**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.

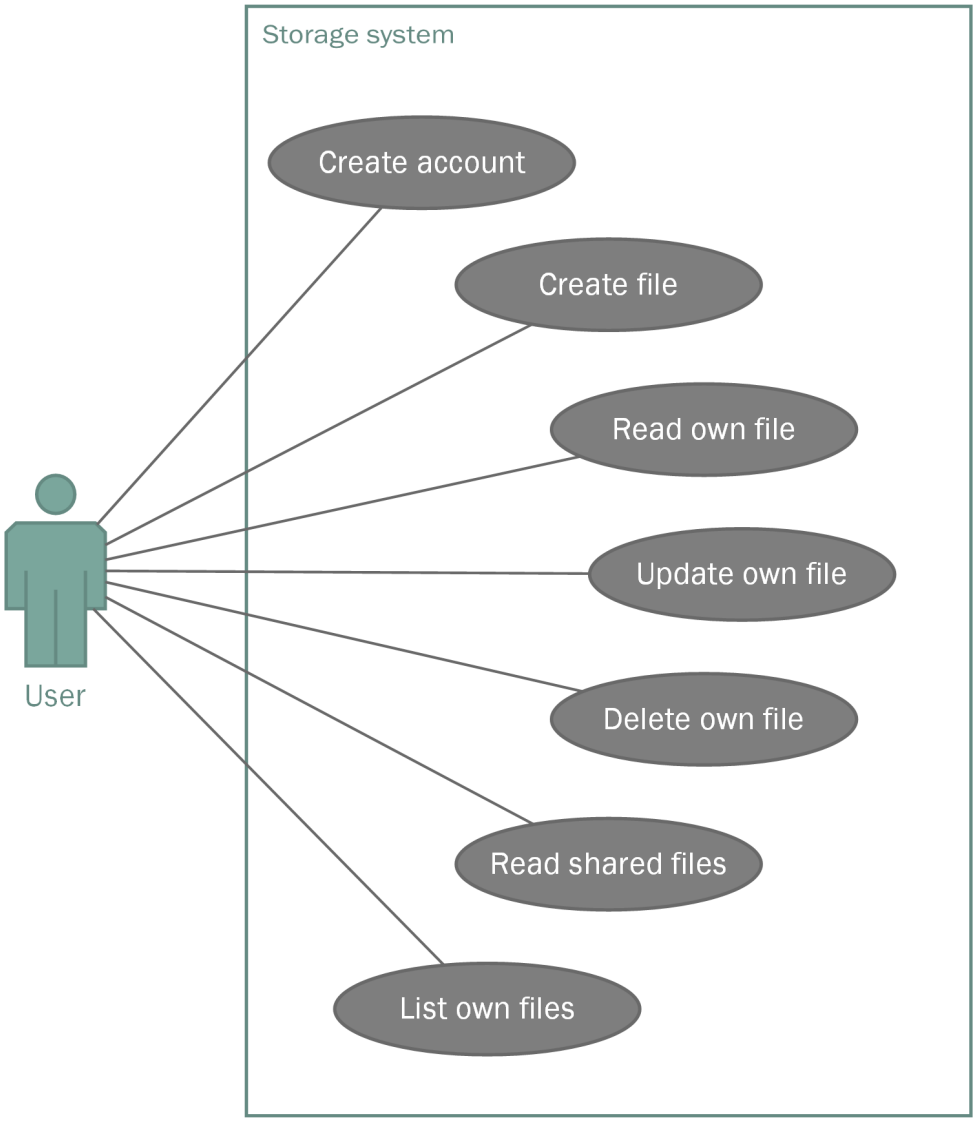
Aplikacja monitoruje wszystkie akcje, jakie podejmują użytkownicy. Są one zapisywane w bazie danych (aktualnie jest to SQLite 3). Przy ich pomocy system ustala priorytety dla poszczególnych zapytań podczas fazy schedulingu.

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

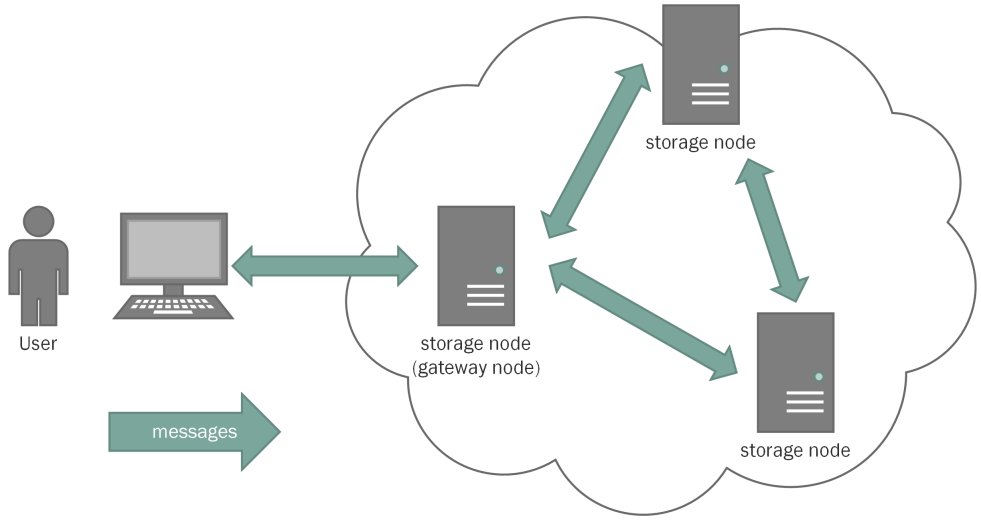
# 2. Architektura

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:

```
1 -record(user, {
2     name        :: nonempty_string(),
3     secret      :: nonempty_string(),
4     create_time :: integer(),
5 }).
```

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:

```
1 -record(file, {
2     owner       :: nonempty_string(),
3     vpath       :: nonempty_string(),
4     bytes       :: integer(),
5     access_mode :: integer(),
6     create_time :: integer()
7 }).
```

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

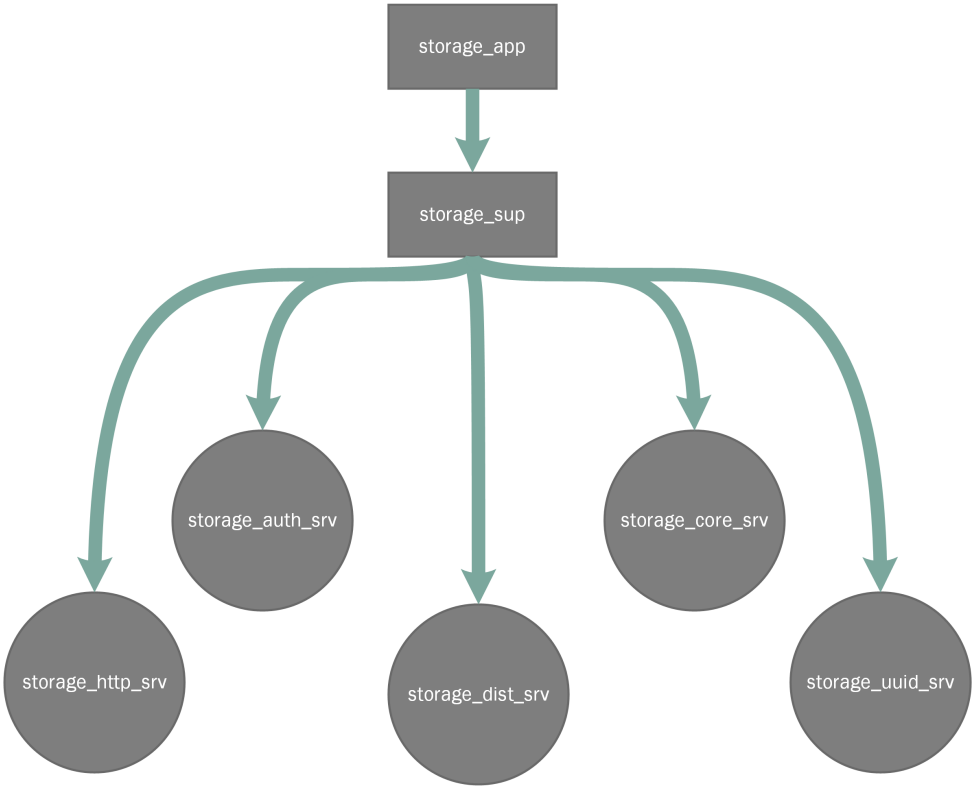
Użycie modułu storage_http_srv jest opcjonalne – użytkownik może również skorzystać z biblioteki klienckiej (napisanej w Erlangu) i pominąć wysyłanie zapytania przez protokół HTTP.

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

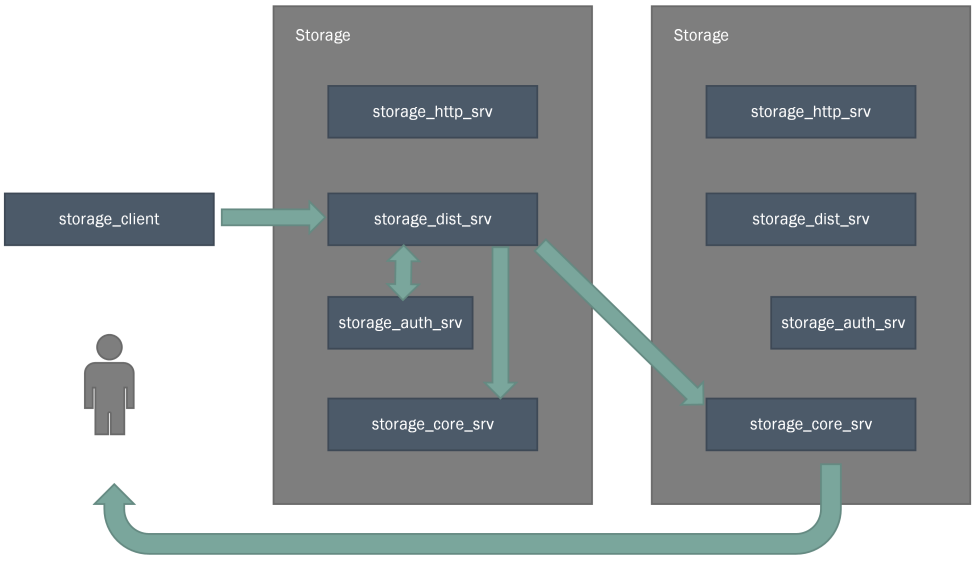
### 2.3.1. Komunikacja wysokopoziomowa

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



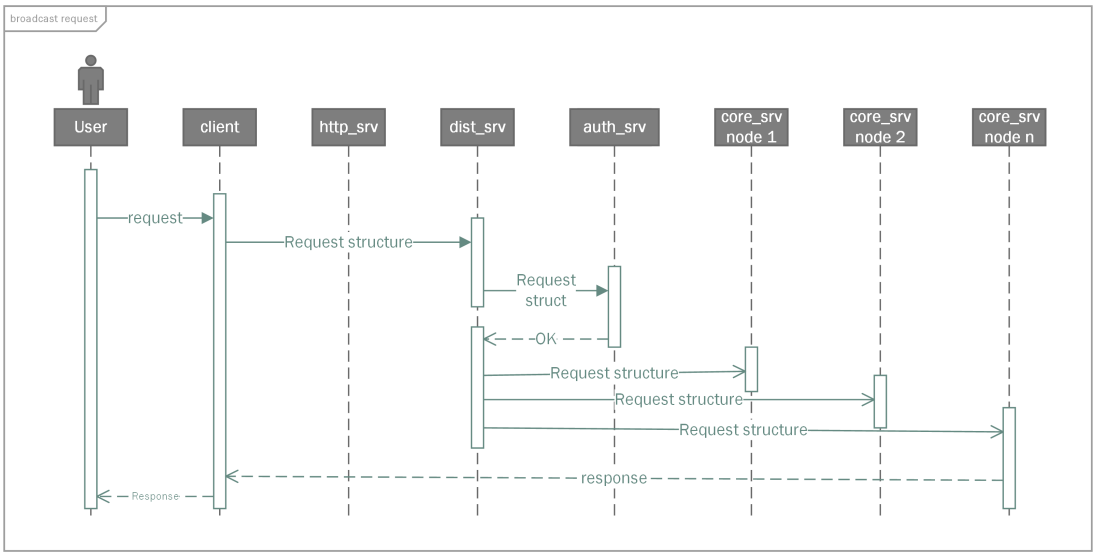

Rysunek 3: Supervision tree. Aplikacja zbudowana jest z pięciu gen_serverów. Każdy jest osobnym procesem. Ponadto storage_core_srv zarządza pulą procesów wykonawczych.




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.

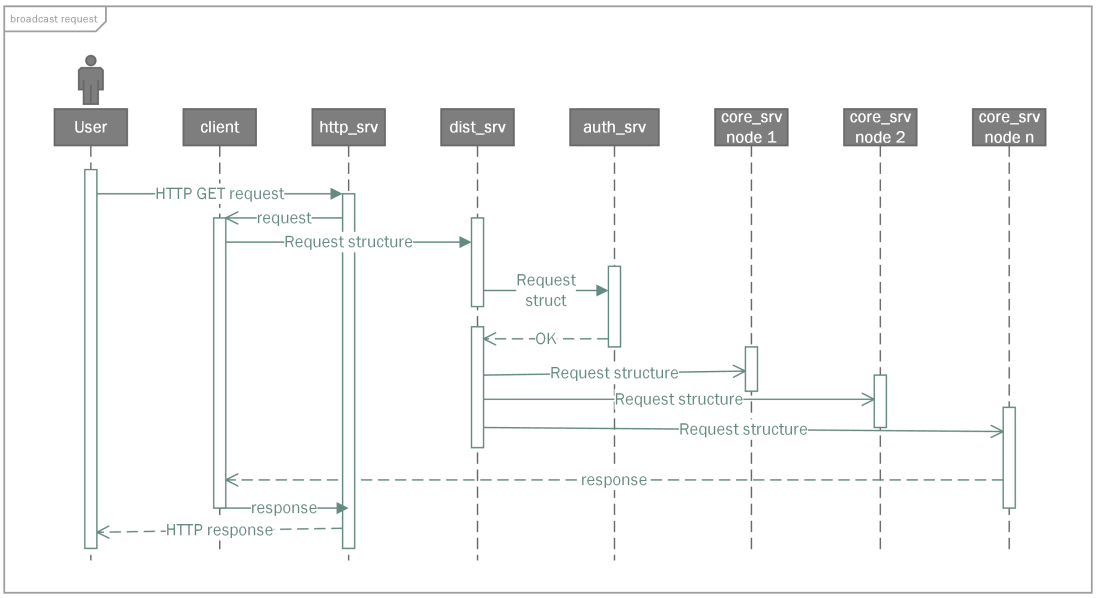
Rys. 5 pokazuje diagram sekwencji obsługi zapytań rozgłoszeniowych z wykorzystaniem biblioteki klienckiej. Zapytania tego typu to zapytania read, delete oraz find. Rys. 6 pokazuje te same akcje przy wykorzystaniu modułu HTTP. Widać, że moduł HTTP wewnętrznie korzysta z biblioteki klienckiej. Rysunki 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.

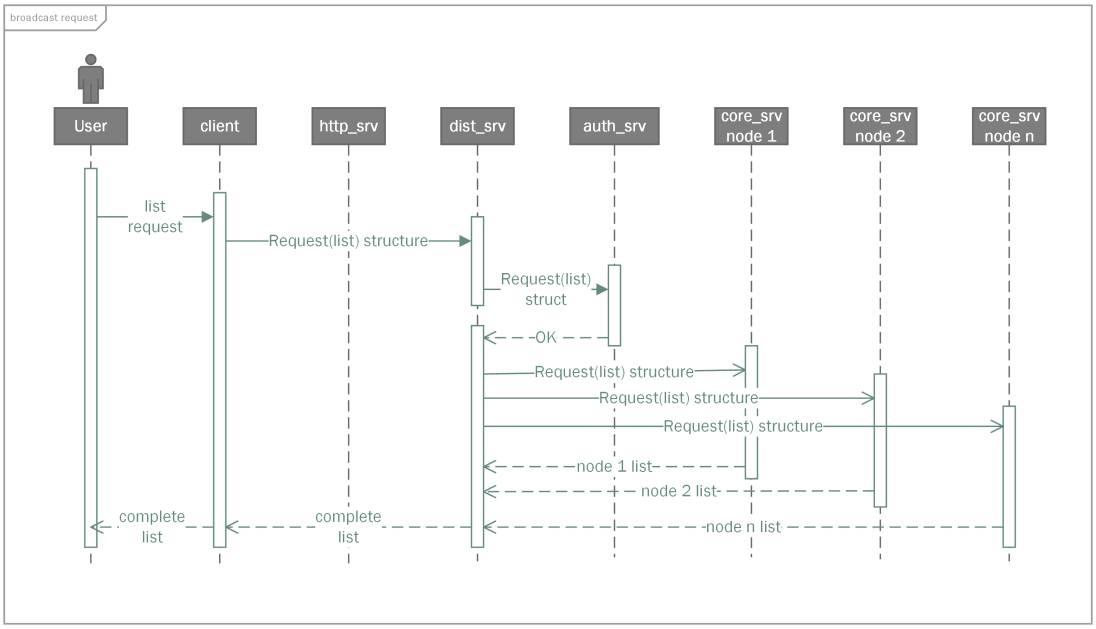
W przypadku zapytania o listę wszystkich plików (zapytanie list), przeszukane muszą zostać wszystkie węzły. Odpowiedź nie wraca jednak z każdego z nich bezpośrednio do klienta, lecz do modułu dist_srv, odpowiedzialnego za połączenie wszystkich list w jedną listę wynikową. Sytuację tę przedstawia diagram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

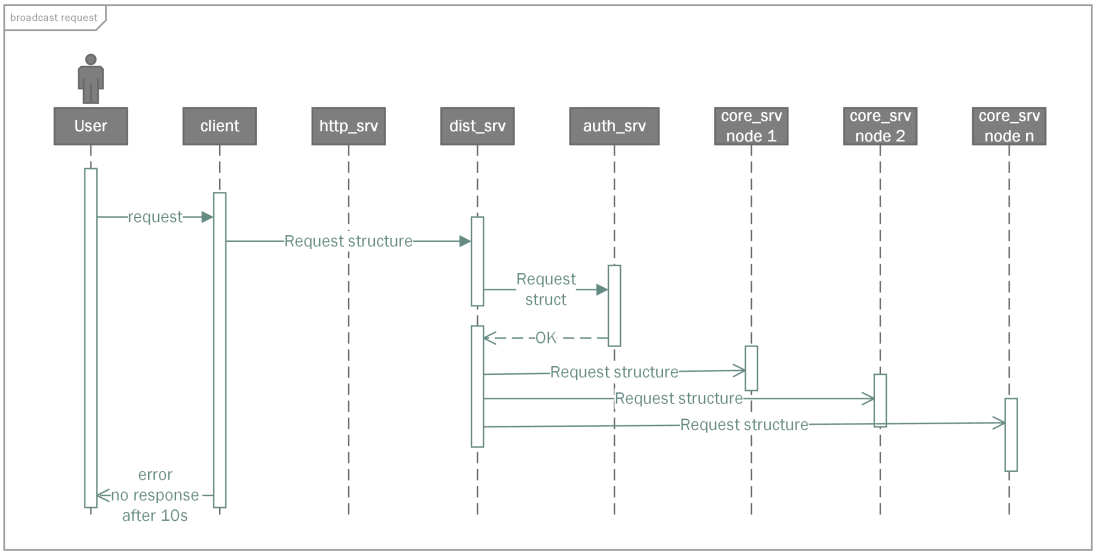



Rysunek 6: Zapytanie rozgłoszeniowe z wykorzystaniem modułu HTTP. Użytkownik wysyła zapytanie metodą GET, które tłumaczone jest na odpowiednią strukturę Request.




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).

# 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