Uniwersytet Warszawski
Wydział Matematyki, Informatyki i Mechaniki

Konrad Witkowski
Nr albumu: 189454

Rozproszony system plików do obsługi serwisów internetowych

Praca magisterska
na kierunku INFORMATYKA

Praca wykonana pod kierunkiem
dr Janiny Mincer-Daszkiewicz
Instytut Informatyki

czerwiec 2010

Oświadczenie kierującego pracą

Potwierdzam, że niniejsza praca została przygotowana pod moim kierunkiem i kwalifikuje się do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

Data Podpis kierującego pracą

Oświadczenie autora pracy

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

Oświadczam ponadto, że niniejsza wersja pracy jest identyczna z załączoną wersją elektroniczną.

Data Podpis autora pracy

Streszczenie

Systemy rozproszone, a w szczególności rozproszone systemy plików, stają się obecnie bardzo popularne. Ze względu na wiele pozytywnych cech takich systemów, znajdują one coraz szersze zastosowanie w wielu dziedzinach. Obsługa serwisów internetowych, ze względu na strukturę oraz ilość utrzymywanych tam danych, stanowi jedno z zastosowań, do którego rozproszone systemy plików zdają się pasować idealnie.

W niniejszej pracy przedstawiony został projekt, implementacja i testy rozproszonego systemu plików do obsługi serwisów internetowych. W pierwszej części pracy opisane zostały istniejące rozwiązania oraz ich cechy wpływające na możliwość wykorzystania w różnych zastosowaniach. Następnie opisano cele stawiane przed tworzonym systemem. W kolejnych rozdziałach przedstawiony został ogólny projekt systemu oraz projekt jego implementacji. W ramach pracy wykonano testy na gotowym systemie oraz dokonano analizy płynących z nich wniosków. Ostatni rozdział stanowi podsumowanie całej pracy i zawiera informacje o możliwych rozszerzeniach systemu.

Słowa kluczowe

system rozproszony, system plików, serwis internetowy, bezpieczeństwo, spójność, sieci komputerowe

Dziedzina pracy (kody wg programu Socrates-Erasmus)

11.3 Informatyka

Klasyfikacja tematyczna

D. Software
D.4. Operating Systems
D.4.3. File Systems Management

Tytuł pracy w języku angielskim

Distributed file system for internet services

Spis treści

Spis rysunków

Wprowadzenie

Na początku 2009 roku liczba użytkowników internetu na świecie przekroczyła miliard. W tym samym czasie liczba rozmaitych serwisów internetowych osiągnęła próg 200 milionów. Według różnych badań, 90% internautów korzysta z poczty elektronicznej, a ponad 70% przegląda specjalizowane strony tematyczne w poszukiwaniu informacji. Ponadto coraz większą popularnością cieszą się serwisy z plikami multimedialnymi, jak YouTube.com, oraz różnego rodzaju portale społecznościowe, wśród których na pierwszy plan wysuwają się MySpace i Facebook. Mimo iż wszystkie te kategorie serwisów internetowych różnią się od siebie znacząco, zarówno treścią, jak i sposobem jej wykorzystania, mają jedną cechę wspólną – większość z nich przechowuje i udostępnia użytkownikom bardzo duże ilości danych. W związku z ciągle rosnącymi potrzebami konieczne stało się opracowanie systemów plików mogących przechowywać i sprawnie udostępniać milionom użytkowników serwisów internetowych petabajty danych.

W 2003 roku w firmie Google powstał Google File System – rozproszony system plików pracujący na popularnych komputerach klasy PC [6]. Ponieważ był został on stworzony dla zastosowań wewnętrznych, nie było potrzeby implementowania interfejsu POSIX, a model spójności danych mógł zostać dostosowany do potrzeb konkretnego zastosowania (więcej o modelach spójności można znaleźć w [13]). Poza podstawowymi celami projektowymi, jak wydajność, skalowalność i duża dostępność danych, system miał być zoptymalizowany dla niestandardowych wzorców dostępu. Duże sekwencyjne odczyty oraz dopisywanie danych do pliku dużymi porcjami są najczęściej wykonywanymi operacjami w systemie. Wynika to z głównego zastosowania, dla którego system powstał – współpraca z silnikiem wyszukiwarki internetowej. Inne operacje, takie jak odczyt i nadpisywanie danych, są obsługiwane, ale ich znaczenie jest marginalne. Dodatkowo, jednym z priorytetowych celów projektowych było wykorzystanie maksymalnej przepustowości sieci przesyłowej (w opozycji do minimalizacji opóźnień w dostarczaniu danych). Takie założenie, w połączeniu z typowymi wielkościami przechowywanych plików wahającymi się od kilkuset megabajtów do kilku terabajtów, ogranicza pola zastosowań systemu. W kolejnych latach powstało kilka implementacji systemów opartych na GoogleFS lub do niego podobnych (np. HDFS – Hadoop Distributed File System [7] lub MooseFS firmy Gemius [10] – korzystanie z obydwu jest darmowe, w odróżnieniu od GoogleFS). Większość z nich odziedziczyła cechy systemu bazowego.

Innym przykładem systemu stworzonego na potrzeby wewnętrzne był Amazon Dynamo [5]. Główne zastosowanie systemu związane jest z serwisami oferowanymi przez Amazon (Amazon Web Services), szczególnie S3 (Simple Storage Service). Poza wymaganiami skalowalności i dużej dostępności, Dynamo projektowane było pod inny model zapytań. Przechowywane dane są małe (poniżej 1 MB), a operacje dotyczą dokładnie jednego wpisu identyfikowanego po kluczu binarnym. W przeciwieństwie do GoogleFS, Dynamo jest nastawiony na minimalne opóźnienia w dostarczaniu danych. Dostęp do danych jest zapewniany przez funkcje *put()* i *get()* realizując idee pamięci typu *klucz-wartość* (ang. *key-value storage*). Podobnie jak GoogleFS, Dynamo umożliwia konfigurowanie stopnia replikacji danych dla każdej

instancji systemu. Gwarantowana jest *spójność ostateczna* danych, co oznacza, że jeżeli wystarczająco długo nie będzie awarii, to wszystkie zmiany zostaną rozpropagowane. Ponadto system wykorzystuje kilka bardziej złożonych mechanizmów dla zapewnienia wysokiej dostępności i odporności na awarie.

Istnieje wiele innych rozproszonych systemów plików oferujących funkcjonalność ściśle związaną z określonym zastosowaniem. Uniwersalne systemy plików, często udostępniające interfejs zgodny ze stanardem POSIX (np. Ceph [14]), zapewniają złożone wymagania dotyczące spójności, ale realizowane jest to poprzez zwiększenie narzutu związanego z synchronizacją. To z kolei przekłada się na zmniejszenie wydajności systemu przy określonych zasobach sprzętowych.

Celem mojej pracy jest zaprojektowanie i zaimplementowanie rozproszonego systemu plików, który współpracowałby z różnymi serwisami internetowymi w celu przechowywania i udostępniania danych. Zastosowanie systemu wiąże się głównie z dwoma najbardziej popularnymi typami usług:

- serwisy poczty elektronicznej,
- serwisy z plikami multimedialnymi.

Zastosowania te mają pewne charakterystyczne dla siebie wymagania, ale można wymienić również kilka wspólnych. Jak w większości serwisów, oczekuje się od użytego systemu plików zapewnienia skalowalności i dużej dostępności, a także odporności na awarie (zob. [4]). Ze względu na to, że dane przesyłane są do użytkowników, a nie do innych systemów komputerowych, ważne jest, aby opóźnienia w systemie były minimalne.

Systemy poczty elektronicznej przechowują bardzo dużo niewielkich plików. Podstawowym obiektem w systemie pocztowym jest wiadomość. Rozmiar większości wiadomości waha się od kilkuset bajtów do kilkudziesięciu megabajtów. Model dostępu do danych obejmuje kilka elementów. Dane są w systemie zapisywane raz, a odczytywane zwykle kilka razy. Dostęp do danych jest losowy, jednak najczęściej odczytywane są dane ostatnio zapisane. Liczba obiektów przechowywanych w systemie pocztowym jest bardzo duża. Statystyczny użytkownik przechowuje w skrzynce pocztowej kilka tysięcy wiadomości, co odpowiada kilku miliardom plików przechowywanych w systemach o milionie użytkowników.

Dodatkowo, systemy pocztowe umożliwiają generowanie list wiadomości zawartych w skrzynce użytkownika. Jest to operacja równie częsta, jak odczyt czy zapis wiadomości. Przekłada się ona na sekwencyjny odczyt dużych fragmentów danych w użytym systemie plików. Dlatego w takich systemach konieczna jest efektywna realizacja dużych, sekwencyjnych odczytów. Usuwanie danych zwykle dotyczy wielu obiektów.

Systemy z plikami multimedialnymi przechowują dane znacznie większych rozmiarów. Podstawowym obiektem w takim systemie jest plik, którego wielkości zaczynają się od kilkudziesięciu kilobajtów, a kończą na kilkuset megabajtach. Liczba plików w systemie jest znacznie mniejsza i sięga od kilku do kilkunastu milionów. Dostęp do plików jest losowy, a podstawowe operacje dotyczą zapisu i odczytu danych identyfikowanych poprzez klucz binarny. Raz zapisane dane są wielokrotnie odczytywane (zwykle tysiące razy).

Żaden ze znanych mi rozproszonych systemów plików nie realizuje wymagań stawianych przez obydwa wymienione typy serwisów. GoogleFS efektywnie realizuje operacje sekwencyjne, na dużych danych, ale losowy dostęp do niewielkich plików jest dla niego problemem. Amazon Dynamo zapewnia efektywne operacje na losowych danych, jednak nie był projektowany do operacji sekwencyjnego odczytu. Poza tym korzystanie z systemu jest płatne, a żadne otwarte implementacje nie istnieją. Stosowanie systemów zapewniających pełną spójność i zgodnych ze standardem POSIX mija się z celem, gdyż wydajność tych systemów jest

mocno obniżona poprzez realizację złożonych mechanizmów wewnętrznych. Żaden z przedstawionych systemów nie umożliwia także konfiguracji stopnia replikacji dla konkretnego obiektu – stopień konfiguracji ustalany jest per instancja systemu. Taka konstrukcja wymaga kilku instalacji, jeżeli chcemy przechowywać dane z różnym stopniem zabezpieczenia.

Z tych powodów postanowiłem stworzyć system plików spełniający wszystkie wymagania stawiane przez wymienione serwisy internetowe, łączący w sobie pozytywne cechy opisanych systemów.

Praca składa się z kolejnych rozdziałów, odpowiadających etapom tworzenia systemu:

- w rozdziale **Projekt systemu** zostaną opisane założenia projektowe stawiane przed projektem oraz ogólny opis systemu z podziałem na główne elementy,
- rozdział **Projekt implementacji** będzie obejmował dokładny opis struktur danych i algorytmów wykorzystywanych przy budowie systemu, a także wszystkie operacje, na których wykonanie pozwala system,
- rozdział **Testy systemu** zawiera opis testów przeprowadzonych na systemie oraz wnioski z nich płynące w odniesieniu do spełniania celów projektowych,
- rozdział **Podsumowanie** zawiera podsumowanie pracy oraz opis możliwych kierunków rozbudowy systemu.

Rozdział 1

Projekt systemu

1.1. Założenia projektowe

W tym rozdziale przedstawiony zostanie projekt rozproszonego systemu plików do obsługi serwisów internetowych. Ze względu na dość złożone wymagania odnośnie takich systemów plików, przyjęto pewne założenia dotyczące modelu danych.

Przechowywanie dużej liczby obiektów w każdym systemie wiąże się z koniecznością przechowywania dużej ilości metadanych. Z tego powodu w projektowanym systemie podstawowym obiektem będzie rekord. Rekordy z kolei będą zorganizowane w pliki. Taka struktura danych ułatwia rozproszenie metadanych w systemie. Gdy w treści rozdziału będzie mowa o obiekcie systemu chodzi tu zawsze o rekord, chyba że z kontekstu wynika inaczej. Plik systemowy będzie stanowić zbiór rekordów. Niezależnie od zastosowania, rekord systemowy będzie odpowiadał ciągowi danych binarnych przechowywanych w systemie, a plik będzie stanowił zbiór rekordów.

Przyjęcie takiego sposobu organizacji danych umożliwia dobry podział logiczny systemu oraz dobrze współgra ze wszystkimi wymaganiami stawianymi przed systemem, które przedstawię dalej w tym punkcie. Punkt ten zawiera także opis środowiska pracy systemu. W kolejnych punktach przedstawię architekturę systemu i sposoby rozwiązania wybranych problemów.

1.1.1. Cele projektowe

Rozpoczynając pracę nad rozproszonym systemem plików chciałem stworzyć system na tyle uniwersalny, aby nadawał się do wykorzystania w wielu serwisach internetowych.

Wśród tych serwisów brałem pod uwagę głównie dwa zastosowania:

- serwisy obsługujące systemy poczty elektronicznej,
- serwisy z plikami multimedialnymi.

Serwisy te różnią się pod względem struktury i sposobu dostępu do danych. W kolejnych punktach przedstawiłem wymagania stawiane przed systemem wraz z krótkim opisem z jakiego zastosowania dane wymaganie wynika.

Dużo małych obiektów

W serwisach pocztowych mamy czasami do czynienia z bardzo dużą liczbą przechowywanych wiadomości, których rozmiary są bardzo małe (większość wiadomości to pliki tekstowe o

rozmiarach nie przekraczających kilku kilobajtów). Przykładowo, jeżeli w projektowanym systemie będziemy przechowywać dane na komputerze, którego dyski twarde mają łączną pojemność jednego terabajta, to liczba wiadomości (w systemie reprezentowane jako rekordy pliku) przechowywanych na tym komputerze może być rzędu kilku milionów.

Dostęp do wielu obiektów jednocześnie

W serwisach pocztowych mamy także do czynienia z zapytaniami dotyczącymi wielu wiadomości jednocześnie. Przyjmując, że przechowujemy wiadomości jako rekordy pliku (wtedy cały plik odpowiada skrzynce użytkownika systemu pocztowego), generowanie listy wiadomości dla serwerów POP3 czy IMAP będzie wymagało pobrania listy wszystkich rekordów w pliku. W związku z tym konieczne jest wprowadzenie zapytań generujących takie listy. Aplikacje typu Webmail pozwalają także na wyświetlanie list wiadomości w skrzynce pocztowej wraz z tematami, adresatami oraz innymi metadanymi. Tego typu zastosowanie wymaga wprowadzenia do systemu zapytań odczytujących wiele rekordów lub ich części w jednym zapytaniu (generowanie wielu zapytań powodowałoby zbyt duży narzut czasowy na komunikację).

Równoległy dostęp do losowych obiektów

Serwisy internetowe udostępniające pliki multimedialne mają nieco inne wymagania. Przechowywane obiekty są większych rozmiarów (np. niektóre pliki mogą mieć rozmiary przekraczające kilkaset megabajtów). Dostęp do danych jest losowy i w większości przypadków wiele plików multimedialnych jest odczytywanych równolegle (przeglądarki internetowe pobierają obiekty w wielu wątkach). Prawie wszystkie zapytania dotyczą odczytu, zapisu lub usuwania plików. Przechowywanie plików multimedialnych w wielu fragmentach wymaga wsparcia ze strony programu dla równoległych operacji na wielu losowych rekordach systemu.

Minimalne opóźnienia

Jak w każdym systemie współpracującym z interaktywnymi użytkownikami zewnętrznymi, bardzo ważną sprawą są minimalne opóźnienia w dostarczaniu danych. Zbyt długi czas odpowiedzi systemu może spowodować, że użytkownicy przestaną z niego korzystać. Z tego powodu bardzo ważną kwestią jest zachowanie minimalnych opóźnień w rozpoczęciu dostarczania danych do użytkownika. Opóźnienia przekraczające kilkadziesiąt milisekund są już zauważalne i wpływają na spadek komfortu pracy z systemem. Występowanie takich opóźnień jest dopuszczalne, jednak powinno to dotyczyć raczej sytuacji awaryjnych niż typowej pracy systemu oraz być ograniczone do niewielkiej części zapytań.

Ilość danych

Serwisy internetowe często przechowują bardzo duże ilości danych. Szczególnie dotyczy to serwisów udostępniających różne rodzaje plików multimedialnych. Z tego względu każdy rozproszony system plików tworzony na potrzeby serwisów internetowych musi być zdolny do przechowywania dużych ilości danych. Projektowany system będzie umożliwiał przechowywanie kilkuset terabajtów danych, wykorzystując w tym celu kilkaset komputerów.

Zwielokrotnianie

Zwielokrotnianie jest podstawowym sposobem radzenia sobie z awariami, skalowalnością, dostępnością i sposobem na poprawienie efektywności oraz równoważenie obciążenia składowych

systemu (zob. [1] oraz [3]). Jednak wykorzystanie zwielokrotniania pociąga za sobą wiele problemów związanych ze spójnością danych. Projektowany przeze mnie system przechowuje dane w wielu kopiach, przez co konieczne jest stworzenie mechanizmów zapewniających spójność danych.

Wydajność ponad pełną spójność

Zapewnienie spójności w systemie rozproszonym wymaga wiele pracy i zwykle negatywnie odbija się na wydajności. W większości przypadków, jeżeli nie jest to konieczne, przyjmuje się rozluźniony model spójności, który upraszcza tworzenie systemu i znacznie poprawia jego wydajność. W moim systemie przyjmuję model **spójności ostatecznej** (ang. *eventual consistency*), który zakłada stopniowe uspójnianie danych jeżeli przez dłuższy czas nie będą dokonywane żadne modyfikacje. Ostatecznie wszystkie modyfikacje zostaną rozpropagowane na wszystkie repliki zasobu (pliku). Biorąc pod uwagę zastosowanie projektowanego systemu, taki model wydaje się dobrym kompromisem (zob. [15]).

Pojedynczy punkt awarii

Bardzo ważnym wymaganiem związanym z tolerowaniem awarii jest nieposiadanie w systemie **pojedynczego punktu awarii** (ang. *single point of failure*). Za punkt taki jest uważany dowolny element systemu, którego awaria uniemożliwi pracę z systemem. W projekcie należy unikać występowania takich punktów, przy jednoczesnym uwzględnieniu faktu, że całkowite ich wyeliminowanie może drastycznie pogorszyć efektywność. Z tego względu przyjęto rozwiązanie kompromisowe, które jest całkowicie wystarczające dla przewidzianych zastosowań systemu.

Konfiguracja stopnia zabezpieczeń

W wielu zastosowaniach systemów plików mamy do czynienia z różnymi typami danych. Niektóre z tych danych wymagają dużej dostępności, gdyż czytane są przez wielu klientów, inne nie są czytane prawie wcale. W niektórych sytuacjach dane mają bardzo małą użyteczność i nic wielkiego się nie stanie jeżeli „zgubimy" pewną część z nich. Jeżeli będziemy stosować ten sam poziom zabezpieczeń dla wszystkich typów danych, to będziemy niepotrzebnie marnować zasoby. Dlatego jednym z wymagań dla systemu jest możliwość konfigurowania stopnia, w jakim „dbamy" o dane. Chodzi o to, żeby klienci mogli indywidualnie decydować, czy zapisywany plik będzie przechowywany w jednej, dwóch czy dziesięciu kopiach w zależności od wymagań i przewidywanego obciążenia.

Skalowalność

Jest to jedno z podstawowych wymagań dla większości systemów rozproszonych. System powinien mieć możliwość ciągłego powiększania swojej pojemności i obsługi coraz większej liczby klientów poprzez dokładanie nowych komputerów. Wszelkie czynności związane ze zwiększaniem pojemości systemu (dokładanie nowych komputerów lub dysków twardych) powinny być całkowicie niezauważalne dla klientów pracujących z systemem. Operacja taka powinna być możliwa w każdym momencie, a nowe komputery (dyski) powinny przejąć część zadań umożliwiając równomierne obciążenie wszystkich węzłów. Oznacza to także, że system powinien potencjalnie umożliwiać obsługę tysięcy klientów, choć może się to wiązać z koniecznością dołączenia wielu nowych serwerów.

Proste API

Bardzo wysoki priorytet ma zdefiniowanie jak najprostszego interfejsu systemu udostępnianego aplikacjom klienckim. Podstawowe operacje na plikach, takie jak odczyt, zapis, generowanie listy plików, powinny być rozszerzone o inne, umożliwiające wykonywanie bardziej złożonych zapytań bez dużego wpływu na wydajność. Zdefiniowany zbiór zapytań ma umożliwiać wykonywanie wszystkich operacji wykorzystywanych przez serwisy internetowe.

Prosta obsługa

Ważnym czynnikiem przy korzystaniu z różnych systemów jest ich prosta obsługa. Należy zaprojektować system w taki sposób, aby jego obsługa ograniczała się do instalacji oprogramowania i podłączenia komputera do sieci. Wszelkie parametry pracy powinny być ustalane przez system dynamicznie, aby uniemożliwić wystąpienie pomyłki przy konfiguracji oraz lepiej dostosować system do zmiennych warunków środowiska pracy.

1.1.2. Docelowe środowisko pracy systemu

Jako docelowe środowisko wybrany został system operacyjny Linux działający na komputerach klasy PC. Komunikacja między węzłami będzie odbywać się z wykorzystaniem protokołów TCP/IP oraz UDP/IP. Pliki przechowywane będą na partycjach z systemem plików EXT3 (praca w trybie *ordered* – tylko kronikowanie metadanych – zastosowanie pełnego kronikowania nie będzie miału tutaj sensu ze względu na narzut związany z kopiowaniem danych do dziennika, a projektowany system będzie miał własne mechanizmy zabezpieczające przed utratą danych).

1.2. Architektura systemu

1.2.1. Logiczna struktura danych

Wszystkie dane w systemie będą pogrupowane w **pliki**. Każdy plik w systemie będzie składał się z **rekordów**. Rekord będzie stanowił najmniejszą porcję danych w systemie. Będzie to ciąg danych binarnych o dowolnej strukturze i dowolnej długości. Pliki będą stanowiły zbiory rekordów bez powtórzeń i bez żadnego ustalonego porządku.

Każdy plik w systemie będzie identyfikowany 64-bitowym kluczem (identyfikator pliku). Identyfikator pliku będzie unikatowy w całym systemie. Każdy rekord pliku będzie identyfikowany 64-bitowym kluczem (identyfikator rekordu) unikatowym w ramach jednego pliku.

Liczba rekordów w pliku będzie ograniczona przez zakres 64-bitowego identyfikatora rekordu. W praktyce będzie to ograniczenie nieosiągalne. Poza tym będzie także istniało ograniczenie związane z pojemnością dysku twardego komputera.

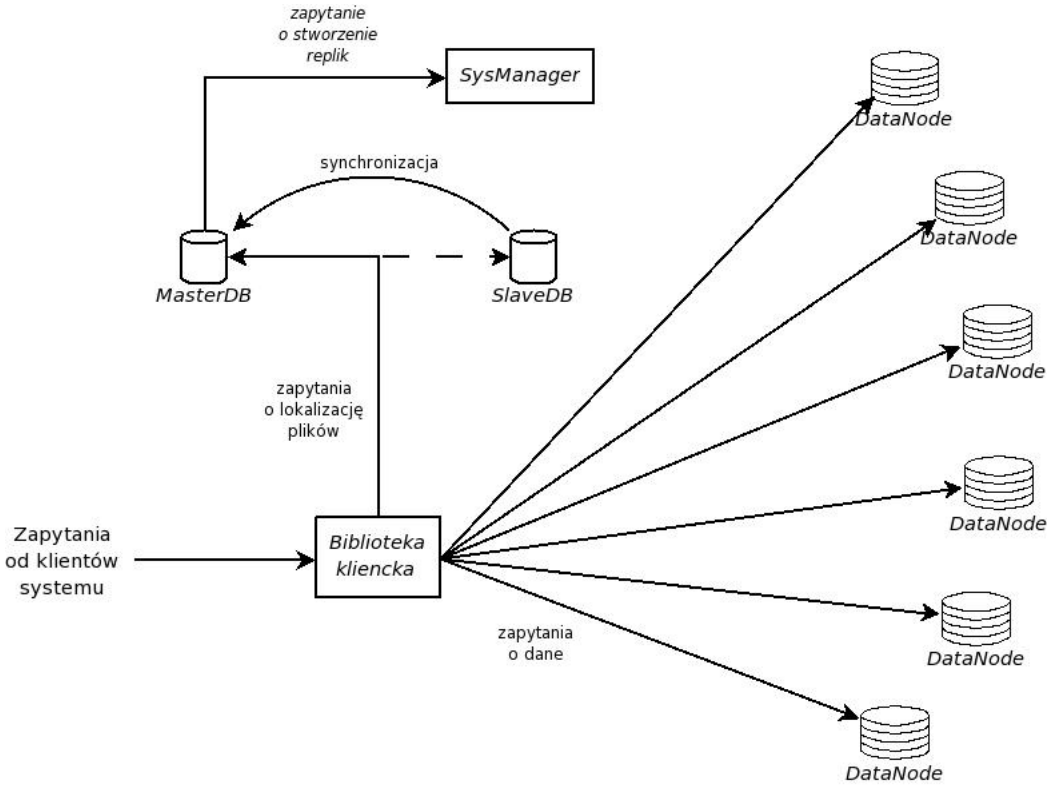
1.2.2. Układ danych w systemie

Dane w systemie będą przechowywane w serwerach danych *DataNode*. Każdy serwer danych będzie pracował na oddzielnym, dedykowanym komputerze. Każdy z plików będzie przechowywany w systemie w kilku kopiach (poziom replikacji). Liczba kopii będzie ustalana przy tworzeniu pliku. Każda z kopii (replik) pliku będzie przechowywana na innym serwerze danych. Lista komputerów, na których są przechowywane repliki pliku będzie tworzyć **lokalizację** pliku. Lokalizacje plików będą przechowywane w bazie danych lokalizacji *MasterDB*.

Lokalizacje dla plików będą generowane na podstawie aktualnego stanu systemu. Przy tworzeniu pliku będą wybierane komputery, które mają wystarczająco dużo miejsca na dysku i nie są zbyt obciążone. Za przydział komputerów i fizyczne utworzenie replik będzie odpowiadał serwer zarządzający systemem *SysManager*. Lokalizacja dla każdego pliku będzie tworzona niezależnie, na podstawie stanu systemu w momencie tworzenia pliku. Każda z replik pliku będzie przechowywać wszystkie rekordy pliku.

1.3. Obsługa zapytań klienta

Na rysunku 1.1 przestawione zostały główne elementy systemu związane z obsługą zapytań klienta.



Rysunek 1.1: Podsystem obsługi zapytań klienckich

Podstawowym elementem systemu będzie zbiór komputerów z serwerami danych (*DataNode*). Na tych komputerach będą przechowywane wszystkie pliki systemu. Aby możliwe było wyszukiwanie komputerów, na których znajduje się plik, klienci będą zadawali zapytania do bazy danych lokalizacji (*MasterDB*). Baza danych lokalizacji będzie przechowywała informacje o wszystkich plikach w systemie. Tworzenie nowego pliku w systemie będzie wykonywane poprzez bazę danych lokalizacji. Przydziałem i tworzeniem replik dla nowego pliku będzie zajmował się serwer zarządzający (*SysManager*). Zapisy i odczyty rekordów będą odbywały się poprzez bezpośrednią komunikację klienta z serwerem danych przechowującym replikę pliku. Najpierw klient pobierze lokalizację z bazy danych lokalizacji, a następnie będzie wykonywał operacje na pliku i jego rekordach poprzez bezpośrednią komunikację z replikami.

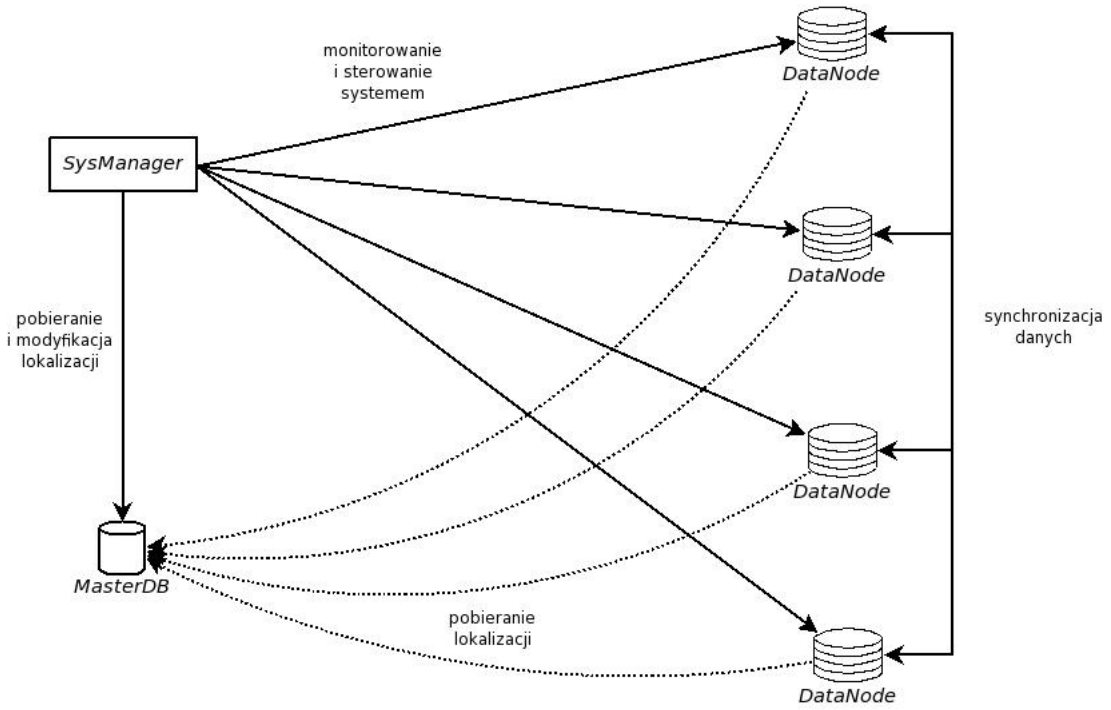
1.3.1. Awarie bazy danych lokalizacji

Baza danych lokalizacji będzie pojedynczym punktem awarii systemu. Będzie także wąskim gardłem przy komunikacji klientów z systemem. Aby rozwiązać te problemy do systemu dodane zostaną kopie bazy danych lokalizacji (*SlaveDB*). Kopie bazy danych będą odpowiadać na zapytania klientów o lokalizację plików w sytuacji awarii lub przeciążenia głównej bazy danych (linia przerywana na rysunku 1.1). Kopie bazy danych będą się synchronizowały z główną bazą danych poprzez protokół synchronizacyjny.

1.4. Utrzymanie systemu

W rozproszonych systemach składających się z wielu węzłów naturalną rzeczą jest występowanie awarii. Z tego względu każdy system rozproszony powinien być na takie awarie przygotowany i mieć możliwość pracy pomimo występowania awarii. W projektowanym systemie będzie zaimplementowany podsystem utrzymywania, który będzie monitorował pracę całego systemu i podejmował działania mające na celu zminimalizowanie zagrożenia dla danych wynikającego z awarii lub przeciążeń sprzętu.

Rysunek 1.2 przedstawia główne elementy podsystemu utrzymania.



Rysunek 1.2: Podsystem utrzymania systemu

Podsystem utrzymania składać się będzie z kilku elementów. Podstawowym z nich będzie serwer zarządzający *SysManager*, który pełni rolę nadzorcy. Głównym zadaniem serwera zarządzającego, poza przydzielaniem replik dla nowych plików, jest monitorowanie pracy całego systemu i wykonywanie operacji mających zapewnić wysoką dostępność i bezpieczeństwo danych.

1.4.1. Awarie komputerów

Serwer zarządzający będzie monitorował stan wszystkich komputerów w systemie. Jeżeli któryś z komputerów ulegnie awarii, wszystkie pliki, które się na nim znajdują zostana przeniesione na inny komputer (skopiowane z innych replik pliku). Kopiowanie danych będzie wykonywane bezpośrednio między serwerami danych, aby uzyskać większą wydajność i nie przeciążać łącza, jednak za samo tworzenie replik i wymuszanie ich synchronizacji będzie odpowiedzialny serwer zarządzający.

1.4.2. Modyfikacja lokalizacji

Serwer zarządzający, wykonując swoje zadania, będzie modyfikował w bazie danych lokalizacji informacje o położeniu plików. Aby serwer danych mógł usunąć pliki, których już nie ma przechowywać, będzie musiał cyklicznie odpytywać bazę danych lokalizacji o listę plików, za które jest odpowiedzialny (linia przerywana na rysunku 1.2). Jeżeli na liście komputerów z replikami pliku nie będzie adresu serwera danych, a przechowuje on dane pliku, to wszystkie rekordy tego pliku zostaną skopiowane do replik z listy po czym dane zostaną usunięte z dysku lokalnego komputera. Jeżeli dane pliku będą kopiowane z kilku komputerów do jednego, to komputer docelowy sam rozwiąże konflikty poprzez przyjęcie tylko pierwszego zapisu i odrzucenie pozostałych (identyfikatory rekordów w pliku będą unikatowe).

1.5. Biblioteka kliencka

Dostęp do systemu realizowany będzie przez bibliotekę kliencką zapewniającą metody do realizacji wszystkich dostępnych dla klientów operacji.

1.5.1. API systemu

Wszystkie wykonywane w systemie operacje będą operacjami atomowymi. System nie będzie udostępniał mechanizmu transakcji. Zmiany dokonywane przez jednego klienta mogą być widoczne natychmiast przez innych (z uwzględnieniem przyjętego modelu spójności).

Z punktu widzenia klienta systemu możemy zdefiniować dwie grupy operacji:

- operacje na plikach (tworzenie i usuwanie),
- operacje na rekordach (zapis, odczyt, usuwanie, listowanie).

Operacje na plikach

Jedną z operacji na plikach będzie **utworzenie pliku**. Utworzenie pliku nie będzie wymagało komunikacji klienta z serwerami danych, a jedynie z bazą danych lokalizacji plików. Podczas tworzenia nie będą wysyłane żadne dane pliku – jego obecność będzie jedynie rejestrowana w systemie co umożliwia późniejszy zapis rekordów z danymi. Tworzenie pliku będzie wymagało podania identyfikatora pliku, który musi być unikatowy w systemie. Identyfikatory dla plików powinny być generowane w aplikacji zewnętrznej z uwzględnieniem wymagania niepowtarzalności. Najłatwiej jest to uzyskać poprzez generowanie identyfikatorów sekwencyjnych z dołączeniem znacznika czasu.

Kolejną operacją udostępnianą przez system będzie operacja **usuwania pliku**. Usuwanie również będzie przeprowadzane jedynie poprzez komunikację z bazą danych lokalizacji. Operacja ta wykonywana będzie asynchronicznie. Klienci już połączeni z serwerem danych będą mogli korzystać z danych pliku aż do zakończenia zapytania. Usuwanie pliku zostanie

odłożone w czasie, aż do momentu gdy serwer danych zada zapytanie do bazy danych lokalizacji o położenie pliku – będzie robił to cyklicznie dla wszystkich plików, za które będzie odpowiedzialny. W odpowiedzi na takie zapytanie baza danych zwróci informacje, że plik nie istnieje już w systemie, a serwer danych będzie mógł usunąć jego dane z dysku.

Operacje na rekordach

Większość zapytań z aplikacji klienckich do systemu będzie dotyczyła operacji wykonywanych na rekordach. Udostępnionych zostanie kilka prostych operacji, które umożliwią spełnienie wszystkich wymagań stawianych przed systemem w kontekście przedstawionych wcześniej zastosowań.

Zapis rekordu do pliku będzie jedną z podstawowych operacji wykonywanych przez klientów. Zapis będzie odbywać się bezpośrednio z aplikacji klienckiej do wszystkich serwerów danych przechowujących replikę pliku (po uprzednim uzyskaniu informacji o lokalizacji replik z bazy danych lokalizacji). Rekordy będą synchronicznie zapisywane do jednej z replik (najmniej obciążonej), a następnie asynchronicznie przesyłane do pozostałych replik. Takie rozwiązanie pozwala na niewielkie obciążenie kanałów komunikacji z klientem oraz na lepsze wykorzystanie sieci komunikacyjnej pomiędzy serwerami danych (która zwykle ma większą wydajność, dzięki bliskości fizycznej sprzętu, niż sieć łącząca z klientami systemu).

Odczyt rekordu będzie również wykonywany bezpośrednio. Po pobraniu lokalizacji z bazy danych lokalizacji, wybrana zostanie replika o najmniejszym obciążeniu, i do niej zostanie wysłane zapytanie o dane rekordu. Jeżeli odczyt z tej repliki się nie powiedzie, z powodu awarii, przeciążenia lub braku danych, zapytanie zostanie wysłane do kolejnej repliki w kolejności rosnącego obciążenia komputerów. Oczywiście możliwy będzie także odczyt fragmentu rekordu.

Usuwanie rekordu polega na usunięciu wybranego rekordu z pliku. Polecenie będzie wykonywane bezpośrednio na serwerach danych. Rekord nie zostanie natychmiast usunięty, ale zostanie zaznaczony jako *do usunięcia*. Odczyt takiego rekordu nie będzie możliwy. Ostateczne usunięcie rekordu nastąpi gdy wszystkie repliki pliku będą miały ten rekord oznaczony jako *do usunięcia*. Możliwa jednak będzie sytuacja, w której polecenie usunięcia rekordu nie dotrze do wszystkich replik i przez to będzie można odczytać rekord (wynika to z przyjętego modelu spójności).

System nie będzie udostępniał operacji modyfikacji zapisanego rekordu. Rekord będzie mógł być usunięty i zapisany ponownie, jednak dopiero po ostatecznym jego usunięciu ze wszystkich replik. Zamiast usuwania lepiej będzie tworzyć nowe rekordy, a w aplikacjach wyższego poziomu zastępować dowiązania do starych rekordów nowymi.

1.6. Konflikty

System nie będzie udostępniał żadnego mechanizmu transakcji. Z tego powodu przy zapisie danych do systemu będą mogły wystąpić konflikty. Konflikt przy próbie utworzenia pliku w systemie będzie rozwiązywany przy pomocy bazy danych lokalizacji. Sytuacja będzie bardziej skomplikowana przy zapisie rekordów do pliku. Jeżeli dwa procesy klienckie będą równocześnie próbowały zapisać rekord o tym samym identyfikatorze do pliku, to będziemy mieli do czynienia z problemem wyścigu. Może dojść do sytuacji, w której obydwie wersje rekordu zostaną zapisane, ale zachowanie systemu przy próbie odczytu rekordu będzie niezdefiniowane (system może zwracać różne dane). Podobna sytuacja będzie miała miejsce, gdy niektóre repliki pliku nie będą w pełni zsynchronizowane (nie będą przechowywały wszystkich zapisanych do pliku rekordów).

Prawdopodobieństwo takiej sytuacji może zostać zminimalizowane przez przeprowadzenie przed zapisem sprawdzenia czy rekord o podanym identyfikatorze istnieje już w pliku, jednak nie rozwiązuje to w pełni problemu – nadal może dojść do wyścigu.

Problem ten wynika z przyjętego modelu spójności i jest bardzo trudny do rozwiązania. Zamiast go rozwiązywać, zwiększając znacznie narzut na wykonanie każdej operacji, w systemie zastosowane zostanie inne podejście. Aplikacje klienckie same będą generowały identyfikatory rekordów tak, aby były one unikatowe w skali pliku (można to zrobić choćby uzależniając identyfikator rekordu od znacznika czasu wykonania operacji i skrótu generowanego na podstawie danych rekordu). Dzięki tekiemu podejściu, które jest dość proste do zaimplementowania, nie będzie potrzeby konstruowania złożonych algorytmów zapewniających spójność danych kosztem dużego narzutu (algorytmy te zwykle korzystają z pewnego centralnego punktu do rozwiązywania konfliktów, który automatycznie staje się pojedynczym punktem awarii systemu).

1.7. Tolerowanie awarii

Aby umożliwić tolerowanie awarii, w systemie zostanie zastosowane zwielokrotnianie danych (replikacja plików). Wszystkie dane będą przechowywane w wielu kopiach (stopień replikacji będzie można ustalić przy tworzeniu pliku w systemie). W przypadku awarii jednej z replik pliku zapytania będą kierowanie do innych replik. Część awarii w systemie będzie chwilowa i ponownie uruchomiony komputer będzie nadal w pełni sprawnym elementem systemu. Jednak większość awarii wynika z uszkodzeń (logicznych lub fizycznych) nośników danych. Tego typu awarie nie mogą być naprawione automatycznie lub szybko, dlatego konieczne jest wprowadzenie mechanizmów umożliwiających normalną pracę systemu *pomimo* ich występowania. W tym celu będzie wprowadzony mechanizm utrzymania systemu (nadzorowany przez serwer zarządzający) opisany w punkcie 1.4. Więcej o tolerowaniu awarii w systemach rozproszonych można znaleźć w [8].

1.8. Pojedynczy punkt awarii

W systemie występuje pojedynczy punkt awarii, którym będzie baza danych lokalizacji plików. Będzie ona stanowić także wąskie gardło w komunikacji z klientami systemu. Każde zapytanie o dane pliku musi być poprzedzone zapytaniem o jego lokalizację. Aby uniezależnić system od awarii bazy danych lokalizacji, wprowadzony zostanie mechanizm umożliwiający korzystanie z kopii bazy danych lokalizacji. Kopie te będą pracować na oddzielnych komputerach i, w przypadku awarii lub przeciążenia głównej bazy danych, będą przyjmowały zapytania o lokalizację plików od klientów systemu.

Kopie bazy danych lokalizacji muszą się synchronizować z główną bazą danych lokalizacji, aby przechowywać aktualne dane o lokalizacji plików. Będzie to realizowane z wykorzystaniem specjalnie zaprojektowanego protokołu synchronizacyjnego, który będzie wykorzystywał sumy kontrolne i powodował niewielki narzut na funkcjonowanie systemu.

Dodatkowo, lokalizacje plików będą modyfikowane dość rzadko (głównie w czasie awarii sprzętu), więc nawet nieco starsze dane lokalizacyjne będą umożliwiały klientom poprawne wykonywanie operacji na plikach systemu.

1.9. Spójność

Przyjęty w projektowanym systemie model spójności jest modelem rozluźnionym. Dzięki temu możliwe jest znaczące zwiększenie efektywności pracy systemu. Jednak takie podejście ma pewne minusy. W niektórych zastosowaniach wymagane jest utrzymywanie pełnej spójności danych i w takich zastosowaniach projektowany system nie będzie mógł działać. W serwisach internetowych zwykle jednak główny nacisk położony jest na wydajność pracy systemu, a *spójność ostateczna* jest modelem wystarczającym.

W systemie będzie istniało kilka elementów wspierających zapewnienie spójności ostatecznej. Podstawowym takim elementem jest serwer zarządzający, który będzie stale monitorował stan systemu (komputerów) oraz poszczególnych plików. Wszelkie niespójności danych będą wykrywane i usuwane poprzez odpowiednie operacje synchronizacyjne. Drugim elementem są serwery danych, które same będą monitorować stan replik przechowywanych przez siebie plików i, w miarę możliwości, usuwać niespójności w plikach. Ostatnim mechanizmem, który pomaga w utrzymaniu spójnego obrazu systemu jest biblioteka kliencka. Konstrukcja wewnętrznych operacji na danych będzie korygować chwilowe niespójności danych poprzez odpytywanie wielu replik plików w celu wykonania operacji.

Rozdział 2

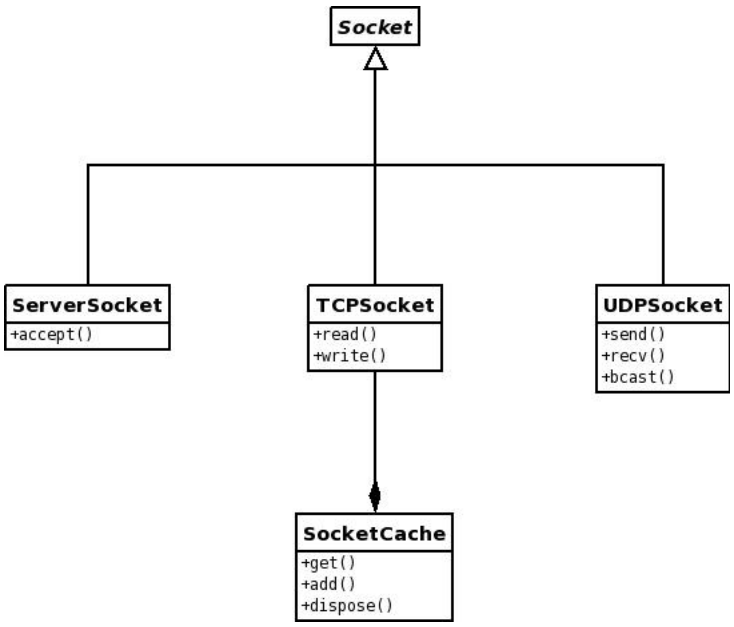
Projekt implementacji

W tym rozdziale opiszę główne elementy implementacji systemu, wspólne dla wszystkich części projektu. Zacznę od opisu obiektów odpowiedzialnych za komunikację w systemie, następnie przedstawię klasy narzędziowe współdzielone przez wszystkie moduły systemu, a na końcu dokładnie omówię architekturę każdego z elementów systemu.

2.1. Komunikacja

2.1.1. Obiekty komunikacyjne

Rysunek 2.1 przedstawia diagram klas głównych obiektów gniazd komunikacyjnych systemu wraz z podstawowymi metodami do realizacji zapytań.



Rysunek 2.1: Diagram klas obiektów gniazd komunikacyjnych

Klasa *Socket*

Klasa *Socket* będzie abstrakcyjną klasą bazową dla innych klas gniazd wykorzystywanych przy komunikacji. Klasa ta będzie zawierać implementacje metod umożliwiających wykona-

nie wielu niskopoziomowych operacji na gniazdach sieciowych (np. ustawiane nieblokującego trybu pracy, limitowanie czasu operacji wejścia-wyjścia na gniazdach i inne związane z obsługą sieci).

W sieciach komunikacyjnych często dochodzi do zakłóceń, przy których wiele przesyłanych pakietów może zostać zgubionych. W takich sytuacjach serwer może bezskutecznie oczekiwać na dane przez bardzo długi czas (jest to czasami wykorzystywane w atakach typu DDoS). Jeżeli wszystkie wątki serwera będą oczekiwać na dane, to serwer będzie niedostępny dla klientów zewnętrznych. Podobny problem występuje wówczas, gdy bufor gniazda połączenia nie jest opróżniany (wynika to z realizacji przesyłania danych, przy której nie przychodzą potwierdzenia dostarczenia pakietu do odbiorcy).

Aby zapobiec takim sytuacjom każda operacja w systemie rozproszonym musi zostać wyposażona w limit czasu wykonania. Jeżeli limit ten zostanie osiągnięty, to operacja kończy się błędem przekroczenia czasu wykonania.

Klasa *ServerSocket*

Obiekty klasy *ServerSocket* reprezentują gniazda serwerów i odpowiadają za przyjmowanie połączeń od klientów poprzez wywołanie metody *accept()*. W wyniku wywołania tworzony będzie obiekt klasy *TCPSocket* (patrz następny punkt), którym będziemy posługiwać się przy komunikacji z klientem serwera.

Klasa *TCPSocket*

Klasa *TCPSocket*, jak wszystkie inne klasy gniazd, będzie dziedziczyć po klasie *Socket*. Obiekty klasy będą reprezentować gniazda połączeń TCP/IP w systemie i odpowiadać za strumieniową transmisję danych. Udostępnione metody umożliwią zapis i odczyt danych z gniazda systemowego. Metody te będą oczywiście wyposażone w limit czasu wykonania, aby sprostać problemom opisanym wcześniej.

Klasa *SocketCache*

Projektowany system będzie działał na dużej liczbie komputerów. Połączenia między tymi komputerami będą wykonywane bardzo często. Zainicjowanie połączenia w protokole TCP/IP (ang. *handshake*) jest procesem dość kosztownym. Jeżeli będziemy inicjować połączenia zbyt często, to odbije się to na wydajności całego systemu. Z tego względu trzeba dążyć do maksymalnego wykorzystania połączeń już nawiązanych. Obiekt klasy *SocketCache* przechowuje w strukturach wewnętrznych obiekty *TCPSocket* i umożliwia ich wielokrotne wykorzystanie bez konieczności przeprowadzania procesu łączenia.

Klasa *UDPSocket*

Obiekty klasy *UDPSocket* będą umożliwiały wysyłanie pakietów przy użyciu protokołu UDP/IP. Metody klasy pozwolą na wysłanie i odbiór datagramów UDP. Dodatkowo klasa będzie udostępniać metody do wysyłania pakietów w trybie rozgłoszeniowym (ang. *broadcast*). Metody te będą wykorzystywane przez moduły monitorujące serwerów danych w celu powiadamiania wszystkich węzłów o stanie systemu.

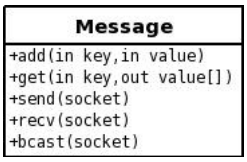
2.1.2. Protokół komunikacyjny

Komunikacja pomiędzy elementami systemu będzie wykorzystywała protokoły TCP/IP oraz UDP/IP warstwy transportowej. Aby umożliwić przesyłanie złożonych struktur i typów da-

nych, stworzona zostanie abstrakcyjna warstwa komunikacyjna oparta na obiektowym modelu komunikatów.

Protokół komunikacyjny jest elementem wspólnym dla wszystkich programów pracujących w ramach systemu, powinien więc umożliwiać przesyłanie wszelkich informacji koniecznych do zapewnienia poprawnej pracy systemu.

Aby to zrealizować, stworzone zostaną obiekty reprezentujące komunikaty w systemie, które następnie będą przesyłane pomiędzy programami przy pomocy gniazd protokołów TCP/IP oraz UDP/IP.



Rysunek 2.2: Klasa komunikatu protokołu komunikacyjnego

Obiekty komunikatów (klasa *Message*) będą udostępniały operacje słownikowe. Operacja *add(key,val)* doda do obiektu wartość *val* pod klucz *key*. Z jednym kluczem będzie można powiązać wiele wartości. Operacja *get(key)* zwraca listę wartości powiązanych z podanym kluczem w obiekcie komunikatu. Metody *send()*, *recv()* oraz *bcast()* będą służyły do przesyłania komunikatów poprzez gniazda odpowiednich protokołów.

Protokół będzie umożliwiał przesyłanie wszystkich podstawowych typów danych, a także ciągów danych binarnych o dowolnej długości. Dzięki temu będzie można przesyłać wszystkie informacje pomiędzy węzłami systemu.

2.2. Pomocnicze klasy narzędziowe

W kodzie programów systemu wykorzystywanych jest wiele dodatkowych klas. W tym podpunkcie wymienię kilka z nich, do których będę się odnosił w dalszej części rozdziału.

2.2.1. Wątki i synchronizacja

Klasa *Thread*

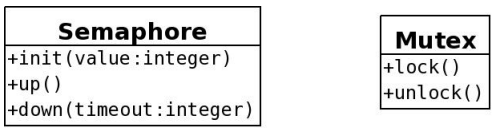
Większość oprogramowania działającego w ramach projektowanego systemu to serwery wielowątkowe. Wprowadzenie klasy bazowej wątku (ang. *thread*) ułatwi późniejszą konstrukcję klas pochodnych dla różnych serwisów.

Rysunek 2.3: Klasa wątku

Abstrakcyjna klasa *Thread* będzie deklarować dwie metody. Metoda *start()* uruchomi wątek systemowy do obsługi obiektu. Metoda *run()* będzie metodą abstrakcyjną, która musi zostać przedefiniowana w klasach pochodnych. Kod tej metody będzie wykonywany przez wątek sytemowy.

Semafory i muteksy

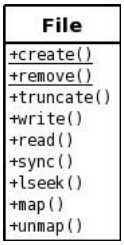
Ze względu na współistnienie wielu wątków w programach konieczne będzie wprowadzenie mechanizmów synchronizacyjnych. Dwie podstawowe klasy służące do tego celu to *Semaphore* i *Mutex*. Klasa *Semaphore* będzie realizować mechanizm synchronizacji z wykorzystaniem nienazwanych semaforów POSIX. Obiekty klasy *Mutex* będą służyć do synchronizacji dostępu do danych współdzielonych przez wątki. Metody definiowane przez obie klasy będą odpowiadać standardowym operacjom wykonywanym na tych strukturach (zob. [2]).



Rysunek 2.4: Obiekty synchronizacyjne

Dostęp do plików dyskowych

Dostęp do plików dyskowych będzie realizowany poprzez obiekty klasy *File*. Klasa wraz z metodami jest przedstawiona na rysunku 2.5.



Rysunek 2.5: Pliki dyskowe

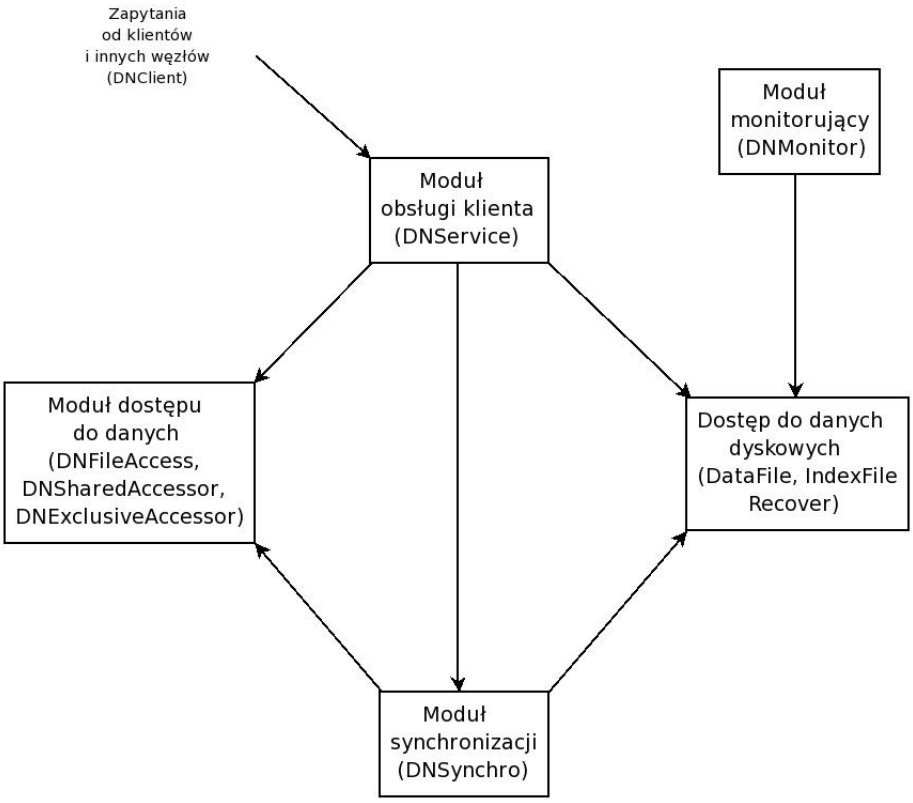
Funkcja statyczna *create()* odpowiadać będzie za tworzenie nowych plików w systemie operacyjnym. Funkcja *remove()* będzie usuwać pliki. *truncate()* pozwala na zmianę wielkości plików. Funkcja *write()* będzie służyć do zapisu danych do pliku, a *read()* do ich odczytu. Funkcja *sync()* pozwalać będzie na wymuszenie, aby strony zmodyfikowane w pamięci podręcznej stron zostały zapisane na dysk (wywołanie systemowe *fsync()* oraz *msync()* jeżeli plik jest mapowany w pamięci). Wywołania metod *map()* oraz *unmap()* będą pozwalały na wykorzystanie mapowania plików dyskowych w pamięci (wywołania systemowe *mmap()* oraz *munmap()*). Więcej o mechanizmach wydajnych operacji wejścia-wyjścia w systemie Linux, z których w systemie będziemy intensywnie korzystać, można przeczytać w [9] oraz w artykule [17].

2.3. Serwer danych *DataNode*

W tym punkcie zostanie opisany projekt implementacji serwera danych. Na początku zostanie przedstawiona ogólna architektura serwera, następnie przedstawie wątki wykonania w serwerze, a na końcu opiszę struktury danych aplikacji w plikach dyskowych oraz interakcje z systemem plików.

2.3.1. Architektura serwera danych

Rysunek 2.6 przedstawia ogólną architekturę serwera danych wraz z klasami wchodzącymi w skład każdego z modułów.



Rysunek 2.6: Architektura serwera danych

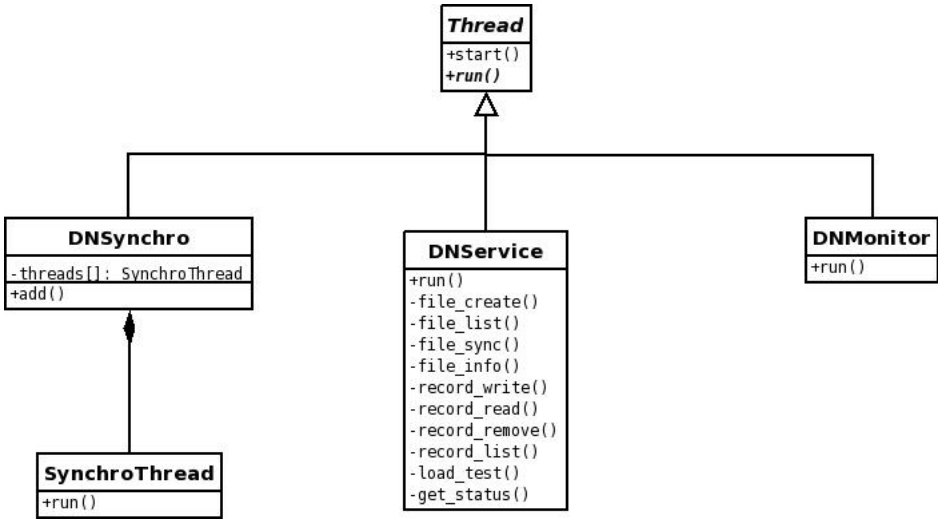
Komunikacja z serwerem danych będzie zrealizowana poprzez obiekty klasy *DNClient*. Każdy obiekt tej klasy będzie udostępniał metody służące do komunikacji z serwerem danych. Zapytania te będą przesyłane przez sieć komunikacyjną z wykorzystaniem protokołu TCP/IP. Za obsługę zapytań po stronie serwera będą odpowiadać obiekty klasy *DNService*. Każdy z obiektów tej klasy będzie dziedziczyć po klasie *Thread* i dzięki temu obsługa zapytań każdego klienta będzie wykonywana w oddzielnym wątku systemu operacyjnego. W serwerze zostanie zaalokowana stała liczba obiektów klasy *DNService* i każdemu klientowi łączącemu się z serwerem zostanie przydzielony obiekt do obsługi zapytań. Dzięki temu nie dopuścimy do przeciążenia serwera zbyt dużą liczbą równoległych klientów. W funkcji głównej (metoda *run()* obiektu *DNService*) serwer oczekuje na komunikaty od klienta. Po otrzymaniu komunikatu serwer wykona odpowiednie zadanie. Jeżeli zadanie polega na dostarczeniu danych pliku, to serwer wykorzysta metody klasy *DNFileAccess*, aby uzyskać dostęp do odpowiednich danych. Jeżeli zadanie dotyczy synchronizacji pliku (np. polecenia wysłane przez serwer zarządzający), to uruchomiona zostanie odpowiednia metoda obiektu klasy *DNSynchro*.

Obiekt klasy *DNFileAccess* reprezentuje strukturę w pamięci serwera danych, która będzie służyć do zarządzania dostępem do danych dyskowych. Aby uzyskać dostęp do danych pliku, będzie wykorzystywany obiekt typu *DNSharedAccessor* lub *DNExclusiveAccessor*. Będziemy tu mieli do czynienia z dostępem w trybie współdzielonym (np. przy operacji odczytu lub zapisu rekordu) lub wyłącznym (przy synchronizacji zawartości pliku z innymi replikami).

Po uzyskaniu dostępu w odpowiednim trybie, będziemy mogli korzystać z danych plików systemowych poprzez obiekty klas *DataFile* i *IndexFile* (obiekt klasy *Recover* będzie wykorzystywany do odzyskiwania danych po awarii serwera).

2.3.2. Wątki wykonania serwera danych

Na rysunku 2.7 przedstawione są klasy reprezentujące wątki serwera danych. Wszystkie wątki serwera należą do jednej z trzech grup. Pierwsza z nich obejmuje obiekty odpowiedzialne za realizację zapytań użytkowników. Do drugiej grupy należą wątki realizujące operacje synchronizujące zawartość pliku z innymi jego replikami. W ostatniej grupie znajduje się jeden wątek odpowiedzialny za monitorowanie stanu komputera i rozsyłanie informacji do innych węzłów systemu.



Rysunek 2.7: Wątki serwera danych

Klasa *DNService*

Obiekty klasy *DNService* będą realizować zapytania klientów serwera danych. W funkcji głównej *run()* obiektu serwer będzie oczekiwał na kolejne komunikaty od klienta. Następnie sterowanie będzie przekazywane do jednej z funkcji obsługujących zapytania. Funkcje z prefiksem „*file_*" będą realizować zapytania związane z plikami systemu. *file_create()* będzie odpowiadać za tworzenie pliku w serwerze danych, *file_list()* za generowanie listy plików przechowywanych w serwerze, a *file_info()* za informacje o konkretnym pliku. Funkcja *file_sync()* będzie realizować specjalne polecenia wysyłane przez serwer zarządzający, które będą wymuszały synchronizację replik pliku.

Funkcje z przedrostkiem „*record_*" będą realizować operacje związane z rekordami. *record_write()* będzie zapisywać rekord w pliku, *record_read()* będzie odpowiadać za odczyt danych rekordu, *record_remove()* będzie usuwać rekord z pliku, a *record_list()* generować listę rekordów zapisanych w pliku.

Dodatkowo w obiektach klasy *DNService* będą występowały dwie metody specjalne. *load_test()* będzie realizować kod specjalnego zapytania klienta, w którym wykonywanych jest kilka operacji dyskowych. Biblioteka kliencka (w obiektach klasy *DNClient*) będzie mierzyć czas wykonania tej operacji. Dzięki temu klienci będą mogli ocenić obciążenie komputera, na którym pracuje serwer danych.

Metoda *get_status()* będzie służyła do pobierania zajętości dysku serwera danych. Będzie to wykorzystywane w serwerze zarządzającym do rozpoznawania komputerów, na których kończy się miejsce na dysku twardym.

Klasa *DNSynchro*

Obiekt klasy *DNSynchro* będzie realizował działania związane z synchronizacją plików pomiędzy replikami. Wewnątrz obiektu tej klasy zostanie zaimplementowana kolejka zadań. Zadania do kolejki będą dodawane poprzez wywołanie metody *add()*. Zadania dodane do kolejki będą przetwarzane przez obiekty klasy wewnętrznej reprezentujące wątki wykonujące synchronizację (klasa *SynchroThread*). Obiekty te będą tworzone w konstruktorze *DNSynchro*. Gdy w kolejce zadań znajdzie się polecenie synchronizacji pliku, jeden z wątków wykona synchronizację pliku podanego w parametrach zadania. Po wykonaniu zadania wątek wróci do oczekiwania na kolejne zadania.

Wątek reprezentowany przez obiekt *DNSynchro* będzie miał jeszcze inne zadania. Jeżeli serwer zarządzający systemu przeniesie plik z jednego serwera danych na inny (np. po awarii) lub jeżeli klient usunie plik z systemu, to ta sytuacja zostanie wykryta właśnie w tym wątku. Jeżeli replika pliku zostanie przeniesiona na inny komputer, to rekordy pliku zostaną tam skopiowane. Jeżeli plik zostanie usunięty, to wątek *DNSynchro* usunie wszystkie dane pliku z lokalnego komputer (serwera danych).

Klasa *DNMonitor*

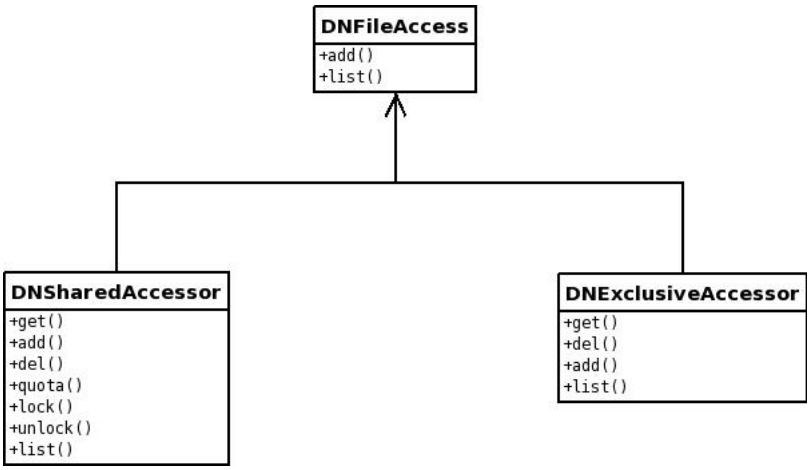
Obiekt klasy *DNMonitor* będzie odpowiedzialny za monitorowanie zajętości dysku twardego, na którym przechowywane są dane serwera. Wątek wewnętrzny obiektu będzie wykonywał kod funkcji głównej *run()*. W funkcji tej, w pętli nieskończonej, pobierane będą informacje o zajętości dysku twardego, a następnie będą one przesyłane (z wykorzystaniem protokołu UDP/IP) do serwera zarządzającego. Działanie tego wątku będzie sygnałem dla serwera zarządzającego, że komputer, z którego wysyłane są pakiety, ma uruchomiony serwer danych (i że w ogóle jest częścią systemu – dzięki temu będzie można łatwo dodawać nowe komputery do systemu bez dopisywania ich do konfiguracji serwera zarządzającego – wystarczy uruchomić na nim wątek monitorujący).

2.3.3. Synchronizacja dostępu do danych

Aby wątki obsługujące zapytania klienckie mogły korzystać z danych zapisanych na dyskach serwera, trzeba wprowadzić mechanizm synchronizacji dostępu. W przeciwnym razie równoległy dostęp do danych plików mógłby doprowadzić do nieprzewidzianych rezultatów. Zadanie to będą realizowały klasy zawarte w module dostępu do danych. Rysunek 2.8 przedstawia klasy związane z synchronizacją dostępu.

Klasa *DNFileAccess*

Będzie to najważniejsza klasa modułu. Jej jedyny obiekt będzie utrzymywał pamięć podręczną informacji zawartych w plikach indeksowych (rozmiary rekordów i ich położenie w pliku z danymi) oraz informacje o blokadach założonych na pliki w serwerze. Informacje o położeniu rekordów w plikach będą zapisywane do obiektu tej klasy w procedurze uruchomienia serwera. Następnie każde zapytanie klienckie odnoszące się do danych pliku będzie mogło korzystać z tych informacji bez konieczności odczytywania ich z dysku z pliku indeksowego.



Rysunek 2.8: Klasy synchronizacji dostępu

Klasa udostępniać będzie bardzo prosty interfejs dostępu do danych. Metoda *add()* obiektu będzie realizować procedurę dodania do pamięci podręcznej serwera informacji o nowym pliku. Metoda *list()* będzie pozwalała na wygenerowanie listy plików. które są przechowywane w pamięci podręcznej obiektu (w serwerze danych). Pozostałe funkcje dostępowe będą realizowane poprzez obiekty pośrednie, po założeniu blokady na dane pliku.

Klasa *DNSharedAccessor*

Obiekty klasy *DNSharedAccessor* będą umożliwiały założenie na dane pliku blokady dzielonej. W konstruktorze obiektu będzie wywoływana funkcja zapisująca informacje o blokadzie na konkretnym pliku do obiektu klasy *DNFileAccess*, który tymi blokadami zarządza. Blokada dzielona będzie umożliwiać równoczesne korzystanie z danych pliku przez wiele wątków czytających, a także zapisujących rekordy. Dodatkowo w obiekcie tej klasy wprowadzone będą dwie metody pozwalające zrealizować sekcję krytyczną przy dostępie do metadanych pliku (na dysku i w obiekcie głównym *DNFileAccess*). Po wywołaniu metody *lock()* wątek wykonujący będzie miał wyłączny dostęp do metadanych i dzięki temu będzie mógł alokować nowe bloki na dysku i modyfikować pamięć podręczną pliku. Takie rozwiązanie pozwala na optymalną realizację operacji na pliku. Szybkie operacje wymagające wyłącznego dostępu do metadanych będą wykonywane pod blokadą wyłączną, a długo trwające operacje wejścia-wyjścia pod blokadą dzieloną. Metody *get()*, *add()*, *del()* oraz *list()* będą służyć do pobierania, dodawania, usuwania oraz listowania informacji o rekordach pliku. Metoda *quota()* będzie umożliwiała pobranie sumarycznej wielkości pliku (wszystkich rekordów). Ponieważ metody te będą korzystały z pamięci podręcznej w obiekcie *DNFileAccess*, konieczne jest, aby były wywoływane przy założonej blokadzie wyłącznej. Aby zakończyć korzystanie z pliku w trybie wyłączności, będzie wywoływana metoda *unlock()*. Wtedy inne wątki klienckie oczekujące na dostęp do danych będą mogły wykonać swoje operacje (w trybie wyłączności).

Klasa *DNExclusiveAccessor*

Obiekty tej klasy będą pełniły podobne funkcje do obiektów klasy *DNSharedAccessor*. Różnicą jest tryb dostępu do danych (rodzaj zakładanej blokady). Metody *add()*, *get()*, *del()* oraz *list()* będą miały analogiczne zadania jak poprzednio. Stworzenie blokady wyłącznej poprzez stworzenie tego obiektu, będzie powodowało, że żadna inna blokada, wyłączna ani dzielona,

nie będzie mogła być założona na pliku. Ten typ blokady będzie wykorzystywany w wątkach synchronizujących zawartość pliku z innymi replikami. Próba założenia innej blokady na tym samym pliku w tej samej instancji serwera będzie kończyła się błędem (nie będzie oczekiwania na zwolnienie blokady). Takie podejście umożliwia wątkom klienckim serwera zwrócenie informacji o założonej blokadzie, aby klienci mogli zadać to samo zapytanie do innej repliki pliku bez oczekiwania na zwolnienie blokady.

2.3.4. Organizacja danych w plikach dyskowych

Dane każdego pliku, który jest przechowywany w systemie, będą przechowywane w dwóch zwykłych plikach dyskowych systemu operacyjnego. Jeden z plików, plik danych, będzie wykorzystywany do przechowywania danych binarnych pliku (jego rekordów). Drugi, plik indeksowy, będzie przechowywał metadane plików tj. identyfikatory i rozmiary rekordów oraz informacje o ich położeniu w plikach z danymi. System plików wykorzystywany na dysku twardym nie będzie miał znaczenia, jeżeli tylko będzie zgodny ze standardem POSIX. Struktura pliku danych będzie bardzo prosta. Kolejne rekordy będą dopisywane na koniec pliku dyskowego. Struktura pliku indeksowego będzie bardziej złożona. Plik indeksowy będzie składał się z wpisów, z których każdy będzie odpowiadał za jeden rekord pliku. Struktura wpisu w pliku indeksowym jest przedstawiona poniżej w języku C.

```
struct index_entry_t
{
    uint64_t id;
    uint32_t size;
    uint64_t offset;
    uint8_t flag;
    uint32_t crc32;
};
```

Każdy wpis indeksowy zawiera pole *id*, które jest identyfikatorem rekordu w pliku systemowym. Pole *size* informuje o długości rekordu, a *offset* o początku danych rekordu w odpowiednim pliku z danymi. Pole *flag* będzie pozwalało na ustawienie flagi we wpisie indeksowym (np. informacji, że rekord został usunięty). Pole *crc32* będzie sumą kontrolną wpisu indeksowego. W przypadku uszkodzenia systemu plików, na którym przechowywane są pliki dyskowe (lub samego urządzenia) suma kontrolna pozwoli to wykryć i odpowiednio zareagować (w praktyce oznaczać to będzie automatyczne usunięcie rekordu z pliku).

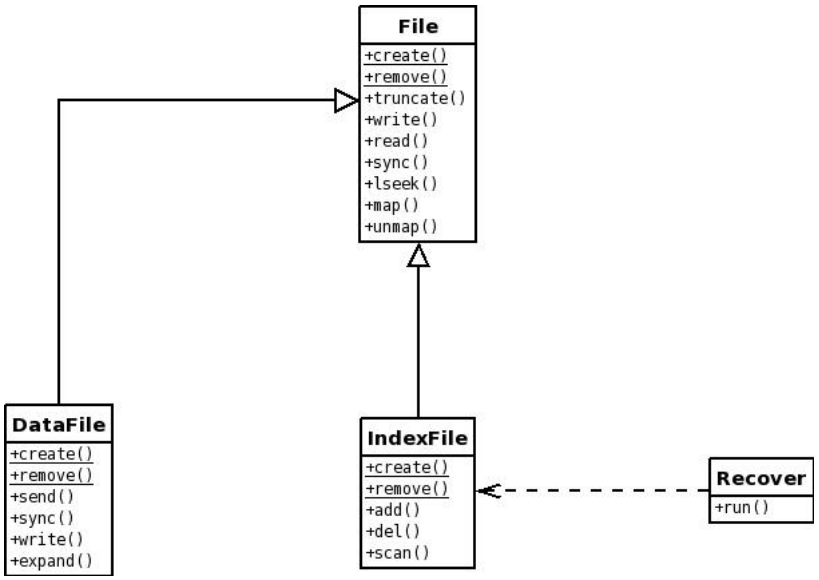
Odczyt danych z plików indeksowych będzie odbywał się tylko w czasie uruchamiania serwera *DataNode*. Następnie dane te będą pobieranie z obiektu klasy *DNFileAccess*, który będzie przechowywany w pamięci.

2.3.5. Klasy do operacji na plikach dyskowych

Moduł dostępu do danych dyskowych będzie składał się z kilku klas odpowiedzialnych za realizację operacji wejścia-wyjścia na plikach systemu operacyjnego. Klasy te będą stanowiły warstwę pośrednią pomiędzy programem serwera, a wywołaniami systemowymi. Klasy modułu zostały przedstawone na rysunku 2.9.

Klasa *DataFile*

Klasa *DataFile* będzie służyła do wykonywania operacji na plikach z danymi rekordów. Operacje *create()*, *remove()*, *write()* i *sync()* będą miały analogiczne działanie jak w klasie bazowej



Rysunek 2.9: Klasy operacji dyskowych

File. Funkcja *send()* będzie miała specjalne znaczenie dla operacji odczytu danych rekordu. Korzystając z niej będzie można w bardzo wydajny sposób przesyłać dane pliku do bufora gniazda sieciowego z pominięciem pamięci podręcznej stron dzięki mechanizmom zawartym w funkcji systemowej *sendfile()*. Funkcja *expand()* będzie pozwalała na zwiększenie rozmiaru pliku z danymi (bez alokacji bloków) w celu przydzielenia fragmentu pliku na dane rekordu. Dzięki tej operacji, która jest bardzo szybka i będzie wykonywana po założeniu blokady wyłącznej, będzie możliwy równoległy zapis danych do tego samego pliku dyskowego z wielu wątków serwera (a przez to również z wielu wątków klienckich systemu).

Klasa *IndexFile*

Klasa *IndexFile* będzie odpowiedzialna za wykonywanie operacji na pliku indeksowym. Operacje *create()* i *remove()* będą miały analogiczne działanie, jak w klasie bazowej *File*. Metoda *add()* będzie umożliwiać dopisywanie danych o nowych rekordach do pliku indeksowego, a *del()* ich usunięcie. Metoda *scan()* będzie wykorzystywana w obiektach klasy *Recover* do pobrania informacji o wszystkich rekordach w pliku przy starcie serwera danych.

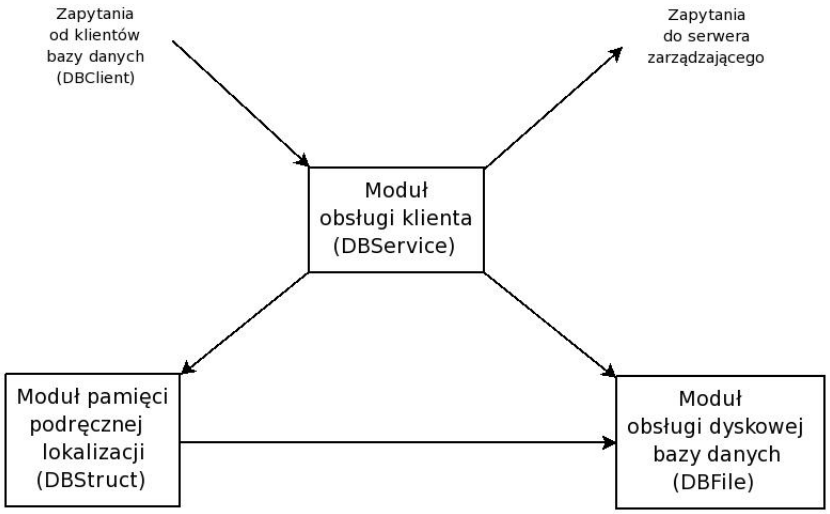
Klasa *Recover*

Obiekt tej klasy będzie wykorzystywany tylko na początku działania serwera danych. Po starcie programu zostanie on utworzony, aby wykonać prace związane z załadowaniem danych zawartych w plikach indeksowych (informacje o rekordach przechowywanych w plikach) do pamięci podręcznej obiektu *DNFileAccess*. Po wykonaniu tego zadania (zakończenie wykonania funkcji *run()*) obiekt jest niszczony. W celu odczytu danych z plików indeksowych obiekt będzie wykorzystywał obiekty klasy *IndexFile* do przeprowadzenia operacji wejścia-wyjścia na plikach dyskowych.

2.4. Baza danych lokalizacji plików *MasterDB*

W tym rozdziale zostanie opisany projekt implementacji bazy danych lokalizacji plików w systemie. Baza danych lokalizacji odpowiada za przechowywanie wszystkich informacji o lokalizacjach replik plików w systemie. Lokalizację pliku będzie stanowił zestaw adresów IP komputerów, na których pracują serwery danych przechowujące repliki pliku.

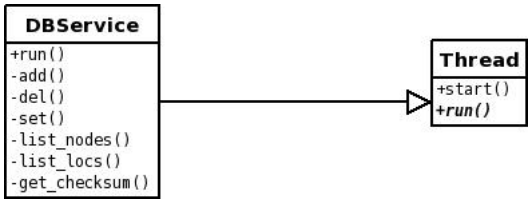
Rysunek 2.10 przedstawia architekturę serwera bazy danych lokalizacji plików.



Rysunek 2.10: Architektura bazy danych lokalizacji plików

2.4.1. Obsługa zapytań klientów

Za obsługę zapytań klientów będzie odpowiedzialna grupa wątków tworzonych przy starcie serwera. Każdy wątek będzie miał przyporządkowany obiekt klasy *DBService*. Metody tego obiektu będą realizowały różne zapytania klientów.



Rysunek 2.11: Klasa *DBSerice*

Metoda *run()* obiektu *DBService* będzie funkcją realizującą pętlę główną wątku obsługi klienta. Każde polecenie przesłane do bazy danych lokalizacji będzie analizowane, a następnie uruchomiona zostanie jedna z metod obsługi (w zależności od polecenia). Metoda *add()* służy do tworzenia nowych plików w systemie. W wyniku jej wywołania baza danych wygeneruje zapytanie do serwera zarządzającego o utworzenie nowego pliku w systemie (jeżeli taki plik jeszcze nie istnieje w pamięci podręcznej lokalizacji). Serwer zarządzający, na postawie statystyk systemu, wybierze komputery z serwerami danych, na których utworzy repliki pliku. Informacja o liście adresów IP tych komputerów zostanie przekazana do bazy danych

lokalizacji gdzie zostanie zapisana w pamięci podręcznej oraz w pliku dyskowym przechowującym informacje o lokalizacjach (obsługiwany w obiekcie *DBFile*). Metoda *del()* będzie służyć do usuwania lokalizacji pliku. Po wywołaniu tej metody plik przestaje istnieć w systemie – wszelkie próby odczytu danych pliku zakończą się informacją o braku pliku. Funkcja *set()* pozwoli na ustawienie lokalizacji pliku (metoda będzie mogła być wywołana tylko przez serwer zarządzający – klienci zewnętrzni nie będą mogli zmieniać lokalizacji plików w systemie).

Podobnie metoda *list_nodes()* (pobranie listy komputerów, na których są jakiekolwiek pliki) będzie wykorzystywana przez serwer zarządzający do badania stanu systemu.

Metody *list_locs()* oraz *get_checksum()* będą służyły do realizacji specjalnych zapytań związanych z obsługą kopii zapasowych bazy danych (opis w dalszej części rozdziału).

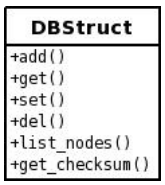
2.4.2. Dyskowa baza danych

W celu umożliwienia czasowego wyłączania komputera wszystkie informacje przechowywane w bazie danych lokalizacji są dodatkowo zapisywane do plikowej bazy danych (plik, na dysku twardym serwera bazy danych lokalizacji, o odpowiedniej strukturze).

Za dostęp do dyskowej bazy danych odpowiadać będzie obiekt klasy *DBFile* (dziedziczy po klasie *File*). Będzie on posiadał jedynie kilka prostych metod. Metoda *add()* będzie dodawać informacje o nowej lokalizacji do pliku bazodanowego, *remove()* będzie ją usuwać, a *set()* umożliwi jej modyfikację. Dodatkowo w obiekcie klasy będzie zaimplementowana metoda *scan()*, która umożliwi odczyt wszystkich danych z pliku jednocześnie (i załadowanie ich do pamięci podręcznej lokalizacji).

2.4.3. Pamięć podręczna lokalizacji

Pamięć podręczna lokalizacji plików będzie zorganizowana w obiekcie klasy *DBStruct*.

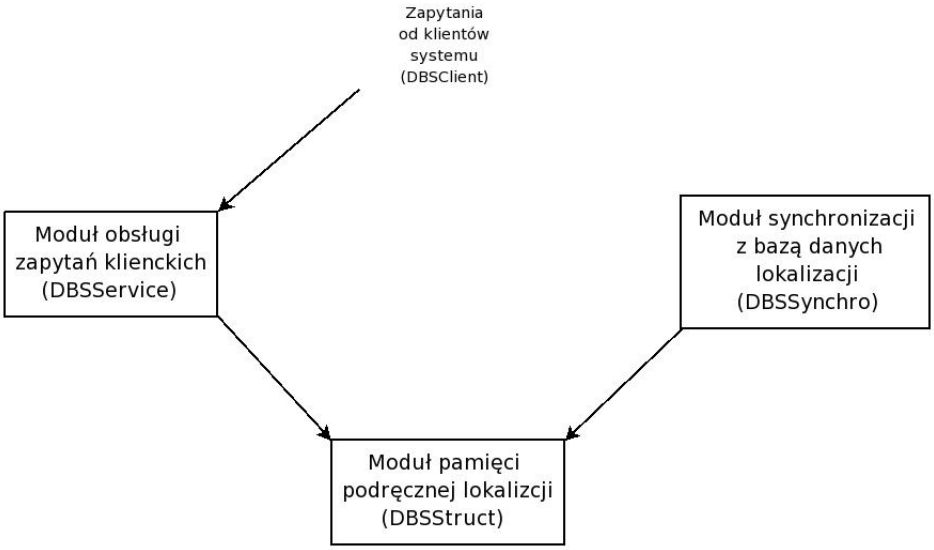


Rysunek 2.12: Klasa *DBStruct*

Metoda *add()* będzie dodawać do pamięci podręcznej informację o lokalizacji pliku. Do pobierania informacji służyć będzie metoda *get()*. Modyfikacji będziemy dokonywać poprzez wywołanie metody *set()*, a usuwanie poprzez *del()*. Metoda *list_nodes()* będzie generować listę adresów IP komputerów, na których znajdują się pliki w systemie (będzie to wykorzystywane w serwerze zarządzającym). Metoda *get_checksum()* będzie służyła do generowania sumy kontrolnej lokalizacji grupy plików (dzięki temu łatwiej będzie można synchronizować zawartość bazy danych lokalizacji z jej kopiami).

2.5. Kopia bazy danych lokalizacji *SlaveDB*

Kopia bazy danych lokalizacji będzie miała za zadanie zapewnić działanie systemu w sytuacji gdy główna baza danych lokalizacji (*MasterDB*) przestanie działać. Dzięki temu w systemie nie będzie żadnego pojedynczego punktu awarii.



Rysunek 2.13: Architektura kopii bazy danych lokalizacji

W przypadku awarii lub przeciążenia głównej bazy danych lokalizacji, kopia bazy danych będzie przejmowała jej zadania. W systemie będzie można zainstalować dowolną liczbę kopii bazy danych. Klienci będą mogą zadawać tylko jeden typ zapytań do kopii bazy danych – pobieranie lokalizacji pliku (metoda *get()* obiektu *DBSService*). Kopie bazy danych nie będą pozwalały na tworzenie nowych plików. Dodawanie nowych wpisów do pamięci podręcznej kopii bazy danych (obiekt klasy *DBSStruct*) będzie wykonywane w oddzielnym wątku (obiekt *DBSSynchro*). Wątek ten będzie odpowiedzialny za synchronizację pamięci podręcznej kopii bazy danych z główną bazą danych lokalizacji. Synchronizacja ta będzie przebiegać bardzo efektywnie dzięki wykorzystaniu zestawu specjalnie zaprojektowanych funkcji, które będą pobierały jedynie te wpisy z głównej bazy danych, które zostały zmodyfikowane lub dodane, ograniczając obciążenie głównej bazy danych lokalizacji.

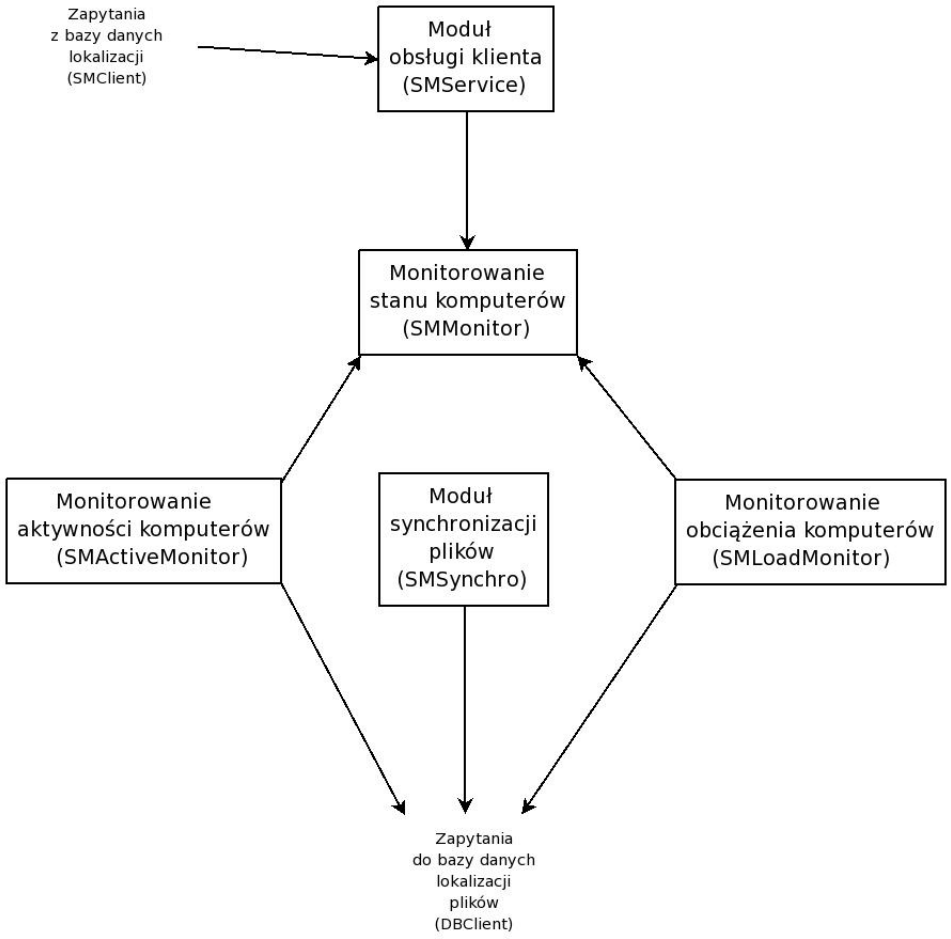
2.6. Serwer zarządzający *SysManager*

W tym rozdziale zostanie opisana architektura serwera zarządzającego systemem. Rysunek 2.14 przedstawia główne elementy składowe serwera.

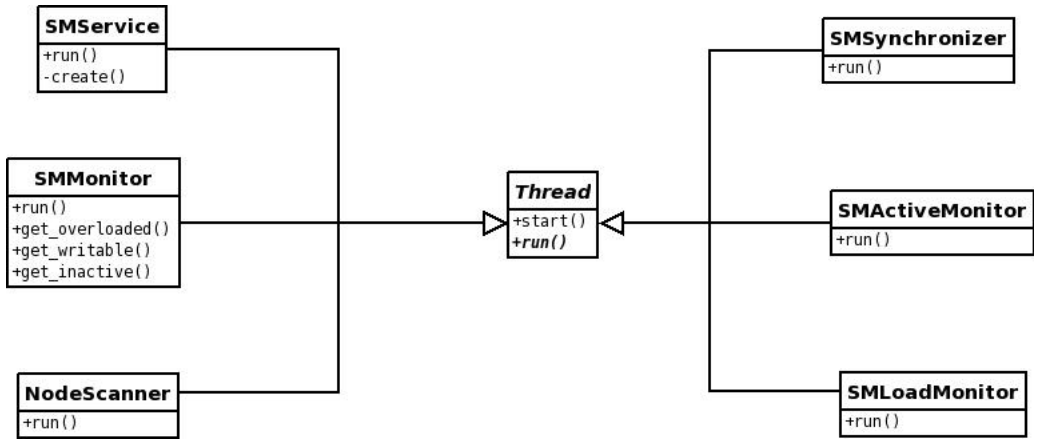
Rysunek 2.15 przedstawia klasy obiektów, które reprezentują wątki wykonania serwera zarządzającego.

2.6.1. Moduł obsługi klienta

Zapytania do serwera zarządzającego będą generowane w bibliotece klienckiej składającej się z klasy *SMClient*. Serwer odpowiada tylko na jeden typ zapytań – o utworzenie replik pliku. Zapytania te generowane będą w bazie danych lokalizacji przy tworzeniu pliku. Obiekty klasy *SMService* będą reprezentowały wątki, które realizują te zapytania. W metodzie *create()*



Rysunek 2.14: Architektura serwera zarządzającego



Rysunek 2.15: Wątki wykonania serwera zarządzającego

będzie wykonywany kod służący do wybrania nowych replik (adresów IP komputerów), na których zostaną założone repliki pliku, a następnie przeprowadzane będzie samo założenie replik tworzonego pliku poprzez wysłanie komunikatów do serwerów danych pracujących na wybranych komputerach. Po wygenerowaniu replik adresy komputerów zostaną przekazane

do biblioteki klienckiej jako wynik wykonania zapytania.

2.6.2. Monitorowanie stanu komputerów

Obiekt *SMMonitor* będzie służył do odbierania informacji od serwerów danych uruchomionych na komputerach w systemie. Zgłoszenia te będą zapisywane do kolejki zgłoszeń. Zgłoszenia z tej kolejki będą przetwarzane w jednym z wątków wewnętrznych (klasa *NodeScanner*, rysunek 2.15). Przetwarzane zapytań będzie polegało na wygenerowaniu zapytania do serwera danych, w którym pobierane będą informacje o zajętości dysku i obciążeniu systemu. Wyniki tych zapytań będą następnie zapisywane w obiekcie *SMMonitor*. Jeżeli komputer, na którym działa serwer danych, nie będzie przysyłał zgłoszeń (brak aktywności komputera) lub nie będzie odpowiadał na zapytania z wątku *NodeScanner*, to zostanie uznany za nieaktywny.

2.6.3. Monitorowanie aktywności komputerów

Za monitorowanie aktywności komputerów będzie odpowiadał obiekt klasy *SMActiveMonitor*. Wątek wewnętrzny obiektu będzie pobierał informacje o nieaktywnych komputerach z obiektu *SMMonitor* poprzez wywołanie metody *get_inactive()*. Następnie z bazy danych lokalizacji będzie pobierana lista plików, które mają replikę na nieaktywnych komputerach. Dla każdego z tych plików będzie generowana nowa replika (metoda *get_writable()* obiektu *SMMonitor*), a następnie rozsyłane będą polecenia wymuszające synchronizację nowo utworzonych replik. Na końcu nowe lokalizacje plików zostaną zapisane do bazy danych lokalizacji. Ta czynność będzie powtarzana do momentu, aż wszystkie repliki plików będą na aktywnych komputerach.

2.6.4. Monitorowanie obciążenia komputerów

Obiekt klasy *SMLoadMonitor* będzie odpowiadał za równoważenie obciążenia na komputerach z serwerami danych, które będą pracowały w ramach systemu. Wątek główny obiektu będzie wywoływał metodę *get_overloaded()* obiektu *SMMonitor*. Metoda ta przekaże listę komputerów, które odpowiedziały komunikatem o przeciążeniu na zapytanie generowane w wątku *NodeScanner*. Dla każdego z takich komputerów zostanie wygenerowane zapytanie do serwera danych o podanie listy intensywnie wykorzystywanych plików (serwery danych przechowują te informacje w statystykach wewnętrznych). Z pobranej listy zostanie wybrany losowo jeden plik, a następnie zostanie dla niego wygenerowana nowa replika (metoda *get_writable()* obiektu *SMMonitor*). Nowa lokalizacja zostanie zapisana do bazy danych. Od tej chwili klienci generujący zapytanie do pliku będą korzystać z nowej lokalizacji, w której nie będzie już przeciążonego komputera (a to spowoduje stopniowe jego odciążenie).

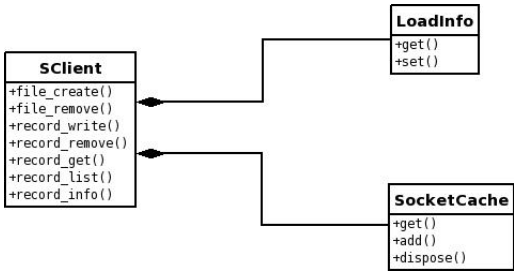
2.6.5. Moduł synchronizacji plików

W projektowanym systemie będzie dochodziło do sytuacji, w których, w wyniku awarii lub chwilowych przeciążeń część replik plików nie będzie przechowywała wszystkich rekordów plików. Aby system zapewniał odpowiedni poziom bezpieczeństwa i dostępności danych, należy takie sytuacje jak najszybciej wykrywać i poprawiać. Pracę tą będą wykonywać w serwerze zarządzającym wątki wewnętrzne *SMSynchronizer* modułu synchronizacji plików. W obiekcie głównym klasy *SMSynchro* przechowywana będzie kolejka zadań złożona z identyfikatorów plików przechowywanych w systemie. Kolejka ta będzie uzupełniana poprzez zapytania do bazy danych lokalizacji. Dla każdego pliku będą generowane zapytania do serwerów danych przechowujących jego repliki w celu pobrania skrótu pliku (64-bitowa liczba reprezentująca

zawartość pliku), dzięki któremu będziemy w stanie szybko wykryć niespójne repliki. Jeżeli niespójność zostanie wykryta (skróty plików różnią się pomiędzy replikami), to zostanie rozesłane polecenie wymuszające synchronizację pliku, które zostanie wykonane przez wątki synchronizujące odpowiednich serwerów danych.

2.7. Biblioteka kliencka

Rysunek 2.16 przedstawia główne elementy biblioteki klienckiej.



Rysunek 2.16: Biblioteka kliencka systemu

Podstawowym elementem biblioteki będzie obiekt kliencki klasy *SClient*. Poprzez wywołania odpowiednich metod obiektu będzie można wykonywać wszystkie operacje na plikach i rekordach w systemie. Metody *file_create()* oraz *file_remove()* będą odpowiadać za tworzenie i usuwanie plików w systemie. Wywołanie *file_write()* pozwala na zapisanie nowego rekordu do pliku. W tej funkcji rekord będzie zapisywany synchronicznie do jednej z replik pliku, a następnie asynchronicznie zostanie skopiowany do pozostałych. Metoda *record_remove()* będzie pozwalała na skasowanie rekordu z pliku (rekord będzie usuwany ze wszystkich replik). Do odczytu danych rekordu będzie służyła metoda *record_read()*. Dzięki *record_list()* będzie można uzyskać informacje o rekordach, które zostały już zapisane do pliku, a metoda *record_info()* pozwoli na pobranie informacji o wybranym rekordzie (rozmiar i czy jest skasowany).

2.7.1. Równoważenie obciążenia w bibliotece klienckiej

Każdy klient systemu, poprzez bibliotekę kliencką, będzie komunikował się, korzystając z wewnętrznego obiektu klasy *SocketCache*, z replikami pliku, którego dane czyta lub zapisuje. Aby ta komunikacja była jak najwydajniejsza, zastosowany zostanie mechanizm pozwalający klientowi na komunikację z replikami pliku, które w tym momencie są najmniej obciążone. Do tego celu w każdym obiekcie klienckim zostanie wprowadzony specjalny obiekt klasy *LoadInfo*, który będzie przechowywał informacje o aktualnym obciążeniu komputera. Przed zadaniem zapytania każda metoda obiektu klienckiego klasy *SClient* będzie badać obciążenie replik (poprzez wywołanie metody *get()* obiektu wewnętrznego *LoadInfo*). Następnie z replik będzie wybierana ta, która ma najmniejsze obciążenie i tam w pierwszej kolejności będzie wysyłane zapytanie. Jeżeli w obiekcie *LoadInfo* nie będzie informacji o obciążeniu komputera lub informacja będzie zbyt stara, to zostanie wywołana funkcja *load_test()* z obiektu klienta *DNClient*. Wynik wywołania tej funkcji zostanie wpisany do obiektu *LoadInfo* poprzez wywołanie metody *set()* tego obiektu. Takie rozwiązanie spowoduje lepsze rozłożenie obciążeń w systemie, przez co poprawi jego wydajność i skalowalność. Więcej o algorytmach równoważenia obciążenia w systemach rozproszonych można znaleźć w [16].

2.7.2. Obsługa kopii bazy danych lokalizacji

Każde zapytanie klienta o dane pliku w systemie wymaga komunikacji z główną bazą danych. Takie rozwiązanie może powodować problemy z korzystaniem z systemu gdy komputer, na którym pracuje baza danych lokalizacji, ulegnie awarii. Aby zabezpieczyć się przed tym problemem wprowadzony zostanie mechanizm kopii zapasowych bazy danych. Biblioteka kliencka będzie potrafiła korzystać z kopii bazy danych (dowolnej liczby kopii) i gdy nie powiedzie się zapytanie do głównej bazy danych (z powodu awarii komputera, ale również przy przeciążeniu) wszystkie zapytania zostaną przekierowane do jednej z kopii. Po ustaniu awarii (lub przeciążenia) wszystkie zapytania wrócą do głównej bazy lokalizacji. Nie wszystkie zapytania do systemu będą mogły być generowane w czasie awarii. Kopie bazy danych nie pozwalają na tworzenie nowych plików w systemie (aby nie powodować rozspójnienia metadanych systemu w sytuacji podziału sieci np. przy awarii łącza między serwerowniami). Z założenia jednak tworzenie plików będzie rzadkie w stosunku do zapytań czytających i zapisujących dane.

2.8. Mechanizmy systemowe

Tworzony system będzie wykorzystywał kilka specyficznych funkcji systemu operacyjnego. Funkcje, o których mowa, służą głównie do przeprowadzania bardzo wydajnych operacji wejścia-wyjścia i są stosowane głównie przy implementacji bardzo wydajnego oprogramowania serwerowego.

2.8.1. Operacje na plikach i gniazdach sieciowych

Ponieważ elementy tworzonego systemu intensywnie wykorzystują pliki dyskowe systemu operacyjnego, konieczne jest zapewnienie, aby operacje wykonywane na plikach dyskowych były jak najwydajniejsze. W tym celu będzie wykorzystywany mechanizm odwzorowywania plików dyskowych w pamięci operacyjnej poprzez wywołania funkcji systemowych *mmap()*, *munmap()* i *msync()*, a także funkcje systemowe umożliwiające przeprowadzenie operacji wejścia-wyjścia bez angażowania procesora głównego do kopiowania danych (ang. *zero-copy I/O*). Funkcje *sendfile()* oraz *splice()*, należące do tej grupy, umożliwiają efektywne operacje kopiowania danych pomiędzy plikami dyskowymi i gniazdami połączeń sieciowych. Więcej o tych funkcjach można wyczytać w [9] oraz w wielu artykułach zamieszczonych na stronach WWW (np. [17]).

Rozdział 3

Testy systemu

W tym rozdziale opisane zostaną scenariusze i wyniki testów przeprowadzonych na tworzonym systemie. Zacznę od przedstawienia testów wydajności systemu. Następnie przedstawię testy funkcji systemu związanych z działaniem i regeneracją systemu po awariach. W przeprowadzanych testach będę wykorzystywał rekordy wielkości od 1 KB do 100 MB. Pokrywają one przedział wielkości porcji danych wykorzystywanych w zastosowaniach, dla których tworzony jest system. Do części testów wybieram rekordy o rozmiarach 10 KB oraz 1 MB. Wielkości te wynikają ze średnich wielkości plików przechowywanych w systemach plików dla serwisów pocztowych oraz z plikami multimedialnymi. W testach wykorzystuję pliki złożone z trzech replik, ponieważ stanowi to najczęściej stosowany kompromis pomiędzy kosztem utrzymywania danych a ich bezpieczeństwem.

3.1. Środowisko testowe

Środowisko testowe składać się będzie z dziesięciu komputerów klasy PC. Każdy z komputerów wyposażony jest w procesor Intel Celeron 2.5 GHz, 2 GB pamięci RAM oraz dysk twardy 200 GB, o prędkości obrotowej 7200. Komputery połączone są siecią o prędkości 1 Gb/s. Jeden z komputerów został wyróżniony jako komputer, na którym będzie uruchomiony serwer zarządzający oraz baza danych lokalizacji. Kolejny został przeznaczony do generowania zapytań klienckich dla testów. Pozostałe komputery są przeznaczone do pracy z serwerami danych. Dane w serwerach danych są przechowywane na partycjach dyskowych z zainstalowanym systemem plików EXT3, który pracuje w trybie *ordered* (tylko kronikowanie metadanych).

3.2. Testy wydajności

W tym punkcie przedstawione zostaną senariusze i wyniki testów wydajności systemu. Testy będą polegały na badaniu prędkości wykonania operacji zapisu i odczytu danych w różnych warunkach. Testy nie będą obejmowały operacji usuwania rekordów ani tworzenia i usuwania plików. Operacje te są bardzo rzadko wykonywane w zastosowaniach, dla których tworzony jest system, i ich prędkość nie będzie miała znaczenia dla oceny spełnialności wymagań w tych zastosowaniach.

3.2.1. Zapis rekordów różnej wielkości

Te testy będą miały za zadanie zbadać zachowanie systemu przy operacji zapisu danych (rekordów) do plików. W teście będziemy badać średnią prędkość uzyskaną przy operacji

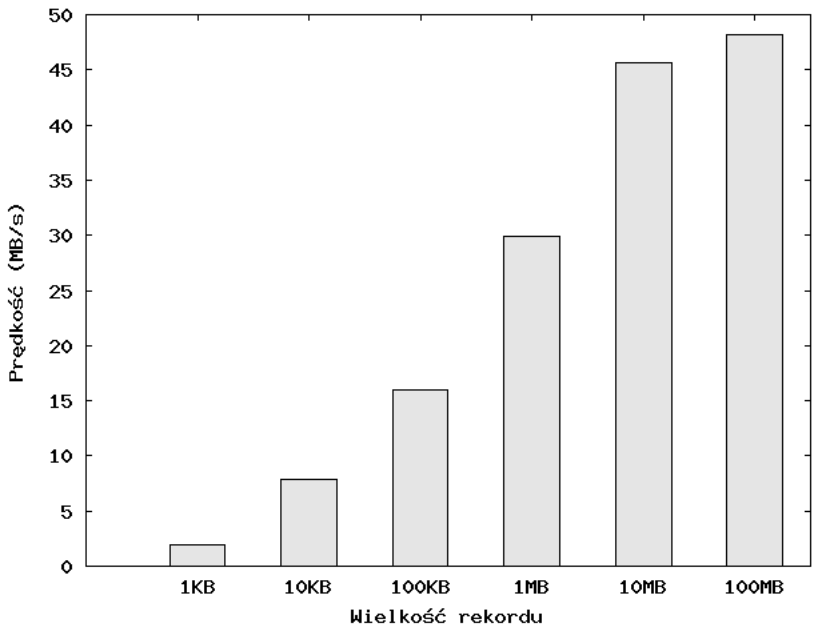
zapisu kolejnych rekordów tej samej wielkości w przedziale czasowym. Test przeprowadzany jest dla kilku rozmiarów rekordów. Rozmiary rekordów zostały tak dobrane, aby pokryć cały zakres możliwości w różnych środowiskach działania systemu.

Warunki przeprowadzenia testów

Przed rozpoczęciem każdego testu tworzony jest w systemie plik złożony z trzech replik. Z komputera klienckiego zapisywane są rekordy o ustalonej wielkości do utworzonego pliku. Zapis odbywa się w ten sposób, że po zakończeniu zapisu jednego rekordu natychmiast jest zapisywany następny (w pętli nieskończonej). Każdy test trwa 30 minut. W czasie testu monitorujemy prędkość przesyłania danych z komputera klienckiego monitorując statystyki interfejsu karty sieciowej. Statystyki przesyłania danych są pobierane co 30 sekund. Wyniki będą prezentowały średnią prędkość zapisu danych do systemu uzyskaną w czasie całego testu.

Test jest powtarzany dla rekordów o wielkościach 1 KB, 10 KB, 100 KB, 1 MB, 10 MB oraz 100 MB.

Wyniki



Rysunek 3.1: Test zapisu rekordów różnej wielkości

Rysunek 3.1 przedstawia wyniki przeprowadzonych testów. Zapisy wielu małych rekordów do systemu nie pozwalają na wykorzystanie pełnej przepustowości dysku twardego. Wynika to z dużych narzutów związanych z częstym wykonywaniem funkcji systemowej *fsync()* oraz częstą modyfikacją metadanych systemu plików, na którym przechowywane są dane serwerów danych. Przy przesyłaniu wielu małych rekordów musimy często wysyłać nagłówki komunikacyjne oraz pobierać lokalizację plików z bazy danych lokalizacji. Ta komunikacja dodatkowo spowalnia działanie systemu dla małych rekordów.

Odchylenie standardowe dla każdego z testów wyniosło w przybliżeniu 10% wartości średniej.

Prędkość działania systemu wzrasta wraz ze wzrostem wielkości zapisywanych rekordów. Przy rekordach o wielkości 10 MB i 100 MB prędkość zapisu danych osiąga granicę przepustowości systemu przy zapisie z jednego klienta do jednego pliku. Poziom prędkości granicznej wynika z algorytmu działania operacji zapisu (dane są buforowane w pamięci, a dopiero później zapisywane na dysk twardy).

Wnioski

Z przeprowadzonych testów wynika, że system dobrze radzi sobie z zapisem rekordów o większych rozmiarach. Zapis wielu małych porcji danych zawsze jest kłopotem dla systemu plików, gdyż powoduje duży narzut na modyfikację metadanych przy wykonaniu operacji. Poza tym zapis jest przeprowadzony z jednego procesu klienckiego. W rzeczywistych warunkach w systemie będzie działało wiele równoległych procesów zapisujących co znacznie przyspieszy prędkość operacji przy małych rozmiarach rekordów (badane w następnych testach).

3.2.2. Odczyt rekordów różnej wielkości

Testy będą miały zbadać zachowanie systemu przy operacji odczytu danych z plików. W jednym teście będziemy badać średnią prędkość odczytu danych przy operacji odczytu rekordów określonego rozmiaru z jednego pliku w przedziale czasowym. Test przeprowadzony będzie dla kilku rozmiarów rekordów.

Warunki przeprowadzenia testów

Przed rozpoczęciem każdego testu tworzony jest w systemie plik złożony z trzech replik. Do pliku zapisujemy rekordy do momentu, aż sumaryczny rozmiar pliku osiągnie 100 GB. Czyli przy teście odczytu rekordów o rozmiarze 1 KB zapiszemy 104857600 rekordy, a przy rekordach wielkości 100 MB zapiszemy ich 1024.

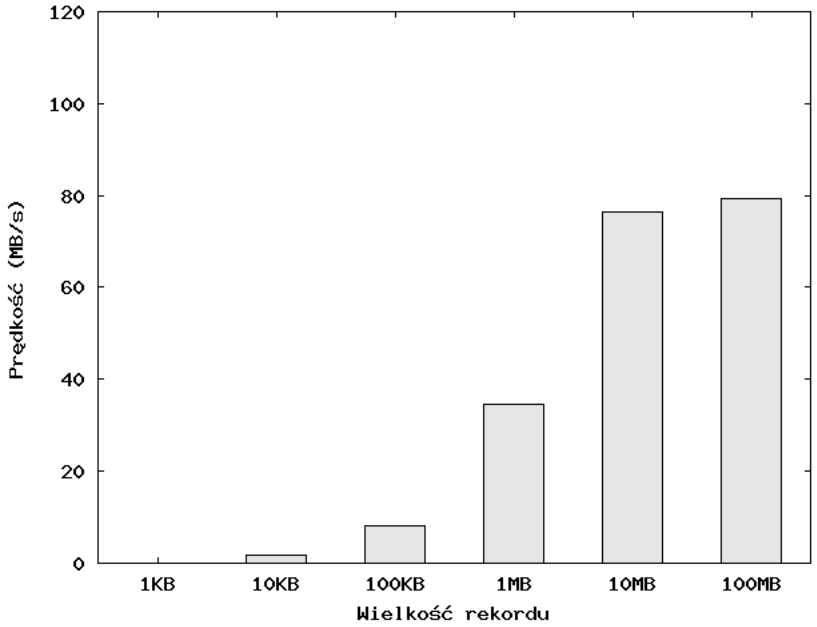
Każdy test będzie polegał na odczycie losowo wybranych rekordów z pliku w pętli przez 30 minut. Dzięki temu, że rekordy są wybierane losowo, system operacyjny serwerów danych nie będzie wykorzystywał pamięci podręcznej stron do optymalizacji dostępu do danych dyskowych. Praktycznie za każdym razem, gdy zostanie wysłane zapytanie o dane rekordu, system operacyjny serwera danych będzie musiał odczytać je z dysku twardego.

Test jest powtarzany dla rekordów o wielkościach 1 KB, 10 KB, 100 KB, 1 MB, 10 MB oraz 100 MB.

Wyniki

Jak widać z rysunku 3.2 odczyt niewielkich porcji danych, z losowych miejsc na dysku twardym bez wykorzystania pamięci podręcznej stron, jest dość wolny. Mamy tutaj do czynienia z kilkoma czynnikami wpływającymi na taki wynik. Najważniejszy wynika z konieczności zmiany położenia głowicy dysków twardych na komputerach z serwerami danych. Poza tym przy niewielkich porcjach danych mamy duży narzut związany z przesyłaniem wielu małych komunikatów w sieci. Przy rekordach większej wielkości problem ten jest znacznie mniejszy. Odczyt większych rekordów powoduje mniej przesunięć głowicy dysku w stosunku do odczytanych danych. Odczyty są bardziej sekwencyjne (dotyczą bloków położonych blisko siebie na dysku).

Odchylenie standardowe dla testów z małymi rekordami wyniosło w przybliżeniu 2% wartości średniej. Dla większych rekordów było równe ok. 10%.



Rysunek 3.2: Test odczytu rekordów różnej wielkości

Wnioski

System dobrze radzi sobie z odczytem danych w większych porcjach. Odczyt mniejszych danych nie jest szybki, ale trzeba pamiętać, że testy są przeprowadzone w specyficznych warunkach. Ze względu na rozmiar danych i losowy charakter dostępu systemy operacyjne serwerów danych nie wykorzystują pamięci podręcznej stron. Ma to szczególnie istotny wpływ na rezultaty odczytu niewielkich porcji danych.

W rzeczywistych aplikacjach, systemy intensywnie wykorzystują niewielki podzbiór przechowywanych danych. Przykładowo w systemach współpracujących z serwisami multimedialnymi najczęściej odczytywane są pliki znajdujące się na stronie głównej serwisu. Dane te są przechowywane w pamięci podręcznej stron systemu operacyjnego więc nie ma konieczności wykonywania odczytu z dysków twardych. Wpływa to w znacznym stopniu na prędkość operacji odczytu co zostanie pokazane w następnym teście.

3.2.3. Odczyt rekordów różnej wielkości (z wykorzystaniem pamięci podręcznej stron)

Kolejne testy będą badać zachowanie systemu przy operacji odczytu danych z plików z wykorzystaniem pamięci podręcznej stron systemu operacyjnego. W jednym teście będziemy badać średnią prędkość odczytu danych przy operacji odczytu rekordów określonego rozmiaru z jednego pliku w przedziale czasowym. Test przeprowadzony będzie dla kilku rozmiarów rekordów.

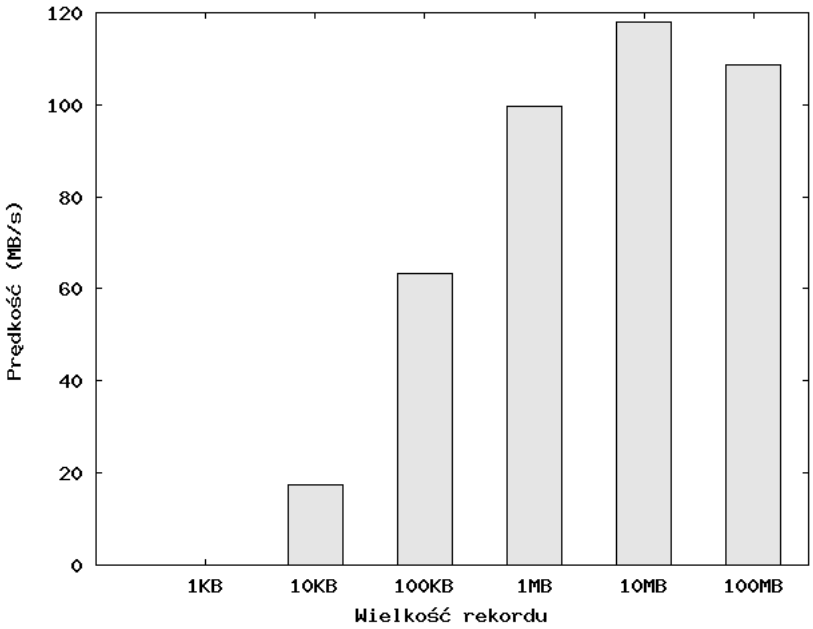
Warunki przeprowadzenia testów

Analogicznie jak poprzednio przed rozpoczęciem każdego testu tworzony jest w systemie plik złożony z trzech replik. Do pliku zapisujemy rekordy do momentu, aż sumaryczny rozmiar pliku osiągnie 100 GB.

Tym razem test będzie polegał na odczycie danych pliku, ale ograniczonej do pewnej części. Z każdego pliku będziemy odczytywać pierwsze rekordy o sumarycznej wielkości 1 GB z pliku (pierwszy gigabajt danych pliku). Dzięki temu pozwolimy aby system operacyjny serwera z danymi wykorzystał pamięć podręczną stron do optymalizacji dostępu do danych dyskowych.

Test jest prowadzony przez 30 minut, a pomiar prędkości odczytu danych jest dokonywany co 30 sekund. Wykonujemy testy dla rekordów o rozmiarach 1 KB, 10 KB, 100 KB, 1 MB, 10 MB oraz 100 MB.

Wyniki



Rysunek 3.3: Odczyt rekordów różnej wielkości z wykorzystaniem pamięci podręcznej stron

Rysunek 3.3 przedstawia wyniki testów. Dla małych rekordów (1 KB) prędkość odczytu jest niewielka. Dla rekordów 10 KB i większych znacznie wzrasta (w porównaniu do poprzedniego testu).

Odchylenie standardowe wahało się od 2% wartości średniej dla testów z małymi rekordami do 10% dla większych rekordów.

Wnioski

Testy pokazują jak duże znaczenie dla prędkości operacji odczytu ma wykorzystanie pamięci podręcznej stron.

Odczyty małych rekordów (1 KB) są nadal dość wolne ze względu na narzut komunikacyjny oraz narzut związany z obsługą wielu małych zapytań w systemie operacyjnym (dużo krótkich wywołań funkcji systemowych). Jednak już dla rekordów wielkości 10 KB uzyskujemy znaczny wzrost prędkości działania operacji. Przy rekordach 10 MB uzyskujemy maksymalną prędkość działania interfejsu sieciowego. Wykorzystanie pamięci podręcznej stron znacznie poprawiło prędkość odczytu danych.

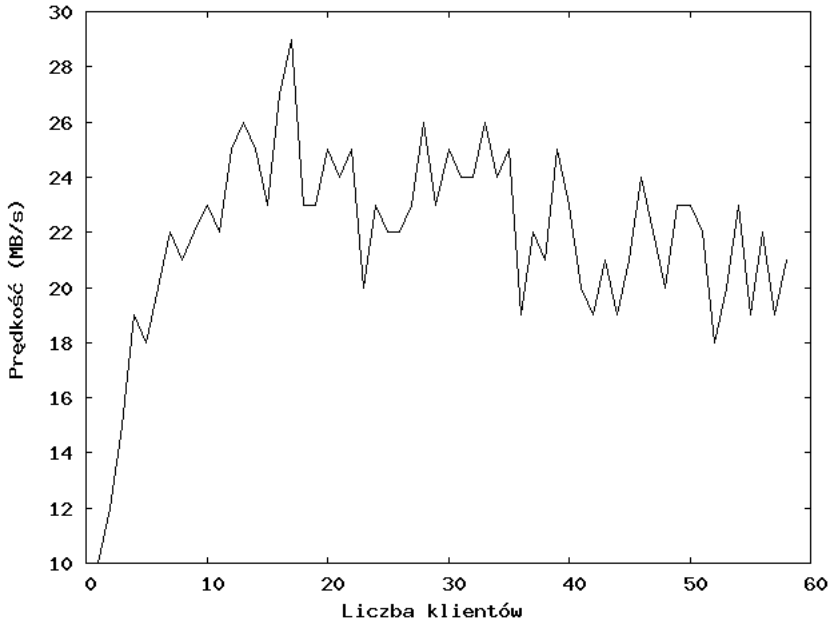
3.2.4. Zapis rekordów z wielu klientów (rekordy 10 KB)

W tym teście zbadane zostanie zachowanie systemu przy operacjach zapisu rekordów o rozmiarach 10 KB z różnej liczby procesów klienckich. Rozmiar rekordu został ustalony ze względu na to, że odpowiada on średniej wielkość wiadomości w systemach przechowujących pocztę elektroniczną. W teście będziemy badać średnią sumaryczną prędkość zapisu danych do systemu (poprzez monitorowanie statystyk interfejsu sieciowego komputera klienckiego) przy rosnącej liczbie klientów.

Warunki przeprowadzenia testów

Każdy proces kliencki tworzy w systemie nowy plik, a następnie zapisuje do niego rekordy 10 KB w pętli. Zaczynamy test z zerową liczbą klientów, a następnie co 10 minut dodajemy jeden proces. W ciągu tych 10-ciu minut mierzymy średnią prędkość zapisu danych (co 30 sekund). Każdy punkt wykresu będzie odpowiadał średniej prędkości zapisu danych przy ustalonej liczbie klientów w ciągu tych 10-ciu minut.

Wyniki



Rysunek 3.4: Zapisy rekordów 1 KB z wielu klientów

Rysunek 3.4 przedstawia wyniki testu. Do 10 klientów systemu mamy do czynienia ze wrostem prędkości działania. Przy dużej liczbie klientów następuje powolny spadek prędkości działania operacji zapisu.

Wnioski

Początkowo obserwujemy wzrost prędkości działania systemu. Wynika to z lepszego zrównoleglenia operacji, dzięki któremu możemy lepiej wykorzystać mechanizmy systemu operacyjnego i algorytmy dostępu do danych dyskowych. Przy większej liczbie równoległych klientów mamy do czynienia z powolnym spadkiem wydajności systemu. Wynika to z faktu, że wiele

procesów klienckich powoduje konieczność częstych zmian pozycji głowic dysków twardych. W ten sposób duża liczba równoległych klientów powoduje, że „przeszkadzają" sobie nawzajem. Dodatkowo w środowisku testowym mamy do dyspozycji jedynie 8 komputerów z serwerami danych.

W środowisku produkcyjnym, gdzie mamy do czynienia z większą liczbą komputerów taki efekt pojawiłby się przy znacznie większej liczbie procesów klienckich.

3.2.5. Zapis rekordów z wielu klientów (rekordy 1 MB)

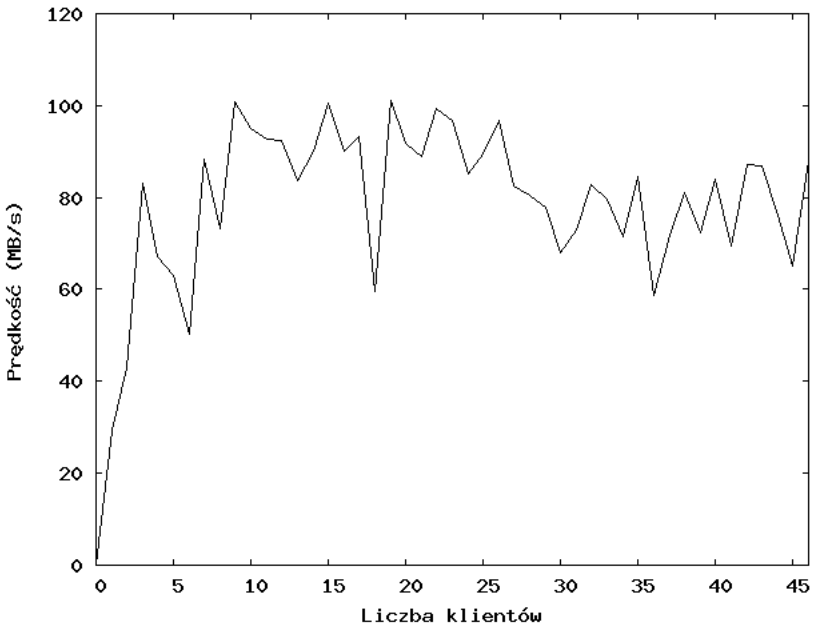
Test jest analogiczny do poprzedniego. Jedyną różnicą będzie wielkość rekordów. Tym razem zapisywane będą rekordy o wielkości 1 MB. Taka wielkość rekordu odpowiada średniej wielkości plików multimedialnych przechowywanym w systemach współpracujących z serwisami multimedialnymi.

W teście będziemy badać średnią prędkość zapisu danych do systemu (poprzez monitorowanie statystyk interfejsu sieciowego komputera klienckiego) przy rosnącej liczbie klientów.

Warunki przeprowadzenia testów

Każdy proces kliencki tworzy w systemie nowy plik, a następnie zapisuje do niego w pętli rekordy 1 MB. Zaczynamy test z zerową liczbą klientów, a następnie co 10 minut dodajemy jeden proces. W ciągu tych 10-ciu minut mierzymy średnią prędkość zapisu danych (co 30 sekund). Każdy punkt wykresu będzie odpowiadał średniej prędkości zapisu danych przy ustalonej liczbie klientów w ciągu tych 10-ciu minut.

Wyniki



Rysunek 3.5: Zapisy rekordów 1 MB z wielu klientów

Rysunek 3.5 przedstawia wyniki testu. Wyniki są podobne do poprzedniego testu. Znowu początkowo mamy do czynienia ze wzrostem wydajności operacji zapisu (przy małej liczbie klientów), a następnie wydajność powoli spada.

Wnioski

W teście mamy do czynienia z podobnymi do poprzedniego czynnikami. Przy małej liczbie klientów prędkość sumaryczna wzrasta ze względu na zrównoleglenie operacji. Później, przy dużej liczbie klientów prędkość spada. Ze względu na większy rozmiar rekordów (1 MB) wykres ma nieco inny kształt (szybszy wzrost prędkości i wolniejszy spadek przy dużej liczbie klientów). Wynika to z faktu, że większe rekordy pozwalają na więcej operacji sekwencyjnych na dysku twardym.

3.2.6. Odczyt rekordów z wielu klientów (rekordy 10 KB)

W tym teście badamy prędkość działania operacji odczytu danych z pliku przy różnej liczbie klientów. Rozmiar rekordu w teście to 10 KB.

W teście będziemy badać średnią prędkość odczytu danych z systemu (poprzez monitorowanie statystyk interfejsu sieciowego komputera klienckiego) przy rosnącej liczbie klientów.

Warunki przeprowadzenia testów

Przed rozpoczęciem testu tworzymy w systemie 100 plików. Każdy z plików składa się z 3 replik. Następnie zapisujemy do każdego z plików 1 GB danych w rekordach o rozmiarze 10 KB.

Każdy proces kliencki będzie losowo wybierał jeden plik, a następnie odczytywał z niego losowo wybrany rekord. Dzięki temu nie będzie wykorzystywana pamięć podręczna stron systemu operacyjnego.

Zaczynamy test z zerową liczbą klientów, a następnie co 10 minut dodajemy jeden proces. W ciągu tych 10-ciu minut mierzymy średnią prędkość odczytu danych (co 30 sekund). Każdy punkt wykresu będzie odpowiadał średniej prędkości odczytu danych przy ustalonej liczbie klientów w ciągu tych 10-ciu minut.

Wyniki

Rysunek 3.6 przedstawia wyniki testu. Początkowo prędkość odczytu rośnie. W późniejszej fazie testu powoli spada.

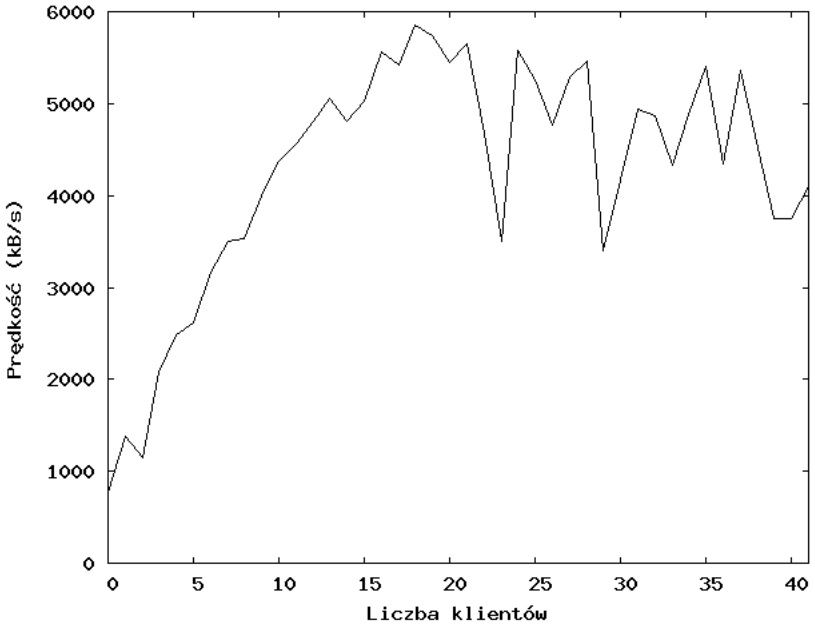
Wnioski

Jak widać zrównoleglenie operacji odczytu niewielkich rekordów bez wykorzystania pamięci podręcznej stron niewiele daje. Prędkość odczytu niewiele różni się niezależnie od liczby klientów. Sytuacja jednak zmieni się znacznie jeżeli pozwolimy, aby system wykorzystywał pamięć podręczną stron.

3.2.7. Odczyt rekordów z wielu klientów (rekordy 10 KB, z wykorzystaniem pamięci podręcznej stron)

W tym teście badamy prędkość działania operacji odczytu danych z pliku przy różnej liczbie klientów. Rozmiar rekordu, jak poprzednio, to 10 KB. Tym razem jednak pozwolimy, aby system wykorzystał pamięć podręczną stron.

W teście będziemy badać średnią prędkość odczytu danych z systemu (poprzez monitorowanie statystyk interfejsu sieciowego komputera klienckiego) przy rosnącej liczbie klientów.



Rysunek 3.6: Odczyt rekordów 10 KB z wielu klientów

Warunki przeprowadzenia testów

Przed rozpoczęciem testu tworzymy w systemie 100 plików. Każdy z plików składa się z 3 replik. Następnie zapisujemy do każdego z plików 1 GB danych w rekordach o rozmiarze 10 KB.

Każdy proces kliencki będzie losowo wybierał jeden plik. Następnie z pliku będą odczytywane tylko wybrane rekordy, znajdujące się w pierwszych 100 MB danych pliku.

Zaczynamy test z zerową liczbą klientów, a następnie co 10 minut dodajemy jeden proces. W ciągu tych 10-ciu minut mierzymy średnią prędkość odczytu danych (co 30 sekund). Każdy punkt wykresu będzie odpowiadał średniej prędkości odczytu danych przy ustalonej liczbie klientów w ciągu tych 10-ciu minut.

Wyniki

Rysunek 3.8 przedstawia wyniki testu.

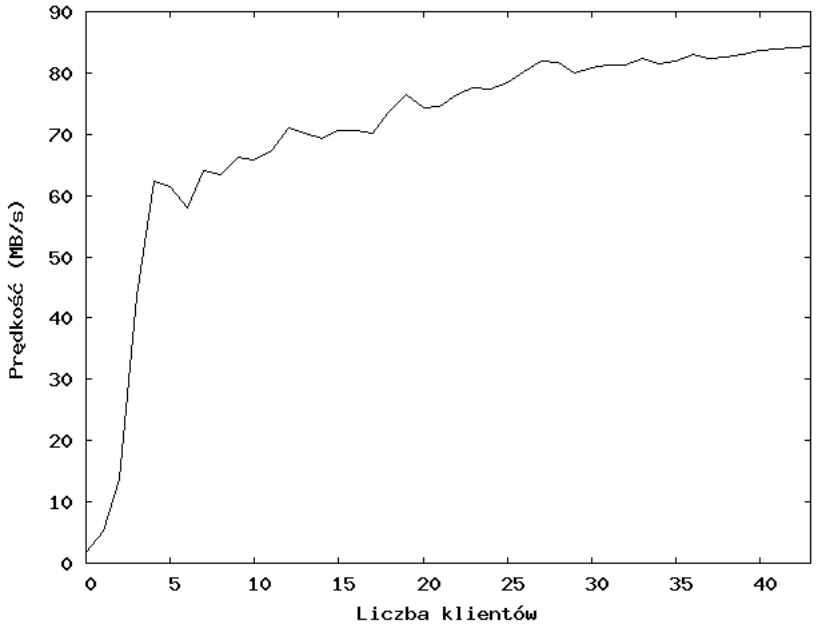
Wnioski

Znowu widzimy znaczenie pamięci podręcznej stron dla działania systemu. Prędkość odczytu danych bardzo szybko osiąga wysokie wartości i się na nich utrzymuje nawet przy dużej liczbie klientów.

W praktycznych zastosowaniach systemu będziemy mieli do czynienia z sytuacją zbliżoną do tej. Większość danych intensywnie wykorzystywana przez system będzie przechowywana w pamięci podręcznej stron i dzięki temu dostęp do nich będzie bardzo szybki.

3.2.8. Odczyt rekordów z wielu klientów (rekordy 1 MB)

Kolejne dwa testy są analogiczne do dwóch ostatnich. Jedyną różnicą jest wielkość rekordu wykorzystywanego do odczytu. Tym razem zastosujemy rekordy o rozmiarze 1 MB.



Rysunek 3.7: Odczyt rekordów 10 KB z wykorzystaniem pamięci podręcznej stron

W testach będziemy badać średnią prędkość odczytu danych z systemu (poprzez monitorowanie statystyk interfejsu sieciowego komputera klienckiego) przy rosnącej liczbie klientów.

Warunki przeprowadzenia testów

Przed rozpoczęciem testu tworzymy w systemie 100 plików. Każdy z plików składa się z 3 replik. Następnie zapisujemy do każdego z plików 1 GB danych w rekordach o rozmiarze 1 MB.

Pierwszy test będzie polegał na losowych zapytaniach do pliku o losowy rekord. Drugi test umożliwi wykorzystanie pamięci podręcznej stron przez generowanie zapytań tylko do wybranych rekordów (pierwsze 100 MB danych pliku).

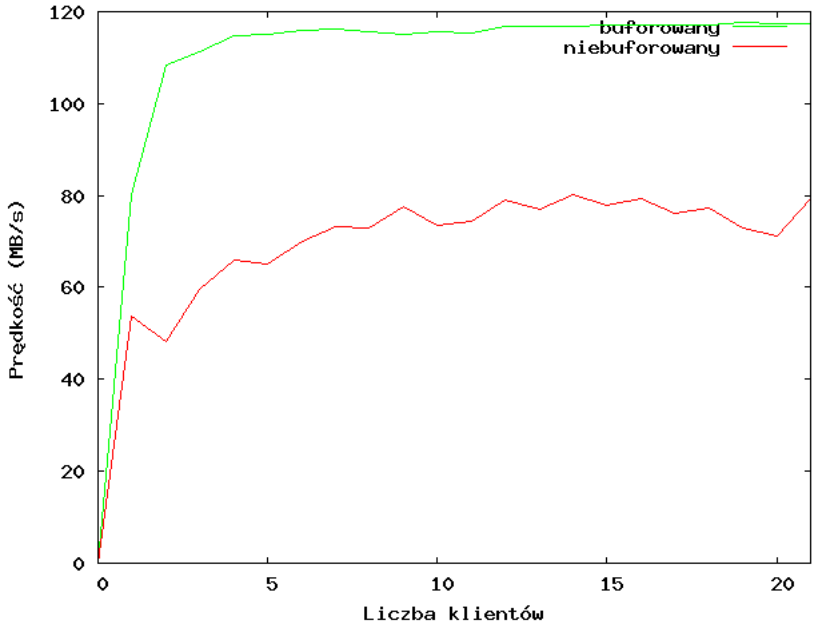
Zaczynamy test z zerową liczbą klientów, a następnie co 10 minut dodajemy jeden proces. W ciągu tych 10-ciu minut mierzymy średnią prędkość odczytu danych (co 30 sekund). Każdy punkt wykresu będzie odpowiadał średniej prędkości odczytu danych przy ustalonej liczbie klientów w ciągu tych 10-ciu minut.

Wyniki

Rysunek 3.8 przedstawia wyniki obydwu testów. Wykres „niebuforowany” oznacza test bez wykorzystania pamięci podręcznej stron, a „buforowany” – test przeprowadzony z wykorzystaniem tego mechanizmu.

Wnioski

Przy większych rekordach różnica w działaniu przy wykorzystaniu pamięci podręcznej stron nie jest tak ogromna jak przy mniejszych, jednak nawet teraz widać wyraźnie jak duże ma to znaczenie dla wydajności systemu.



Rysunek 3.8: Odczyt rekordów 1 MB z wielu klientów

3.3. Testy funkcji

W tym punkcie przedstawione zostaną testy funkcji systemu związane z wydajnością operacji i bezpieczeństwem danych w systemie w czasie awarii. Do testów będziemy wykorzystywać rekordy o rozmiarze 1 MB (taki rozmiar rekordów pozwala dobrze zobrazować niektóre cechy systemu).

3.3.1. Zapis rekordów do systemu przy częstych awariach komputerów

Testujemy zachowanie systemu w czasie operacji zapisu przy różnego rodzaju awariach. Awarie będą wprowadzane często (co 30 sekund), aby uniemożliwić procesowi serwera zarządzającego wygenerowanie nowych replik dla pliku. Test ten ma zbadać zachowanie systemu przy operacjach zapisu danych w przypadku wystąpienia częstych awarii (repliki pliku ulegają awarii w niewielkim przedziale czasowym).

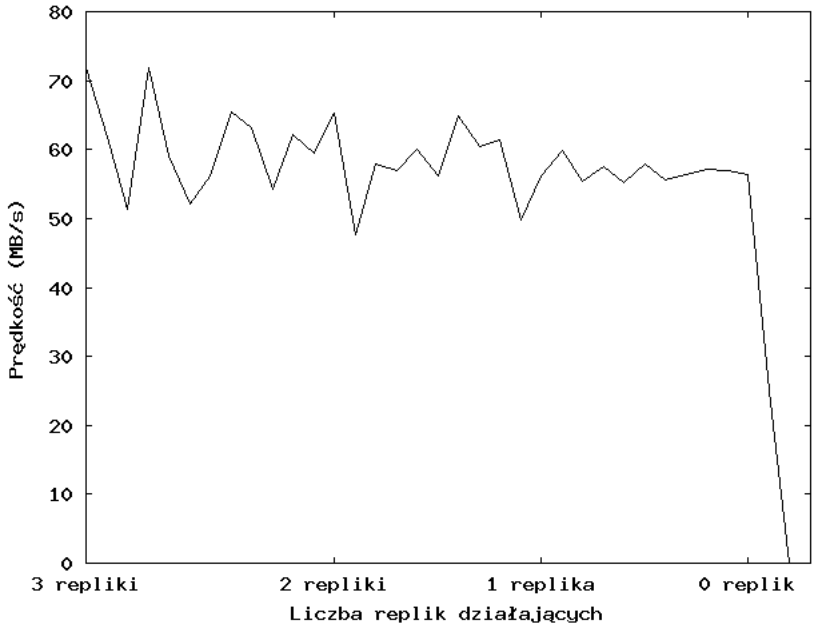
Warunki przeprowadzenia testu

Tworzymy w systemie plik składający się z trzech replik. Uruchamiamy 10 procesów klienckich zapisujących do pliku rekordy. Po 30 sekundach na pierwszym komputerze z lokalizacji pliku uruchamiamy w pętli nieskończonej nowe procesy (komputer przestaje odpowiadać na zapytania). Następnie po 30 sekundach odłączamy (fizycznie) drugi komputer od sieci. Po kolejnych 30 sekundach, na trzeciej replice zabijamy proces serwera danych.

W teście badamy sumaryczną prędkość zapisu danych do systemu, która się zmienia wraz ze zmianą liczby procesów klienckich.

Wyniki

Rysunek 3.9 przedstawia wyniki testu.



Rysunek 3.9: Zapis rekordów przy częstych awariach

Wnioski

Z przedstawionego wykresu wynika, że system jest odporny na częste awarie replik pliku do momentu gdy zabraknie działających replik. Pomimo awarii replik prędkość zapisu rekordów praktycznie nie zmieniła się. Dopiero po wyłączeniu ostatniej repliki system przestał przyjmować dane do pliku. Jednak sytuacje, w których wszystkie repliki ulegają awarii zdarzają się stosunkowo rzadko. Zwykle przerwy między kolejnymi awariami są znacznie dłuższe, a wtedy zachowanie systemu również będzie inne.

3.3.2. Zapis rekordów do systemu przy normalnych awariach komputerów

Testujemy zachowanie systemu w czasie operacji zapisu przy różnego rodzaju awariach. Awarie będą wprowadzane co 2 minuty, aby umożliwić procesowi serwera zarządzającego wygenerowanie nowych replik dla pliku. Test ten ma zbadać zachowanie systemu przy operacjach zapisu danych w przypadku wystąpienia awarii.

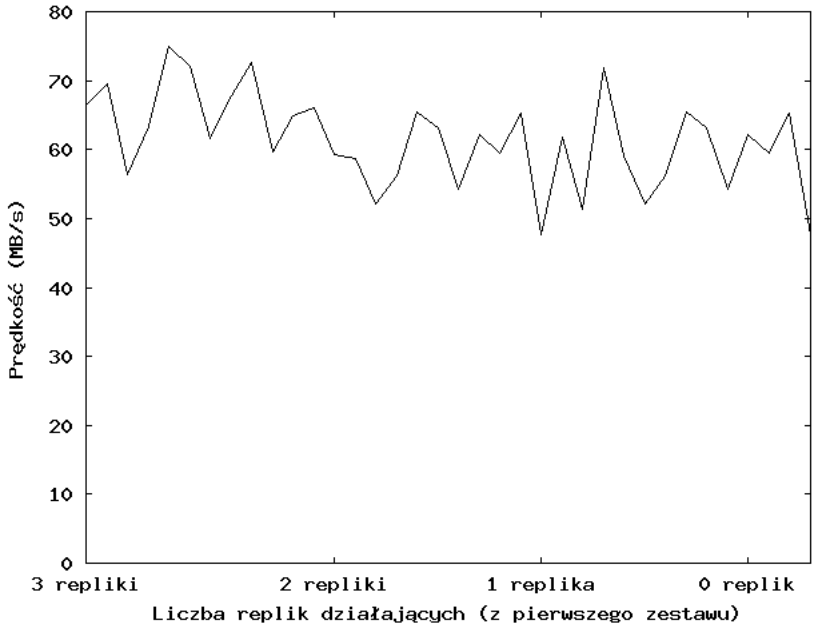
Warunki przeprowadzenia testu

Tworzymy w systemie plik składający się z trzech replik. Uruchamiamy 10 procesów klienckich zapisujących do pliku rekordy. Po 2 minutach na pierwszym komputerze z lokalizacji pliku uruchamiamy w pętli nieskończonej nowe procesy (komputer przestaje odpowiadać na zapytania). Następnie po 2 minutach odłączamy (fizycznie) drugi komputer od sieci. Po kolejnych 2 minutach, na trzeciej replice zabijamy proces serwera danych.

W teście badamy sumaryczną prędkość zapisu danych do systemu ze wszystkich procesów klienckich, która się zmienia w czasie.

Wyniki

Rysunek 3.10 przedstawia wyniki przeprowadzonego testu.



Rysunek 3.10: Zapis rekordów przy normalnych awariach

Wnioski

Z przedstawionych wyników widać, że system jest odporny na awarie, jeżeli tylko damy mu wystarczającą ilość czasu na regenerację. Po 2 minutach serwer zarządzający uznaje nieodpowiadający komputer za nieaktywny i generuje nową replikę która przejmuje zadania. Wydajność działania systemu pozostaje przez cały czas taka sama.

Z przedstawionego wykresu wynika, że system jest odporny na częste awarie replik pliku do momentu gdy zabraknie działających replik. Pomimo awarii replik prędkość zapisu rekordów praktycznie nie zmieniła się. Dopiero po wyłączeniu ostatniej repliki system przestał przyjmować dane do pliku. Jednak sytuacje, w których wszystkie repliki ulegają awarii zdarzają się stosunkowo rzadko. Zwykle przerwy między kolejnymi awariami są znacznie dłuższe, a wtedy zachowanie systemu również będzie inne.

3.3.3. Odczyt rekordów z systemu przy częstych awariach komputerów

Testujemy zachowanie systemu w czasie operacji odczytu przy różnego rodzaju awariach. Awarie będą wprowadzane często (co 30 sekund), aby uniemożliwić procesowi serwera zarządzającego wygenerowanie nowych replik dla pliku. Test ten ma zbadać zachowanie systemu przy operacjach odczytu danych w przypadku wystąpienia częstych awarii (repliki pliku ulegają awarii w niewielkim przedziale czasowym).

Warunki przeprowadzenia testu

W systemie tworzymy plik składający się z trzech replik. Do pliku zapisujemy 10 GB danych w rekordach po 1 MB. Uruchamiamy 10 procesów klienckich czytających losowe rekordy z pliku.

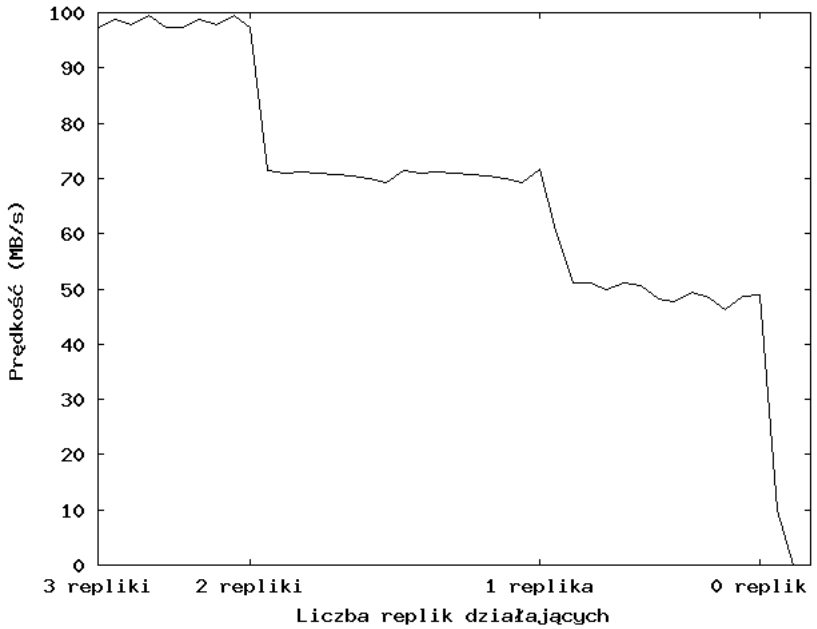
Po 30 sekundach na pierwszym komputerze z lokalizacji pliku uruchamiamy w pętli nieskończonej nowe procesy (komputer przestaje odpowiadać na zapytania). Następnie po 30 se-

kundach odłączamy (fizycznie) drugi komputer od sieci. Po kolejnych 30 sekundach, na trzeciej replice zabijamy proces serwera danych.

W teście badamy sumaryczną prędkość odczytu danych do systemu ze wszystkich procesów klienckich, która się zmienia w czasie (wraz ze zmianą liczby replik).

Wyniki

Rysunek 3.11 przedstawia wyniki przeprowadzonego testu.



Rysunek 3.11: Odczyt rekordów przy częstych awariach

Wnioski

Na rysunku 3.11 widać, że po awarii pierwszej repliki prędkość odczytu danych znacznie spadła. Wynika to z faktu, że pozostałe repliki musiały przejąć nowe zapytania. Po wyłączeniu drugiej repliki prędkość spadła jeszcze bardziej. Wyłączenie trzeciej repliki spowodowało zatrzymanie serwowania danych przez system.

Z testu wynika, że system potrafi sobie poradzić z częstymi awariami, mimo, że odbywa się to kosztem spadku wydajność. Jeżeli jednak pozwolimy na regenerację replik, to zachowanie systemu będzie jeszcze lepsze.

3.3.4. Odczyt rekordów z systemu przy normalnych awariach komputerów

Testujemy zachowanie systemu w czasie operacji odczytu przy różnego rodzaju awariach. Test ten ma zbadać zachowanie systemu przy operacjach odczytu danych w przypadku wystąpienia awarii.

Warunki przeprowadzenia testu

W systemie tworzymy plik składający się z trzech replik. Do pliku zapisujemy 10 GB danych w rekordach po 1 MB. Uruchamiamy 10 procesów klienckich czytających losowe rekordy z

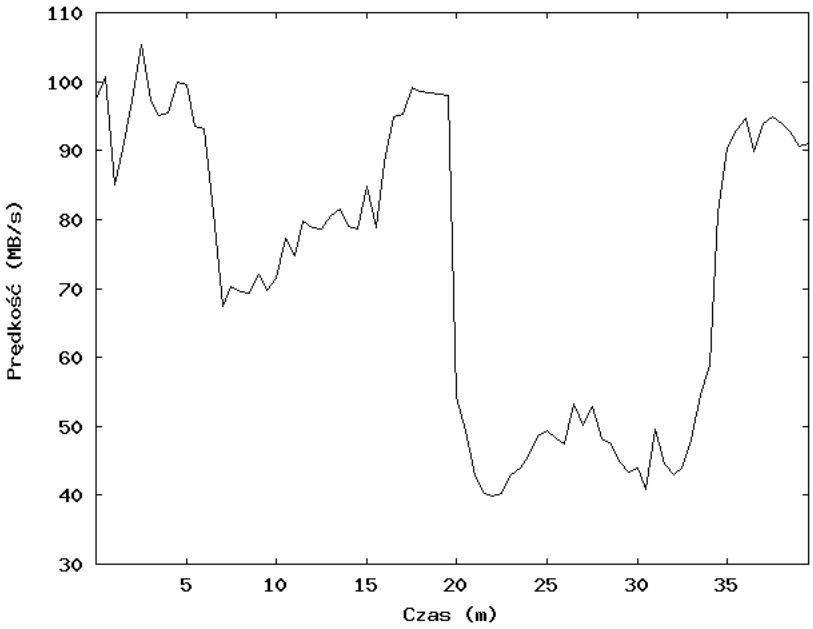
pliku.

Po 5 minutach działania systemu wprowadzimy pierwszą awarię poprzez uruchomienie procesów w pętli nieskończonej na jednej z replik. Następnie pozwolimy systemowi na regenerację, po której zabijemy procesy serwera danych na dwóch pozostałych replikach pliku.

W teście badamy sumaryczną prędkość odczytu danych z systemu ze wszystkich procesów klienckich, która się zmienia w czasie (wraz ze zmianą stanu systemu).

Wyniki

Rysunek 3.12 przedstawia wyniki testu.



Rysunek 3.12: Odczyt rekordów przy awariach

Wnioski

Po wprowadzeniu pierwszej awarii w 5-tej minucie działania systemu spada znacznie prędkość odczytu danych. Po 2 minutach serwer zarządzający rozpoczyna odtwarzanie danych na nowej replice. Około 16-ej minuty testu dane są odtworzone i system odzyskuje pierwotną sprawność. W 20-ej minucie wprowadzamy równocześnie awarie na dwóch replikach pliku. Tym razem spadek prędkości działania jest znacznie większy. Ok. 35-ej minuty testu system powraca do pierwotnej sprawności.

Widać wyraźnie, że system jest odporny na awarie zarówno pojedynczych replik, jak i kilku. Jeżeli stworzymy w systemie plik z większą liczbą replik, to ta odporność będzie jeszcze większa.

3.3.5. Działanie systemu przy awarii bazy danych lokalizacji plików

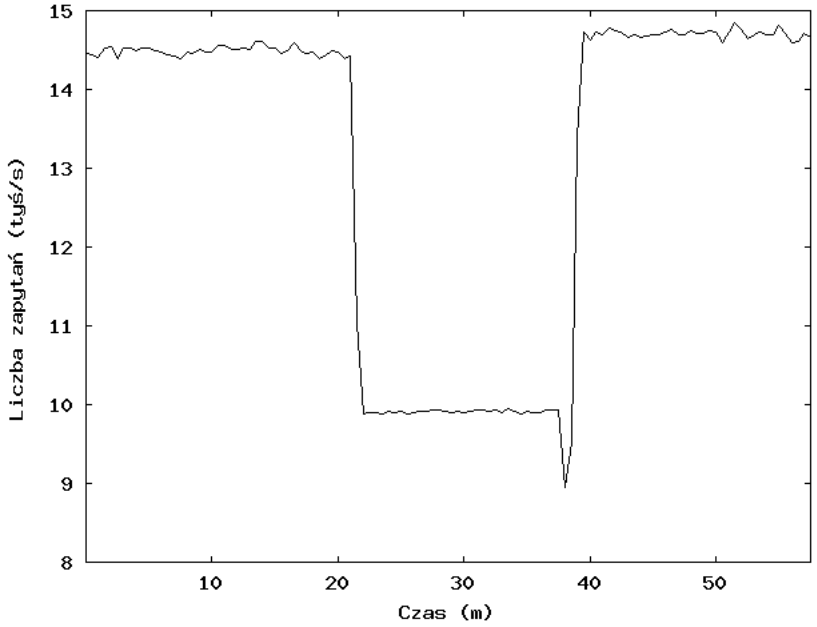
Baza danych lokalizacji plików jest jedynym *pojedynczym punktem awarii* systemu. Większość zapytań klientów odwołuje się do bazy danych w celu pobrania lokalizacji plików w systemie. Bardzo ważne jest więc zapewnienie odpowiedniego poziomu bezpieczeństwa danych lokalizacyjnych.

Warunki przeprowadzenia testu

W kolejnym teście sprawdzane jest działanie mechanizmu kopii bezpieczeństwa bazy danych lokalizacji. W tym celu podłączamy do systemu jeszcze jeden komputer (z nieco słabszym procesorem), na którym uruchomiony zostanie program kopii bazy danych (*SlaveDB*). Program ten będzie synchronizował zawartość pamięci podręcznej z główną bazą danych, a następnie w sytuacji awarii bazy głównej będzie odpowiadał na zapytania klientów o lokalizację plików.

W teście tworzymy w systemie milion plików. Uruchamiamy 10 procesów klienckich odpytujących bazę danych o lokalizacje losowo wybranego pliku (w pętli nieskończonej). Badamy liczbę zapytań o lokalizacje, którą mogą wygenerować klienci systemu. Po 20 minutach zabijamy proces głównej bazy danych. Po kolejnych 20 minutach przywracamy działanie głównej bazy danych.

Wyniki



Rysunek 3.13: Awaria głównej bazy danych lokalizacji

Wnioski

Rysunek 3.13 przedstawia działanie systemu w czasie awarii głównej bazy danych. Widzimy wyraźnie moment, w którym zapytania lokalizacyjne zostały przejęte przez kopię bazy danych. Po ponownym uruchomieniu głównej bazy danych wydajność zapytań lokalizacyjnych wraca do normy.

3.4. Wnioski z testów

Przeprowadzone testy wydajności wykazały, że system dobrze radzi sobie z wymaganiami stawianymi przez środowiska, w których ma być wykorzystywany. System jest w stanie przechowywać dużą liczbę małych obiektów, co pokazały testy z wykorzystaniem rekordów o

wielkości 1 KB. Nie ma również problemów z obsługą wielu równoległych klientów i równoległymi dostępami do wielu obiektów. Opóźnienia generowane przez system również są minimalne, co również było widać w testach wydajnościowych. System nie ma również problemów z przechowywaniem dużej ilości danych. Na platformie testowej złożonej z ośmiu komputerów z danymi utrzymywane było kilkaset gigabajtów danych przez cały czas przeprowadzania testów. Przy odpowiedniej liczbie komputerów nie byłoby żadnych problemów z pomieszczeniem większej ilości danych.

Dzięki zastosowaniu zwielokrotnienia danych oraz odpowiednich algorytmów monitorujących pracę systemu, uzyskano odporność na różnego rodzaju awarie, zarówno chwilowe, jak i rozciągnięte w czasie. Zostało to dobrze pokazane w testach funkcji systemu (bezpieczeństwa danych). Testy, w których zapisywanych było wiele małych rekordów pokazały także, że model spójności ostatecznej bardzo dobrze sprawdza się w zastosowaniach, dla których system był projektowany.

Dzięki zastosowaniu kopii bazy danych lokalizacji (*SlaveDB*) udało się wyeliminować z systemu pojedynczy punkt awarii, co pokazał test działania systemu przy awarii bazy danych lokalizacji.

Skalowalność systemu została zbadana dzięki testom przeprowadzonym z wykorzystaniem wielu równoległych klientów. Testy te pokazały, że system, nawet przy niewielkiej liczbie węzłów, jest w stanie obsługiwać wiele tysięcy operacji na sekundę i przesyłać dane z prędkościami zbliżonymi do granic możliwości sprzętowych komputerów.

Pozostałe cele projektowe, takie jak konfigurowalny stopień zabezpieczeń, proste API, czy prosta obsługa, zostały spełnione już na etapie projektowania systemu.

Rozdział 4

Podsumowanie

Celem pracy było zaprojektowanie i zaimplementowanie systemu, dla którego wymagania zostały wyspecyfikowane w rozdziale pierwszym. Po przeprowadzaniu kilkunastu testów wydajności i dużej liczby testów funkcjonalności systemu, można stwierdzić, że cel ten został osiągnięty.

W pracy przedstawione zostały problemy związane z projektowaniem rozproszonych systemów plików. Wybrane metody rozwiązania tych problemów zostały zaimplementowane w systemie. Na szczególną uwagę zasługuje zastosowany model spójności ostatecznej oraz związane z nim algorytmy synchronizacji danych, których wykorzystanie w znacznym stopniu przyczyniło się do pozytywnego zakończenia projektu. Zwielokrotnianie danych umożliwiło zapewnienie odpowiedniego poziomu ochrony danych i pozwoliło na utrzymywanie wysokiej ich dostępności.

Bardzo ważną cechą systemu jest także wykorzystanie najnowszych funkcji systemowych związanych z operacjami wejścia-wyjścia. Dzięki ich zastosowaniu znacznie poprawiona została wydajność systemu, a jego kod uproszczony.

W systemie udało się także uniknąć (w odniesieniu do większości operacji) pojedynczego punktu awarii. Zostało to osiągnięte dzięki wykorzystaniu kopii bazy danych lokalizacji.

4.1. Dalszy rozwój systemu

System był tworzony głównie z myślą o przechowywaniu danych serwisów poczty elektronicznej oraz serwisów z plikami multimedialnymi. W pracy pokazano, że wymagania stawiane przez te serwisy zostały spełnione. Istnieje jednak kilka pomysłów, których realizacja mogłaby umożliwić znaczne rozszerzenie listy zastosowań systemu.

4.1.1. Dynamicznie ustalany poziom replikacji

W stworzonym systemie dane przechowywane są w wielu kopiach. Liczbę tych kopii (replik) określa poziom replikacji ustalany przy tworzeniu pliku. Takie rozwiązanie jest wystarczające dla większości zastosowań. Czasami jednak zdarzają się sytuacje, w których część plików jest bardzo intensywnie wykorzystywana przez jakiś czas. Mamy z tym do czynienia w wielu serwisach internetowych, gdzie materiały prezentowane na stronie głównej serwisu są znacznie częściej odczytywane niż pozostałe. Może wtedy dojść do przeciążenia komputerów przechowujących te obiekty i spadku dostępności danych. Chcąc rozwiązać ten problem można wprowadzić dynamiczny poziom replikacji, który umożliwi przydział dodatkowych replik dla

pliku gdy zostanie wykryte przeciążenie (np. poprzez monitorowanie odpowiedzi systemu w bibliotece klienckiej).

Czasami mamy także do czynienia z sytuacją odwrotną. Część danych systemu jest wykorzystywana bardzo rzadko albo wcale. W większości sytuacji nie warto przechowywać takich danych w wielu kopiach. Można wtedy zmniejszyć liczbę replik i odzyskać część miejsca na dyskach twardych.

4.1.2. Obsługa komputerów w wielu lokalizacjach fizycznych

Rozproszone systemy plików zwykle wykorzystują komputery znajdujące się w wielu serwerowniach, które są położone w różnych lokalizacjach fizycznych. Dzięki temu, że nie istnieje żadna zależność między serwerowniami (posiadają niezależne zasilania, są niezależnie podłączone do sieci komputerowych, etc.) oraz zwykle są od siebie znacznie oddalone, uzyskuje się większe bezpieczeństwo danych. W obecnej wersji systemu wszystkie komputery są traktowane tak, jakby znajdowały się w jednej serwerowni (nawet jeżeli znajdują się w kilku). Aby poprawić bezpieczeństwo danych w przypadku zaistnienia dużych awarii (związanych z całą serwerownią) należałoby dodać do serwera zarządzającego mechanizm umożliwiający rozpoznanie fizycznego położenia komputera i uwzględnianie tej informacji przy generowaniu replik dla plików. Oczywiście dotyczy to tylko sytuacji, w których system będzie działał w środowisku wykorzystującym wiele niezależnych serwerowni.

4.1.3. Udostępnienie danych przez NFS

Twórcy serwisów internetowych bardzo często wykorzystują katalogi dyskowych systemów plików, aby oddać hierarchiczne zależności między przechowywanymi danymi. Stworzony w ramach pracy system nie pozwala na tworzenie katalogów. Można to jednak rozwiązać traktując pliki systemu jak katalogi, do których będziemy dopisywać dane (rekordy mogą reprezentować wpisy katalogowe). Następnie dane te można udostępnić klientom zewnętrznym poprzez wprowadzenie do systemu pośrednika, który tłumaczyłby operacje w systemie na komunikaty protokołu sieciowego (np. NFS – *Network File System*).

4.1.4. Uzyskanie pełniej spójności danych

W niektórych sytuacjach wymaga się, aby rozproszony system plików zapewniał pełną spójność danych. Stworzony system nie spełnia takiego wymagania, gdyż był projektowany z myślą o innych zastosowaniach. Przyjęcie rozluźnionego modelu spójności pozwoliło na zwiększenie efektywności działania, ale jednocześnie wykluczyło niektóre obszary zastosowania systemu. Możliwe jest jednak, przy rezygnacji z pewnych cech systemu, uzyskanie pełnej spójności utrzymywanych danych. Poprzez dodanie do systemu warstwy pośredniej w postaci serwera rozstrzygającego konflikty wersji obiektów (konflikty powstające przy zapisie i usuwaniu danych), możemy zapewnić spójny obraz systemu przez cały czas działania. Takie rozwiązanie pociąga za sobą jednak istnienie w systemie pojedyńczego punktu awarii, który jednocześnie staje się wąskim gardłem w komunikacji z klientami. W wielu przypadkach może okazać się to jednak wystarczające (np. w zastosowaniach, w których nie ma wysokich wymagań dotyczących niezawodności systemu).

Dodatek A

Zawartość płyty CD

Na załączonej płycie CD znajdują się następujące pozycje:

- **kwitkowski.pdf** – treść pracy magisterskiej w formacie PDF,
- **kwitkowski-src/** – treść pracy magisterskiej w formacie LaTeX,
- **system-src/** – kod źródłowy systemu,
  - **client/** – biblioteka kliencka systemu,
  - **datanode/** – kod serwera danych,
  - **masterdb/** – główna baza danych lokalizacji,
  - **net/** – komponenty odpowiedzialne za komunikację sieciową,
  - **slavedb/** – kopia bazy danych lokalizacji,
  - **sysmanager/** – kod serwera zarządzającego
  - **utils/** – klasy narzędziowe systemu,
- **system-tests/** – skrypty i programy służące do przeprowadzania testów systemu,
- **test-results/** – wyniki przeprowadzonych testów przeprowadzonych.

Bibliografia

[1] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, *Farsite: federated, available, and reliable storage for an incompletely trusted environment*, 5th Symposium on Operating Systems Design and Implementation, 2002

[2] Mordechai Ben-Ari, *Podstawy programowania współbieżnego i rozproszonego*, WNT 2009

[3] Nicolas Bonvin, Thanasis G. Papaioannou, Karl Aberer, *Dynamic cost-efficient replication in data clouds*, International Conference on Autonomic Computing, 2009

[4] George Coulouris, Jean Dollimore, Tim Kindberg, *Systemy rozproszone. Podstawy i projektowanie*, WNT 1999

[5] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, Werner Vogels, *Dynamo: Amazon's Highly Available Key-value Store*, ACM SIGOPS Operating Systems Review, 2007

[6] Sanjay Ghemawat, Howard Gobioff, Shun-Tak Leung, *The Google File System*, ACM SIGOPS Operating Systems Review, 2003

[7] Strona domowa projektu Apache Hadoop, http://hadoop.apache.org/

[8] Hsien-Tsung Chang, *Load balancing and fault-tolerance for scalable network file systems using by web services*, WSEAES 13th International Conference on Computers, 2009

[9] Robert Love, *Linux. Programowanie systemowe*, Helion 2008

[10] Strona domowa projektu MooseFS, http://www.moosefs.org/

[11] Yasushi Saito, Svend Frolund, Alistair Veitch, Arif Merchant, Susan Spence, *FAB: building distributed enterprise disk arrays from commodity components*, ACM SIGARCH Computer Architecture News, 2004

[12] Richard W. Stevens, *UNIX programowanie usług sieciowych*, WNT 2002

[13] Andrew S. Tanenbaum, Maarten van Steen, *Systemy rozproszone. Zasady i paradygmaty*, WNT 2006

[14] Sage Weil, *Ceph: Reliable, Scalable, and High-Performance Distributed Storage*, http://ceph.newdream.net/weil-thesis.pdf, 2007

[15] Artykuł kompromisach pomiędzy spójnością a dostępnością danych w systemach rozproszonych, Werner Vogels, *Eventually Consistent*, http://www.allthingsdistributed.com/2008/12/eventually_consistent.html, 2008

[16] Di Wu, Ye Tian, Kam-Wing Ng, *Resilient and efficient load balancing in distributed hash tables*, Jurnal of Network and Computer Applications, 2009

[17] Artykuł na temat technik *zero-copy I/O* w systemie Linux, http://www.linuxjournal.com/article/6345