuje dokumentacja administratora.

Materiały źródłowe

[1] en.wikipedia.org. Genetic algorithm. http://en.wikipedia.org/wiki/Genetic_algorithm#Related_fields.

[2] Ericsson AB. Otp design principles. http://www.erlang.org/doc/design_principles/des_princ.html.

Spis rysunków