Uniwersytet Warszawski
Wydział Matematyki, Informatyki i Mechaniki

Tomasz Weksej
Nr albumu: 219722

Niezawodność w rozproszonych systemach bazodanowych

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

W pracy przedstawię projekt i implementację mechanizmu odpowiedzialnego za zapewnianie niezawodności w systemie rozproszonym, stanowiącego część Gemius BigTable — implementacji rozproszonego systemu bazodanowego inspirowanego podobnym rozwiązaniem wydanym w 2006 roku przez Google.

W systemach rozproszonych, jako że są one bardziej skomplikowane od systemów scentralizowanych, zapewnianie niezawodności jest kluczowe — mamy tu do czynienia z awariami części systemu w czasie jego pracy czy współbieżnością, nie tylko na poziomie procesów, ale i komputerów.

W pracy opiszę kilka istniejących rozwiązań, zdefiniuję na czym polega zapewnianie niezawodności w systemie rozproszonym oraz opiszę zarówno stworzone rozwiązanie, jak i proces jego testowania.

Słowa kluczowe

distributed database system, reliability, fault tolerance, high availability, recoverability

Dziedzina pracy (kody wg programu Socrates-Erasmus)

11.3 Informatyka

Klasyfikacja tematyczna

D. Software
D.4 Operating Systems
D.4.5 Reliability

Tytuł pracy w języku angielskim

Reliability in distributed database systems

Spis treści

Wprowadzenie

Rozproszona baza danych (ang. *distributed database*, DDB) to zbiór logicznie powiązanych baz danych rozproszonych w sieci komputerowej. **System zarządzania rozproszoną bazą danych** (ang. *distributed database management system*, D-DBMS) to oprogramowanie zarządzające DDB i zapewniające mechanizm dostępu do danych ukrywający przed użytkownikiem fakt ich fizycznego rozproszenia (*transparentny* dostęp do danych). **Rozproszony system bazodanowy** (ang. *distributed database system*, DDBS) to rozproszona baza danych wraz z jej systemem zarządzania ([Purd06]).

DDBS są wybierane przez rozmaite organizacje ze względu na ich przewagę nad standardowymi systemami bazodanowymi (DBMS) na kilku płaszczyznach (za [Elma03]):

1. **Transparentny dostęp do danych** — dostęp do danych jest realizowany w identyczny sposób, niezależnie od fizycznej obecności danych w systemie (*transparentność dystrybucji*), co jest znacznie prostsze w użyciu niż zarządzanie kilkoma niezależnymi DBMS.

2. **Zwiększona niezawodność i dostępność** — często uważana za największą zaletę DDBS. **Niezawodność** jest definiowana jako prawdopodobieństwo działania systemu (bycia *dostępnym*) w danym czasie, natomiast **dostępność** jako prawdopodobieństwo że system jest dostępny przez cały określony przedział czasu. W związku z tym, że dane przechowywane przez DDBS są rozdystrybuowane pomiędzy kilka elementów, awaria niektórych z nich nie wpływa na dostępność danych przechowywanych przez działające elementy. Dalsze usprawnienie może polegać na **replikacji** danych (przechowywania danych w kilku kopiach przez kilka elementów). Zwiększa to zarówno niezawodność, jak i dostępność. Dla kontrastu, w scentralizowanych systemach bazodanowych awaria jedynego węzła oznacza niedostępność całego systemu.

3. **Zwiększona wydajność** — w DDBS zadania zlecone przez użytkownika przetwarzane są równolegle przez kilka elementów, co oczywiście jest szybsze niż przetwarzanie ich sekwencyjnie przez jeden. Poza tym rozproszenie bazy danych między kilka węzłów skutkuje tym, że każdy z elementów zarządza mniejszą bazą niż oryginalna, a więc wszystkie lokalne operacje są bardziej wydajne.

4. **Zwiększona skalowalność** — aby osiągnąć większą wydajność wystarczy po prostu zwiększyć liczbę węzłów tworzących system (czyli *dodać* podzespoły). Jest to znacznie kosztowniejsze w przypadku systemów scentralizowanych, gdzie polega to na *wymianie* podzespołów.

Niezawodność w rozproszonych systemach bazodanowych

Nie ma jednak róży bez kolców — wymienione zalety są okupione dużo większym skomplikowaniem DDBS. Musi być wykonana dodatkowa praca, aby: (1) zapewnić transparentność systemu; (2) zarządzać wieloma węzłami wchodzącymi w skład systemu; (3) zapewnić niezawodność w systemie — odpowiednio synchronizować operacje wykonywane w każdym z węzłów czy reagować na awarie elementów systemu i podtrzymywać jego sprawność. Właśnie zapewnianie niezawodności w rozproszonych systemach bazodanowych jest tematem tej pracy magisterskiej.

Cel pracy

Do dzisiaj DDBS doczekały się kilku implementacji. Za pierwszą można uznać Google BigTable (wydaną w 2006 roku), która okazała się swego rodzaju "proof of concept" — uruchamiając system na tysiącach zwykłych, tanich komputerów klasy PC udało się osiągnąć wydajność superkomputerów. Nie zostało ujawnionych wiele informacji o szczegółach technicznych tego projektu, ale sam fakt, że z BigTable korzysta wiele usług Google obsługujących naraz miliony użytkowników, zadziałał inspirująco na twórców oprogramowania rozproszonego i kilka firm i instytucji starało się stworzyć podobne rozwiązanie. W wyniku tego powstały m.in. wydane jako *open source* systemy HBase i Hypertable.

Przydatność DDBS jest najbardziej widoczna w projektach przetwarzających duże ilości danych. Jedną z polskich firm realizujących tego typu przedsięwzięcia jest **Gemius**. Gemius to firma zajmująca się badaniami Internetu, analizująca sporą część ruchu w polskim i środkowoeuropejskim Internecie. Początkowo starano się wykorzystać istniejące systemy o ogólnie dostępnych źródłach, jednak na przeszkodzie stanęły: (1) mała stabilność ówczesnych ich wersji; (2) model danych niepozwalający na bezpośrednie wykorzystanie rozwijanych przez wiele lat przez Gemius metod kompresji. Więcej o istniejących systemach będzie mowa w rozdziale 1.

To wszystko wymogło na firmie Gemius stworzenie własnej, autorskiej implementacji DDBS. Jest ona opracowywana przez kilkuosobowy zespół, w skład którego wchodzi m.in. autor. Gdy dołączałem do projektu był on w bardzo wstępnej fazie realizacji, dostarczając jedynie część ze swojej docelowej funkcjonalności i nie zapewniając niezawodności.

Celem pracy magisterskiej jest zaprojektowanie i zaimplementowanie w Gemius BigTable części realizującej zapewnianie niezawodności.

W rozdziale 1 dokonamy przeglądu istniejących rozproszonych systemów bazodanowych. W rozdziale 2 przedstawimy system rozwijany przez autora: Gemius BigTable. W rozdziale 3 sprecyzujemy na czym polega **zapewnianie niezawodności** w DDBS oraz przyjrzymy się odpowiedzialnej za to części systemu stworzonej przez autora. Natomiast w rozdziale 4 omówimy metody testowania poprawności implementacji.

Podziękowania

Dziękuję wszystkim osobom które przyczyniły się do powstania tej pracy. Dziękuję pani dr Janinie Mincer-Daszkiewicz za wsparcie i opiekę merytoryczną nad pracą. Dziękuję panom dr Aleksemu Schubertowi i Piotrowi Taborowi za konsultacje dotyczące projektu protokołów wykorzystywanych w systemie. Specjalne podziękowania należą się moim współpracownikom przy projekcie: Mariuszowi Gądarowskiemu (lider zespołu), Jakubowi Boguszowi, Kamilowi

Nowosadowi i Jarkowi Wódce. Dziękuję też firmie Gemius za możliwość opublikowania wypracowanych rozwiązań w formie pracy magisterskiej i za odwagę powierzenia studentowi całości prac dotyczących zapewniania niezawodności systemu.

Rozdział 1

Rozproszone bazy danych

W tym rozdziale opiszemy istniejące implementacje DDBS: Google BigTable, będące jej dość wierną kopią HBase i Hypertable oraz powstały niedawno, prezentujący zupełnie inne podejście, Apache Cassandra.

1.1. Google BigTable

Google BigTable (BT) jest implementacją DDBS wykonaną w latach 2004-06 przez firmę Google. Dziś jest wykorzystywana jako baza danych przez takie serwisy jak Gmail, YouTube, Google Maps, Google Reader, hosting Google Code, Google Earth, Google Books, Blogger.com, czy Orkut ([WikiBT]).

Do dziś BT pozostaje wewnętrznym projektem firmy Google i nie jest udostępniony do (bezpośredniego) użytku zewnętrznego. Co więcej, zostało ujawnionych bardzo niewiele technicznych informacji na jego temat — jedynym źródłem jest [BTab06], dokument przygotowany na 7. konferencję *Operating Systems Design and Implementation* w 2006 roku (OSDI '06), i zawierający bardzo ogólny opis przeznaczenia, architektury i działania systemu, bez dokładniejszych szczegółów technicznych.

[BTab06] posłużył jako inspiracja do stworzenia Gemius BigTable. Architektura GBT, bardzo podobna do BT, została opisana w rozdziale 2, więc w tym rozdziale opiszemy ją bardzo ogólnie, bliżej przyglądając się znanym różnicom między tymi dwoma systemami.

BT jest zaprojektowane, by obsługiwać petabajty danych na "setkach lub tysiącach maszyn, oraz by umożliwić łatwe dodawanie nowych maszyn do systemu i natychmiastowe wykorzystywanie ich na rzecz systemu bez dodatkowej rekonfiguracji" ([WikiBT]). Model danych przechowywanych przez BT łączy cechy zarówno wierszowo, jak i kolumnowo zorientowanych baz danych. BT jest rozproszoną, rzadką, stałą wielowymiarową posortowaną mapą. Mapa ta jest indeksowana kluczem wiersza (*row key*), kluczem kolumny (*column key*), sygnaturą czasową (*timestamp*); każda wartość przechowywana przez mapę jest pewnym odgórnie nieinterpretowanym ciągiem bajtów:

$$(\text{row:string}, \text{column:string}, \text{time:int64}) \rightarrow \text{string}$$

Dodatkowy parametr (sygnatura czasowa) pozwala na wersjonowanie przechowywanych przez BT danych. Jest to podyktowane wymogami projektowymi: analizując potencjalne zastosowania BT jego autorzy doszli do wniosku, że będzie należeć do nich przechowywanie nie tylko samych danych, lecz także ich wcześniejszych kopii. Natomiast kolumny są zgrupowane w zbiory nazywane **rodzinami kolumn** (ang. *column family*), które stanowią podstawową jednostkę dostępu do danych. W intencji twórców leży, aby liczba rodzin kolumn w tabeli

była niewielka (rzędu setek), natomiast rozmiar każdej z rodzin (liczba kolumn) jest nieograniczony.

BT przechowuje dane jako pliki w rozproszonym systemie plików Google File System ([GFS03]). Dane są podzielone na **tablety** — ciągłe podzbiory rekordów z danej tabeli, każdy o rozmiarze ok. 200 MB, wydajnie skompresowane. Do kompresji używa się algorytmów *BMDiff* (algorytm optymalny przy kompresji zbioru danych o podobnej wartości, opierający się na przechowywaniu jedynie różnicy pomiędzy wartościami poszczególnych elementów; szerzej opisany w [BMDiff99]) oraz *Zippy* (nieujawniony algorytm, mniej optymalny od LZO jeśli chodzi o stopień kompresji, ale wydajniejszy jeśli chodzi o czas kompresji/dekompresji). Tak skompresowane dane przechowywane są w plikach o formacie *SSTable*, który jest podobny do formatu *CommonLib*, opisanego w rozdziale 2.4.1. Informacja o lokalizacji tabletów jest przechowywana w dwóch specjalnych tabletach nazywanych `META0` i `META1` (**metatablety**). Pierwszy z nich zawiera informacje o lokalizacji tabletów `META1`, które z kolei przechowują informacje o lokalizacji tabletów z danymi. Tablet `META0` jest przechowywany przez **serwer główny** (MS), natomiast pozostałe tablety są przechowywane przez **serwery tabletów** (TS). Ich rola w systemie jest podobna jak w przypadku Gemius BigTable (rozdział 2).

Podobnie jak to wygląda w przypadku Gemius BigTable, w BT u podstaw trwałości danych leży rozproszony system plików i metoda replikacji. Natomiast by zapewnić spójność systemu i poprawną reakcję na awarie jego składników, korzysta się z rozproszonego systemu blokad o nazwie **Chubby**. Chubby został opisany w [Chub06], udostępnia przestrzeń nazw składającą się z katalogów i plików, każdy z nich może zostać użyty jako blokada; odczyty i zapisy do plików są atomowe. Każdy z klientów utrzymuje **sesję** z serwisem Chubby; gdy jest ona zakończona (np. w wyniku utraty połączenia), wszystkie założone w jej czasie blokady są utracone. Klienci mogą także rejestrować **wywołania zwrotne** (ang. *callbacks*) na plikach i katalogach Chubby, aby otrzymywać powiadomienia o zmianie ich stanu.

BigTable korzysta z Chubby m.in. aby reagować na awarie serwerów tabletów. Gdy serwer tabletów rozpoczyna swą pracę, tworzy w odpowiednim katalogu (**katalogu serwerów**) plik Chubby o unikalnej nazwie i zakłada na niego wyłączną blokadę. Serwer główny monitoruje zawartość tego katalogu, aby odkrywać nowo pojawiające się serwery tabletów. Serwer tabletów przestaje obsługiwać przydzielone mu tablety gdy utraci swoją blokadę, np. w wyniku problemów sieciowych czy zakończenia sesji Chubby. W tym przypadku będzie on próbował na nowo założyć blokadę na ten plik tak długo, jak ten plik będzie istniał; w przeciwnym przypadku praca serwera tabletów zakończy się. Serwer główny jest odpowiedzialny za monitorowanie czy TS dalej obsługują przydzielone im tablety, poprzez okresowe odpytywanie TS o stan ich blokady. W przypadku, gdy TS nie odpowiada, MS próbuje założyć wyłączną blokadę na odpowiadający temu TS plik Chubby. Powodzenie tej operacji oznacza, że TS ma problemy z dostępem do serwisu Chubby (lub zakończył swoje działanie), więc MS usuwa odpowiadający mu plik aby zapewnić, że TS nie będzie więcej obsługiwać przydzielonych mu tabletów. Wtedy MS może je rozdystrybuować do pozostałych TS. Aby BT nie był podatny na problemy sieciowe między MS a Chubby, MS kończy swoją pracę jak tylko jego sesja Chubby zakończy się. Nie zmienia to jednak przydziału tabletów do TS.

Gdy nowy MS rozpoczyna pracę, wykonuje następujące kroki: (1) MS zakłada wyłączną blokadę na specjalny plik główny Chubby, co zapewnia że naraz nie działają dwie instancje MS. (2) MS sprawdza zawartość katalogu serwerów, aby dowiedzieć się o działających TS. (3) MS komunikuje się z każdym z TS, aby dowiedzieć się jakie tablety są przez nich obsługiwane. (4) MS korzystając z informacji zawartych w tablecie `META0` sprawdza jakie tablety są obecne w systemie; jeśli są jeszcze jakieś nieobsługiwane — przydziela je TS.

Jeśli przed ostatnim krokiem okaże się, że nie wszystkie metatablety są obsługiwane, to MS kolejno przydziela je do TS, potem ma już dostęp do informacji o tabletach obecnych w

systemie.

1.2. HBase i Hypertable

Google BigTable stał się prekursorem rozproszonych systemów bazodanowych i doczekał się implementacji o ogólnie dostępnych źródłach. Zarówno **HBase**, jak i **Hypertable** są dość wiernymi kopiami produktu Google.

HBase (HB) to część szkieletu **Apache Hadoop**, w którego skład wchodzi również **Hadoop Distributed File System** (HDFS) — rozproszony system plików, który w HB spełnia podobną rolę co GFS w BigTable. Całość napisana jest w języku Java.

Hypertable (HT) nie jest rozprowadzany razem z implementacją rozproszonego systemu plików, ale jest przygotowany do współpracy z HDFS i Kosmos File System (otwarta implementacja wzorowana na GFS). Zawiera własną implementację Chubby (o nazwie Hyperspace), SSTable (CellStore). Jest napisany w języku C++.

Początkowo właśnie z Hypertable wiązano największe nadzieje w kontekście wykorzystywania ich przez Gemius. Niestety, w tym okresie był to projekt bardzo niestabilny. Co więcej, posiada odziedziczony z BigTable kolumnowo zorientowany model danych, który utrudnia wydajne wykorzystanie rozwijanych przez Gemius metod kompresji. Zdecydowano się więc na własną implementację BigTable, dostosowaną do istniejących już w Gemius technologii.

1.3. Apache Cassandra

Apache Cassandra (AC) został opublikowany niedawno, długo po rozpoczęciu prac nad GBT. W odróżnieniu od wcześniej opisanych projektów oparty jest na zupełnie innej architekturze.

AC jest DDBS rozwijanym na potrzeby firmy Facebook — posiadacza największej na świecie platformy społecznościowej (`www.facebook.com`) obsługującej w godzinach szczytu setki milionów użytkowników za pomocą dziesiątek tysięcy serwerów rozsianych po całym świecie. Przy tym rzędzie wielkości awarie sprzętu są na porządku dziennym, więc priorytetami AC są skalowalność, dostępność i szeroko pojęta niezawodność.

AC została zaprojektowana, by zaspokoić potrzeby przechowywania danych niezbędnych do przeszukiwania wiadomości (*InboxSearch problem*, IS). InboxSearch jest funkcjonalnością pozwalającą użytkownikom na przeszukiwanie ich skrzynek odbiorczych. W tym przypadku oznaczało to wymóg wydajnej obsługi miliardów zapisów do bazy dziennie oraz wydajnego skalowania wraz ze wzrostem liczby użytkowników. InboxSearch został udostępniony w czerwcu 2008 dla 100 milionów użytkowników; w 2009 korzystało już z niego 250 milionów użytkowników i Cassandra dobrze wywiązywała się ze swych zadań. Cassandra jest dostępna za darmo wraz ze swym kodem źródłowym, na licencji Apache License 2. Dziś jest wykorzystywana m.in. przez takie systemy jak Digg, Twitter, Reddit, Rackspace, Cloudkick, Cisco, SimpleGeo, Ooyala czy OpenX.

Model danych Cassandry jest wzorowany na tym w BT: **tabela** jest rozproszoną wielowymiarową mapą indeksowaną **kluczem**. **Wartość** jest ustrukturyzowanym obiektem. Klucz wierszowy jest napisem o nieograniczonym rozmiarze (choć zwykle jest to między 16 a 36 bajtów). Kolumny są zgrupowane, podobnie jak w BT, w **rodziny kolumn**, choć AC rozwija ten pomysł i rozróżnia dwa typy: **proste** i **złożone rodziny kolumn** (ang. *Simple, Super Column Families*). O tych drugich można myśleć jak o rodzinach rodzin kolumn.

W odróżnieniu od BT, wszystkie węzły tworzące AC są równoważne (a więc system ten nie ma **pojedynczego punktu awarii** (ang. *single point of failure*)). AC stosuje więc in-

ną metodę na rozdystrybuowanie danych pomiędzy węzły. AC korzysta z funkcji haszującej zachowującej porządek (czyli $a > b \Rightarrow h(a) > h(b)$), której przeciwdziedzina traktowana jest jako przestrzeń cykliczna ("pierścień") wartości (gdzie najmniejsza wartość "zawija się" po największej). Każdy z węzłów tworzących system przypisany jest do pewnej wartości, która wyznacza jego *pozycję* w pierścieniu. Obsługę każdego rekordu przydziela się odpowiedniemu węzłowi wg następującej procedury: obliczamy sygnaturę klucza rekordu, a następnie wędrujemy wzdłuż pierścienia w kierunku zgodnym z ruchem wskazówek zegara aż znajdziemy węzeł przypisany do wartości większej niż obliczona sygnatura (w rzeczywistości nie czynimy tego liniowo, każdy węzeł utrzymuje dla optymalizacji połączenia nie tylko ze swoimi sąsiadami). Węzeł ten będzie obsługiwał tenże rekord. Jedną z zalet takiego podejścia jest to, że awaria któregoś z węzłów dotknie tylko jego sąsiadów, nie wpływa w żaden sposób na pozostałe węzły.

W celu zapewnienia wysokiej dostępności, AC stosuje replikację. Każdy z rekordów występuje w N replikach, gdzie N jest współczynnikiem konfigurowalnym dla każdej instancji AC. Węzeł zarządzający danym przedziałem jest odpowiedzialny za to, by każdy z rekordów do niego należących był zreplikowany w odpowiedniej liczbie. Istnieją różne *polityki* decydowania o tym, które węzły mają przechowywać zreplikowane rekordy; od najprostszych ($N-1$ kolejnych węzłów) do bardziej skomplikowanych. Na początku działania systemu AC wybiera spośród swoich węzłów *lidera*, którego zadaniem jest instruowanie nowo przyłączających się do systemu węzłów: przydzielanie im miejsca w pierścieniu (przedziału rekordów, które obsługują) oraz przedziałów rekordów, których replikę mają przechowywać.

Decyzja o pełnym rozproszeniu architektury AC sprawia, że wykrywanie awarii węzłów (i propagacja tej informacji w systemie) jest większym wyzwaniem niż w BT. AC korzysta w tym celu ze zmodyfikowanej wersji Φ Accrual Failure Detector ([Defa04], AFD). W AFD główny pomysł polega na tym, że moduł odpowiedzialny za wykrywanie awarii węzłów nie ocenia każdego z nich w skali binarnej (działa – nie działa), ale każdemu z nich przyporządkowuje pewien współczynnik Φ oznaczający **stopień podejrzenia** (ang *suspicious level*), że dany węzeł nie działa. Każdy z węzłów przechowuje w swojej pamięci informację o przychodzących wiadomościach przynoszonych przez **algorytmy plotkujące** (ang. *gossip algorithms*) od pozostałych węzłów i na podstawie czasu ich przyjścia szacuje wartość Φ. Na podstawie wszystkich wartości Φ w poszczególnych węzłach podejmowana jest decyzja o uznaniu danego węzła za niedziałający, co skutkuje wykluczeniem go z systemu.

Więcej o AC można przeczytać w [Laks09]. Realizując GBT zdecydowano się w wielu miejscach na inne rozwiązania, tak więc obydwa te projekty nie mają ze sobą wiele wspólnego.

Rozdział 2

Gemius BigTable

W niniejszym rozdziale opisany zostanie system Gemius BigTable: jego przeznaczenie, wykorzystane techniki i technologie, architektura. Zdefiniujemy tu pojęcia, których używać będziemy w kolejnych rozdziałach.

Mój udział w projekcie nie ograniczał się tylko do stworzenia części zapewniającej niezawodność. Sporo (choć nie wszystkie) z poniższych ustaleń jest również mojego współautorstwa.

2.1. Ogólne informacje

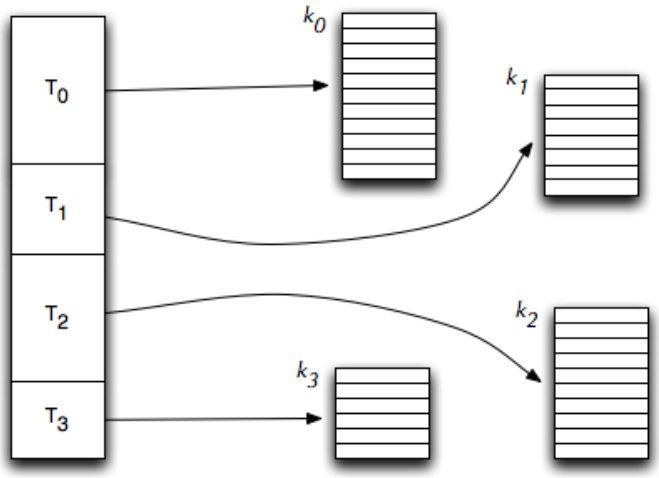
Gemius BigTable (GBT) jest rozproszonym systemem pamięci masowej, zarządzającym ustrukturyzowanymi danymi, zaprojektowanym by osiągnąć wydajną skalowalność dla dużego rozmiaru danych (rzędu petabajtów). Struktura danych przechowywanych przez GBT jest podobna do tych z relacyjnych baz danych: podstawową jednostką danych jest **rekord**, składający się z kilku **pól**, każde przyjmujące wartość ściśle określonego typu. **Typem rekordu** nazwiemy krotkę składającą się z typów pól tego rekordu. Rekordy tego samego typu tworzą **tabelę**. **Typem tabeli** nazwiemy typ rekordów przechowywanych przez nią. Specyfikacja każdej tabeli wyróżnia też pewien podzbiór pól nazywany **kluczem**. Wartości pól należących do klucza jednoznacznie wyznaczają rekord przechowywany przez tę tabelę.

2.2. Model danych

GBT, w odróżnieniu od relacyjnych baz danych, udostępnia pojedyncze tabele bez złączeń. Łatwiej więc tabelę rozumieć jako mapę (zbiór krotek *(klucz, wartość)* taki, że każda wartość klucza pojawia się w nim co najwyżej raz), a GBT jako zbiór map.

Jak już zostało powiedziane, klucz to pewien podzbiór pól. Na każdym typie pól składających się na klucz musi być określony porządek liniowy. Dzięki temu w naturalny sposób zdefiniować można porządek liniowy na kluczach, jako porządek leksykograficzny. To z kolei sprawia, że mamy określony porządek liniowy na rekordach, co jest podstawą do podziału tabeli na **tablety**.

Tablet to pewien ciągły podzbiór rekordów tabeli. Sparametryzowany jest granicami kluczy, które przechowuje. Dla przykładu, na rys. 2.1 przedstawiona jest tabela składająca się z 4 tabletów: T_1, T_2, T_3 i T_4, zawierających rekordy odpowiednio ze zbiorów $(-\infty, k_1)$, $[k_1, k_2)$, $[k_2, k_3)$ oraz $[k_3, \infty)$. Każdy z tych zbiorów jest lewostronnie domknięty.



Rysunek 2.1: Przykładowa tabela składająca się z 4 tabletów. Przy wybranych rekordach zaznaczone są ich klucze

2.3. API

API to interfejs udostępniony klientowi, umożliwiający mu dodawanie i odczyt rekordów z bazy, a także czynności administracyjne. Dokładniej rzecz biorąc, klient może stworzyć tabelę, dodać, usunąć bądź zaktualizować rekordy; iterować po rekordach należących do tabeli, usunąć tabelę oraz usunąć wszystkie znajdujące się w niej rekordy.

Na dzień dzisiejszy istnieje interfejs w języku C++. Jego możliwości prześledzimy na kilku przykładach.

```
1  using namespace BigTable;
2  Access access("192.168.0.1", 6665);
3  Transaction transaction(access);
4
5  Table<RecordType1> table1 = transaction.openTable("ExampleTable1");
6  table1.insert(record1);
7  table1.erase(key2);
8
9  Table<RecordType2> table2 = transaction.openTable("ExampleTable2");
10 table2.update(record3);
11
12 transaction.commit();
```

Listing 2.1: Przykładowe użycie API do dodania rekordów do tabeli

Listing 2.1 przedstawia przykład dodawania rekordów do tabeli. Najpierw tworzona jest struktura `Access`, reprezentująca połączenie z bazą. Konstruktor jako parametry przyjmuje adres IP i numer portu, na którym nasłuchuje aplikacja odpowiedzialna za zarządzanie GBT. Aby rozpocząć transakcję, należy utworzyć obiekt `Transaction`. Dla każdej tabeli biorącej udział w transakcji należy wywołać metodę `Transaction::openTable`, sparametryzowaną nazwą tabeli. Spowoduje to utworzenie struktury `Table<Record>`, gdzie `Record` jest klasą reprezentującą rekord przechowywany przez tę tabelę. Następnie możemy dodać, usunąć rekord lub zaktualizować jego wartość kolejno za pomocą metod `insert(Record)`, `erase(Key)`, `update(Record)`, gdzie `Key` jest klasą reprezentującą klucz tej tabeli. W powyższym przykładzie pominięto tworzenie obiektów `record1`, `key2` i `record3`.

Na koniec należy wywołać metodę `Transaction::commit`, aby zatwierdzić transakcję, lub `Transaction::rollback`, aby ją anulować. Po tym kroku obiekty typu `Transaction` oraz `Table<Record>` stworzone w ramach tej transakcji są bezużyteczne.

```
1  using namespace BigTable;
2  Access access("192.168.0.1", 6665);
3  Select<Record> select = access.makeSelect("ExampleTable");
4
5  Query<Key> query(key1, key2);
6  select.addFilter(query);
7
8  RecordStream<Record> selectResult = select.execute();
9
10 for ( ; !selectResult.end(); ++selectResult )
11     std::cout << selectResult << std::endl;
```

Listing 2.2: Przykładowe użycie API do odczytu rekordów z tabeli

Listing 2.2 przedstawia przykład odczytywania danych z tabeli. Podobnie jak w poprzednim przykładzie `Record` i `Key` są klasami reprezentującymi kolejno rekord i klucz danej tabeli.

Ponownie, najpierw tworzony jest obiekt `Access`, następnie obiekt typu `Select<Record>`, reprezentujący zapytanie do tabeli `ExampleTable`. Następnie ustawiana jest **treść** zapytania — poprzez stworzenie obiektów typu `Query<Key>` i uwzględnianie ich przy zapytaniu poprzez wywołanie metody `Select<Record>::addFilter`. Obiekty `Query<Key>` umożliwiają sparametryzowanie zapytania warunkami podobnymi do tych udostępnianych przez mechanizm zapytań SQL. W naszym przykładzie chcemy pobrać wszystkie rekordy z przedziału $[key1, key2]$. Po zdefiniowaniu warunków zapytania należy wywołać metodę `Select<Record>::execute()`, która przekaże strumień rekordów `RecordStream`.

Warty odnotowania jest fakt, że w powyższym przykładzie zostaną przekazane dane z aktualnej wersji bazy. Co za tym idzie, dwa wywołania metody `execute` wykonane w pewnym odstępstwie czasu mogą przekazać różne odpowiedzi. Czasami jest to niepożądane, często też chcemy wywołać dwa zapytania **atomowo**, tj. mając gwarancję, że przekażą wyniki odpytując bazę będącą dokładnie w tym samym stanie. Mówiąc w języku izolacji transakcji, chcielibyśmy mieć pewność, że dana sekwencja zapytań tworzy transakcję, przy poziomie izolacji SERIALIZABLE. Listing 2.3 wyjaśnia w jaki sposób można to osiągnąć.

```
1  using namespace BigTable;
2  Access access("192.168.0.1", 6665);
3  Select<Record> select = access.makeSelect("ExampleTable");
4
5  Query<Key> query(key1, key2);
6  select.addFilter(query);
7
8  Snapshot snapshot = access.makeSnapshot();
9
10 //kolejne wywolania execute(snapshot) beda dawaly dokladnie ten sam wynik
11 RecordStream<Record> selectResult1 = select.execute(snapshot);
12 RecordStream<Record> selectResult2 = select.execute(snapshot);
13
14 while ( !selectResult1.end() ) {
15     assert(selectResult1 == selectResult2);
16     ++selectResult1; ++selectResult2;
17 }
```

Listing 2.3: Przykładowe użycie API do odczytu rekordów z tabeli z użyciem `Snapshot`

Klasa `Access` umożliwia wykonanie migawki bazy danych. Warto wspomnieć, że ta operacja jest bardzo *tania* — wykonywana przez GBT w czasie stałym, niezależnym od rozmiaru danych w bazie czy liczby transakcji aktualnie wykonywanych; ponadto nie wpływa w żaden sposób na inne transakcje, nie zakłada żadnego rodzaju blokad itp. Więcej o tym jak jest zrealizowana powiemy w dalszych rozdziałach.

Jak wspomnieliśmy wcześniej, API umożliwia wykonanie pewnych czynności administracyjnych. Przyjrzyjmy się teraz przykładowi poświęconemu tej części (listing 2.4).

```
1 using namespace BigTable;
2 Access access("192.168.0.1", 6665);
3 access.createTable("ExampleTable", "ExampleTableType");
4
5 TableAdmin tableAdmin = access.adminTable("ExampleTable");
6
7 tableAdmin.truncate();
8 tableAdmin.drop();
```

Listing 2.4: Przykładowe użycie API do czynności administracyjnych

Klasa `Access` umożliwia stworzenie tabeli, poprzez wywołanie metody `createTable` i podanie jako parametrów nazwy: tabeli i jej typu. W celu wykonania operacji administracyjnych na tabeli należy pobrać obiekt typu `TableAdmin`, który umożliwia nam usunięcie wszystkich rekordów (operacja `truncate`) bądź całej tabeli (operacja `drop`).

2.4. Inne wykorzystane projekty

W poprzednich rozdziałach przedstawiliśmy GBT od strony użytkownika, teraz przyjrzymy się w jaki sposób ta aplikacja jest zrealizowana.

GBT wykorzystuje inne istniejące i rozwijane przez firmę Gemius technologie. Są to: format plików **CommonLib** oraz rozproszony system plików **Moose File System** (MooseFS, MFS).

2.4.1. Format CommonLib

Dane są przechowywane w plikach o formacie **CommonLib**. Można o nich myśleć jak o odpowiedniku Google SSTable. Pliki te są tylko do odczytu, silnie skompresowane i udostępniają wydajną metodę odczytu danego ciągłego podzbioru rekordów (dzięki czemu GBT również charakteryzują te dwie ostatnie cechy). Plik ten przechowuje rekordy posortowane wg klucza, zgrupowane w **bloki** stałego, konfigurowalnego rozmiaru (standardowo 64 KB). Każdy plik wyposażony jest także w **indeks**, zawierający informacje o liczbie bloków i skrajnych wartościach kluczy dla każdego bloku. Odczyt danych z pliku tego formatu realizowany jest w następujący sposób: najpierw wykonywane jest wyszukiwanie binarne na indeksie w celu zlokalizowania bloku zawierającego rekord o danym kluczu. Następnie blok ładowany jest do pamięci, odpowiedni rekord może być wtedy odczytany.

Dane z kolejnych transakcji przechowywane są w osobnych plikach. Każdy z rekordów wyposażony jest w dodatkową kolumnę (pole) zawierającą informację czy dany rekord w tej transakcji jest wstawiany do tabeli (operacja `INSERT`), usuwany (`DELETE`) czy też jego wartość jest aktualizowana (`UPDATE`). Informacja ta jest potrzebna przy pobieraniu rekordów: gdy chcemy pobrać rekord o danym kluczu k, to przeglądamy pliki w kolejności od tych przechowujących najbardziej aktualne dane (takie uszeregowanie plików jest możliwe, więcej powiemy o tym później), poszukując zadanej wartości klucza. Gdy natrafimy na nią i rekord

ten pochodzi z operacji `DELETE`, to informujemy o braku rekordu o zadanym kluczu, wpp. przekazujemy jego wartość.

Oczywiście, jeżeli na tabeli wykonano by wiele transakcji, to poszukiwanie rekordu o danym kluczu wymagałoby przeszukania wielu plików, co znacznie podrożyłoby operację `SELECT`. W związku z tym GBT udostępnia operację `MERGE`, w wyniku której zadany zbiór plików z danymi zostanie scalony w jeden. Operacja ta jest **systemowa**, to znaczy wykonywana przez system, a nie zlecana przez użytkownika. Więcej o tej operacji powiedziane będzie później.

2.4.2. Moose File System

MooseFS jest odpornym na awarie rozproszonym systemem plików. Rozprasza dane pomiędzy kilka fizycznych serwerów, które są widoczne dla użytkownika jako pojedynczy zasób. Jeśli chodzi o standardowe operacje na plikach, MooseFS udostępnia podobne możliwości jak najpopularniejsze systemy plików obsługiwane przez uniksopodobne systemy operacyjne: posiada hierarchiczną strukturę (drzewo katalogów), przechowuje atrybuty pliku (uprawnienia, czasy ostatniego dostępu i modyfikacji); umożliwia tworzenie plików specjalnych (urządzeń blokowych i znakowych, potoków i gniazd), dowiązań symbolicznych (dowiązań do innych plików, niekoniecznie na MooseFS) oraz twardych dowiązań (różnych nazw plików opowiadających tym samym danym fizycznym na MooseFS). Dostęp do plików może być limitowany na podstawie adresu IP klienta i/lub hasła.

Dodatkowymi cechami MooseFS są:

- większa odporność na awarie (dane mogą być przechowywane w kilku kopiach na kilku fizycznych komputerach),
- możliwość dynamicznego zwiększania przestrzeni dyskowej poprzez dołączenie do systemu nowych komputerów/dysków,
- możliwość przechowywania usuniętych plików przez dany okres czasu ("kosz" na poziomie systemu plików),
- możliwość stworzenia **migawki** (ang. *snapshot*) pliku — spójnej kopii pliku, nawet w czasie gdy plik jest właśnie zapisywany/odczytywany.

System MooseFS składa się z dwóch typów węzłów: **serwera głównego** (zarządzającego systemem) oraz **serwerów kawałków** (ang. *chunk servers*, węzłów odpowiedzialnych za przechowywanie i udostępnianie danych).

MooseFS udostępniony jest na licencji GPL. Ze strony [MFS] można pobrać jego kod źródłowy, a także więcej o nim przeczytać.

To właśnie wykorzystanie MooseFS leży u podstaw zapewniania **trwałości** danych przechowywanych przez GBT (ang. *durability*, jeden z warunków ACID). Jak już wspomnieliśmy, dane przechowywane są jako pliki w formacie CommonLib. Pliki te przechowywane są właśnie w MooseFS.

Ponadto wykorzystanie MooseFS umożliwia poszczególnym składnikom systemu dostęp do wspólnej przestrzeni dyskowej. Więcej o tym napisano w kolejnym podrozdziale.

2.5. Składniki systemu

Implementacja GBT składa się z czterech części. Są to:

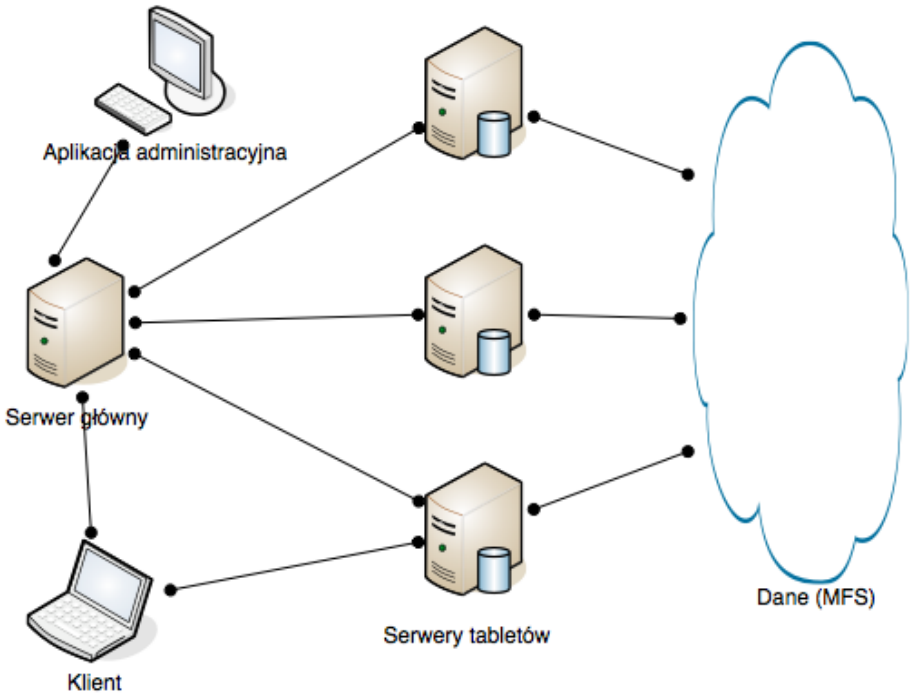
- klient,
- serwer tabletów (TabletServer),
- serwer główny (MasterServer),
- aplikacja administracyjna.

2.5.1. Klient

Klient to biblioteka dołączana do wszystkich programów użytkowników i udostępniająca im API umożliwiające dostęp do systemu. Więcej o API powiedzieliśmy w podrozdziale 2.3.

2.5.2. Serwer tabletów (TabletServer)

TabletServer (serwer tabletów, w skrócie TS) to węzeł zajmujący się obsługą tabletów. W skład jego obowiązków wchodzi m.in. transfer rekordów z obsługiwanego tabletu do/od klienta i wykonywanie wszystkich operacji administracyjnych na tablecie zleconych przez serwer główny (m.in. polecenia scalenia plików z danymi, czy usunięcia niepotrzebnych plików). TS mogą być dynamicznie dodawane (lub usuwane) z systemu, przejmując na siebie część obciążenia.



Rysunek 2.2: Składniki Gemius BigTable

Wszystkie obsługiwane przez TS dane związane z tabletem (rekordy, pliki z metainformacjami) są umieszczane na wspólnej dla wszystkich TS przestrzeni dyskowej z systemem plików MooseFS. Wszystkie TS przechowują pliki w tym samym katalogu. Poniżej przedstawiona jest jego struktura:

- `T_{`*`id_tabeli`*`}` — katalog przechowujący pliki związane z daną tabelą, np. `T_1`;

– `t_{id_tabletu}` — katalog przechowujący pliki związane z danym tabletem. Numer tabletu jest w postaci szesnastkowej i zawsze stałej długości 8 znaków, np. `t_0000001a`:
  * `data-tmp.{id_transakcji}-{numer_części}` — plik przechowujący dane z niezatwierdzonej jeszcze transakcji o numerze `id_transakcji`. W ramach jednej transakcji może być stworzonych wiele takich plików — jeśli np. w czasie transakcji dojdzie do podziału tabletu (więcej o tej operacji w rozdziale 2.7), to pliki te są numerowane chronologicznie (`numer_części`). Po wykonaniu operacji zatwierdzenia nazwy tych plików zostaną zmienione na opisane w kolejnym punkcie:
  * `data.{id_pierwszej}.{id_ostatniej}` (gdzie obydwa parametry są numerami operacji zatwierdzenia) — plik z danymi, zawiera wpisy o wszystkich rekordach dodanych, usuniętych bądź zmodyfikowanych podczas transakcji, które zakończyły się operacją zatwierdzenia o numerach od `id_pierwszej` do `id_ostatniej`. Np. plik `data.3.3` zawiera dane odpowiadające zatwierdzonej transakcji o numerze 3. Jeśli w ramach danej transakcji zostanie stworzonych kilka plików, numer ich części będzie uwzględniony w numerze operacji `COMMIT`, np. plik `data.3-2.3-2` zawiera (drugą) część danych z zatwierdzonej transakcji o numerze 3;
  * `fileinfo.xml` — zwany także **metaplikiem** — plik zawierający informacje o plikach z danymi danego tabletu: nazwę, rozmiar, numer pierwszej i ostatniej operacji `COMMIT` (ew. jej części), z których dane są przechowywane w pliku;
  * pliki tymczasowe — w katalogu odpowiadającym tabletowi tworzone są też pliki tymczasowe przechowujące część przesyłanych właśnie rekordów. Za ich tworzenie i usuwanie odpowiedzialna jest biblioteka CommonLib. Informacje o tych plikach nie są uwzględniane w powyższym metapliku.

2.5.3. Serwer główny (MasterServer)

MasterServer (w skrócie MS) to serce systemu. Węzeł ten zajmuje się przydzielaniem tabletów do serwerów tabletów, reagowaniem na przyłączanie i odłączanie się serwerów tabletów, równoważeniem obciążenia TS, zlecaniem operacji administracyjnych na tabletach, zarządzaniem transakcjami itp. Intuicja o podziale kompetencji MS i TS powinna być taka: wszystkie decyzje zapadają po stronie MS, a fizycznie wykonywane są przez TS.

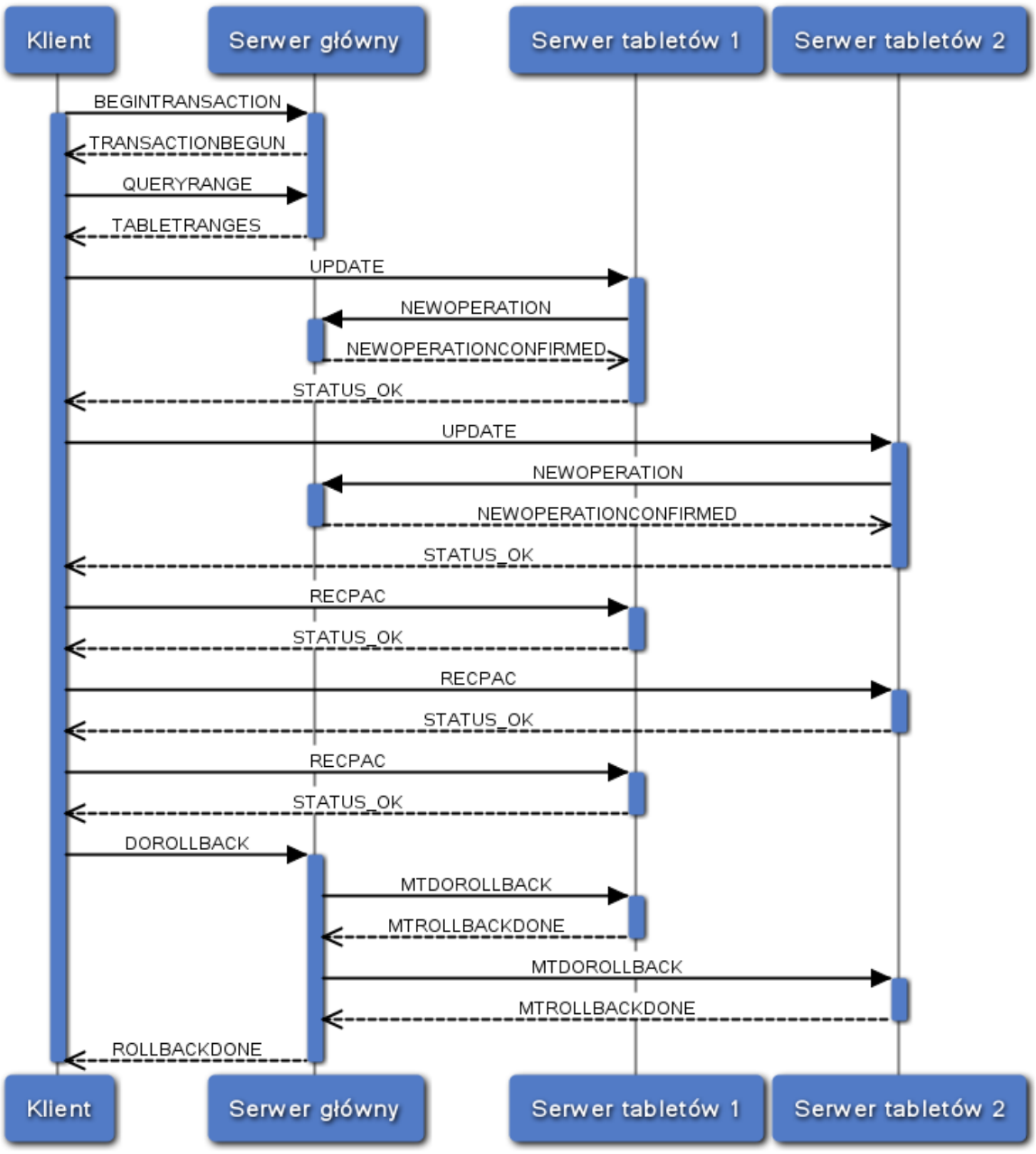
2.5.4. Aplikacja administracyjna

Aplikacja administracyjna to aplikacja umożliwiająca zarządzanie GBT i udostępniającą informacje o systemie. Udostępnia takie operacje jak zatrzymanie systemu, wypisanie tabel istniejących w bazie, wypisanie serwerów tabletów działających w systemie, zmiana polityki scalania plików z danymi lub podziału tabletów itp.

2.6. Ogólny schemat działania systemu

W tym podrozdziale zostanie przedstawiony ogólny schemat działania systemu, w celu wyrobienia odpowiedniej intuicji przed lekturą kolejnych rozdziałów.

Na rysunku 2.3 przedstawiony jest diagram przepływu obrazujący komunikację klienta z systemem składającym się z trzech węzłów (serwera głównego i dwóch serwerów tabletów).



Rysunek 2.3: Diagram przepływu ilustrujący rozpoczęcie nowej transakcji, dodanie rekordów, a następnie wycofanie transakcji

Klient rozpoczyna transakcję, dodaje dane, a później wycofuje rozpoczętą transakcję. Wycofanie (operacja `ROLLBACK`) zostało użyte w tym przykładzie ponieważ jest operacją mniej skomplikowaną niż operacja zatwierdzenia transakcji (`COMMIT`). Więcej o tym jak realizowana jest operacja zatwierdzania powiemy w rozdziale dotyczącym zapewniania niezawodności.

Wymienione operacje mogą mieć w rzeczywistości bardziej skomplikowany przebieg, np. niektóre z jej faz mogą zakończyć się błędem, wtedy system będzie musiał podjąć odpowiednie kroki. Na schemacie przedstawiono jednak najprostszy z przypadków.

Zakladamy, że klient zna już identyfikator (ID) tabeli, do której chce dodać rekordy. Aby rozpocząć transakcję, klient wysyła do serwera głównego wiadomość `BEGINTRANSACTION`, w odpowiedzi dostając `TRANSACTIONBEGUN`, wraz z numerem (ID) nowej transakcji. Przed dodaniem rekordów do tabeli musi dowiedzieć się z jakich tabletów składa się tabela, jakie są skrajne wartości kluczy każdego z nich oraz który TS obsługuje które tablety. W tym celu wysyła do M komunikat `QUERYRANGE` jako parametr podając ID tabeli, a otrzymuje odpowiedź `TABLETRANGES` — listę krotek *(ID tabletu, lewy klucz, prawy klucz, adres IP i port TS)*. Po otrzymaniu jej można już zacząć przesyłać dane. Najpierw klient kontaktuje się z wybranym serwerem tabletów i przesyła mu informację o chęci rozpoczęcia transmisji danych do danego tabletu (komunikat `UPDATE`), sparametryzowane ID tabletu i ID transakcji. TS raportuje udział danego tabletu w transakcji (komunikat `NEWOPERATION`, również sparametryzowany ID tabletu i ID transakcji) w odpowiedzi otrzymując komunikat `NEWOPERATIONCONFIRMED`. Po tym potwierdza klientowi możliwość rozpoczęcia transmisji danych komunikatem `STATUS_OK`.

Po tym ma miejsce transmisja danych. Odbywa się ona bezpośrednio między klientem a TS. Klient przesyła zbiór rekordów w komunikacie `RECPAC`, odbiór każdego jest potwierdzany przez TS.

Gdy transmisja danych zostanie zakończona, klient może zdecydować się na wycofanie transakcji. Przesyła w tym celu do serwera głównego komunikat `DOROLLBACK`, sparametryzowany numerem transakcji. Dla każdego tabletu biorącego udział w transakcji, MS przesyła obsługującemu go TS komunikat `MTDOROLLBACK`. Po zebraniu kompletu odpowiedzi `MTROLLBACKDONE`, potwierdza klientowi zakończenie operacji (komunikat `ROLLBACKDONE`).

Zauważmy na tym przykładzie, że:

- Transmisja rekordów (generująca zdecydowaną większość ruchu sieciowego przez GBT) odbywa się bezpośrednio pomiędzy klientem a serwerami tabletów. Serwer główny, jedyny scentralizowany element systemu, nie jest więc w tym przypadku wąskim gardłem.

- MS zajmuje się jedynie koordynacją zachodzących w GBT operacji (jak np. operacji rozpoczęcia czy wycofania transakcji). To proste i szybkie czynności, nie generujące dużego obciążenia tego węzła.

- Oczywiście transmisja danych do różnych TS odbywa się współbieżnie. Na wykresie nie jest to wprost zaznaczone.

2.7. Operacje udostępniane przez Gemius BigTable

Operacje udostępniane przez GBT możemy podzielić na 3 kategorie:

- **Udostępnione klientowi** — czyli te bezpośrednio wynikające z API udostępnionego klientowi. Można o nich myśleć jak o przypadkach użycia klienta oraz operacjach pomocniczych, bez których przypadków użycia nie można by było zrealizować (jak na przykład operacja `QUERYRANGE`).

- **Administracyjne** — czyli te, których wykonanie można zlecić za pomocą aplikacji administracyjnej. Można o nich myśleć jak o przypadkach użycia administratora bazy.

- **Systemowe** — umożliwiające poprawne i optymalne działanie systemu, ale nie są niczyimi przypadkami użycia. Przykładem jest na przykład przydzielenie tabletu do danego TS, czy scalenie kilku plików z danymi.

W tym podrozdziale powiemy więcej o każdej z operacji, jednocześnie wyjaśniając jak jest ona realizowana, na poziomie komunikacji między MS, TS a klientem/aplikacją administracyjną. W związku z tym dość szczegółowo zostanie opisana zasada działania GBT.

Nazwy operacji zazwyczaj pochodzą od nazw komunikatów je zlecających.

W opisach realizacji operacji uwzględniono jedynie ich najprostsze przebiegi. Oczywiście w trakcie może dojść do awarii, na co system powinien zareagować; zmiany w przebiegach mogą też zależeć od tego w jaki sposób współbieżnie wykonują się niektóre operacje. Więcej zarówno o reagowaniu na awarie, jak i o synchronizacji operacji znajduje się w rozdziale dotyczącym zapewniania niezawodności.

2.7.1. Operacje udostępnione klientowi

GETTABLEID

Użytkownik systemu posługuje się **nazwami** tabel, natomiast system w wielu miejscach (np. jako parametry komunikatów) używa jedynie ich **identyfikatora** (nieujemnej liczby całkowitej). Jeżeli użytkownik wykorzystując API zechce wykonać operację na tabeli o nazwie dotychczas nieznanej aplikacji klienckiej, to wysyła komunikat `GETTABLEID` sparametryzowany nazwą tabeli, w odpowiedzi otrzymuje `TABLEID`.

QUERYRANGE

Operacja `QUERYRANGE` umożliwia klientowi pobranie informacji o tabletach danej tabeli. Klient przesyła komunikat `QUERYRANGE`, w odpowiedzi otrzymuje `TABLETRANGES` wraz z listą krotek *(ID tabletu, lewy klucz, prawy klucz, adres IP i port TS)*.

CREATETABLE

Operacja, w wyniku której tworzona jest nowa tabela oraz jeden tablet obsługujący całą przestrzeń kluczy tej tabeli. Klient wysyła MS komunikat `CREATETABLE` sparametryzowany nazwą tabeli i nazwą typu tabeli (więcej o typach tabel później), odsyłana jest mu wiadomość `TABLECREATED`.

TRUNCATETABLE

Operacja usunięcia wszystkich danych z tabeli. Realizowana jest w następujący sposób: klient przesyła MS komunikat `TRUNCATETABLE`. MS dla każdego tabletu należącego do tej tabeli wysyła obsługującemu go TS komunikat `DROPTABLET`. Po zebraniu kompletu odpowiedzi `TABLETDROPPED`, tworzy nowy tablet w tej tabeli, obsługujący całą przestrzeń kluczy. Klientowi odsyłany jest komunikat `TABLETRUNCATED`.

DROPTABLE

Operacja usunięcia tabeli. Klient przesyła MS komunikat `DROPTABLE`, następnie MS zleca usunięcie wszystkich tabletów podobnie jak w przypadku operacji `TRUNCATETABLE`. Klientowi odsyłana jest wiadomość `TABLEDROPPED`.

BEGINTRANSACTION

Rozpoczęcie transakcji. Klient przysyła komunikat `BEGINTRANSACTION`. MS generuje nowy ID transakcji i odsyła go klientowi w komunikacie `TRANSACTIONBEGUN`.

DOCOMMIT

Zatwierdzenie transakcji. Operacja ta ściśle wiąże się z zapewnianiem niezawodności, więc zostanie szczegółowo opisana później. Na razie warto wspomnieć, że operacje `COMMIT` są numerowane **chronologicznie** nieujemną liczbą całkowitą (ID), tj. ta o mniejszym numerze miała miejsce wcześniej niż ta o większym.

DOROLLBACK

Odrzucenie transakcji. Klient przesyła komunikat `DOROLLBACK`, następnie MS dla każdego tabletu biorącego udział w transakcji przesyła opiekującemu się nim TS komunikat `MTDOROLLBACK`. Po zebraniu odpowiedzi `MTROLLBACKDONE` M przesyła klientowi potwierdzenie `ROLLBACKDONE`.

DISCONNECT

Rozłączenie klienta z serwerem głównym.

SELECT

Operacja umożliwiająca pobranie rekordów z bazy. Klient wymaga połączenia z MS jedynie w celu wykonania operacji `QUERYRANGE` (oraz ew. rozpoczęcia transakcji). Po otrzymaniu informacji które rekordy znajdują się na których TS, łączy się z każdym TS serwującym interesujący go przedział i przesyła komunikat `SELECT`. Następnie TS transmituje klientowi rekordy w niewielkich paczkach, w komunikatach `RECPAC`. Po przesłaniu wszystkich rekordów, wysyła wiadomość `SELECTEND`.

2.7.2. Operacje systemowe

TAKETABLET

Operacja zlecająca TS przejęcie opieki nad danym tabletem. Wywoływana jest w trzech przypadkach:

- gdy tworzony jest nowy tablet, nie posiadający żadnych danych i obsługujący całą przestrzeń kluczy (w wyniku stworzenia nowej tabeli lub wykonania na tabeli operacji `TRUNCATE`),
- gdy jeden z tabletów został podzielony (o operacji dzielenia tabletu będzie mowa dalej) — w wyniku tego należy przejąć nowo powstały tablet,

- gdy danemu tabletowi należy zmienić obsługujący go TS — w celu zrównoważenia obciążenia, bądź z powodu awarii jednego z TS.

MS przesyła wybranemu TS komunikat `TAKETABLET`, w odpowiedzi otrzymuje `TABLETTAKEN`. Komunikat przesyłany jest z flagą `create` informującą o tym czy TS ma stworzyć we wspólnej przestrzeni dyskowej katalog i pliki odpowiadające temu tabletowi. Flaga jest podniesiona jedynie w pierwszym z wymienionych wcześniej przypadków — w pozostałych odpowiednie pliki i katalog już istnieją.

DIVIDETABLET

Operacja podziału tabletu. Wykonywana jest zarówno w celu zrównoważenia obciążenia, jak i rozdzielenia danych pomiędzy TS.

Interesującym kierunkiem badań może być **polityka** podziału tabletu, tj. scenariusz kiedy podejmować decyzję o tym jaki tablet podzielić (i wg jakiego klucza), aby system działał jak najsprawniej. GBT jest zaprojektowany w taki sposób, by łatwo można było modyfikować dotychczasowe i dodawać nowe polityki — poprzez stworzenie klas implementujących dany interfejs. W systemie można używać wszystkich stworzonych polityk, indywidualnie przypisując każdej tabeli politykę, z której ma korzystać. Politykę można także zmieniać dynamicznie, w trakcie działania systemu.

Polityki mogą opierać się na rozmaitych informacjach: rozmiarze danych serwowanych przez tablet, liczbie odwołań do nich, liczbie transakcji, w których tablet bierze/brał udział itp.

W celu podziału tabletu MS przesyła do TS komunikat `DIVIDETABLET` zawierający m.in. ID dzielonego i nowo stworzonego tabletu i klucz, wg którego należy podzielić tablet. Załóżmy, że w systemie był obecny tablet o ID t_1, serwujący rekordy o kluczach z przedziału $[k_1, k_2)$. Zlecono podział tego tabletu wg klucza k_0 i stworzenie nowego tabletu t_2. Od teraz tablet t_1 serwuje dane z przedziału $[k_1, k_0)$, a tablet t_2 dane z przedziału $[k_0, k_2)$. W odpowiedzi TS przesyła MS komunikat `TABLETDIVIDED`. Następnie M przydziela nowy tablet któremuś z TS.

Wykonanie operacji podziału tabletu ma być z założenia szybkie po stronie TS. W związku z tym nie są w jej trakcie kopiowane żadne rekordy. TS tworzy katalog dla nowego tabletu i tworzy w nim **twarde** dowiązania do wszystkich plików z danymi obsługiwanymi przez dzielony tablet. Oczywiście w związku z tym zarówno dzielony, jak i nowo powstały tablet zawierają pliki z rekordami nieobsługiwanymi przez nie. Informacje o tych rekordach w tych plikach zostaną jednak usunięte przy najbliższej operacji scalania tych plików.

Dzięki temu, że katalogi i pliki tabletu tworzone są we wspólnej dla wszystkich TS przestrzeni dyskowej opartej na MooseFS, nowo powstały tablet może być przejęty przez inny TS niż ten, który obsługuje dzielony tablet. Nie ma potrzeby transferu jakichkolwiek rekordów między TS.

W przyszłości planowane jest udostępnienie także dualnej operacji, operacji **scalania** tabletów.

MERGE

Operacja scalania plików z danymi. Jak powiedzieliśmy wcześniej, dane z poszczególnych transakcji przechowywane są w osobnych plikach. Gdyby kontynuować to bezrefleksyjnie, operacja `SELECT` stałaby się dużo droższa (wymagałaby otwarcia i przeglądania wielu plików), a także dane zajmowałyby znacznie większą niż optymalna ilość miejsca (np. rekord który został dodany, usunięty, a później znowu dodany, byłby przechowywany trzykrotnie). W tym celu wprowadzono operację `MERGE`.

Podobnie jak w przypadku operacji podziału tabletu, tak i tu ciekawym zagadnieniem jest polityka scalania, tj. podejmowanie decyzji o tym kiedy i jakie pliki scalać. Strategia implementacyjna jest w tym przypadku podobna; w systemie na raz można korzystać z wielu polityk. Każdy tablet ma przypisaną indywidualnie politykę scalania.

W celu zlecenia TS scalenia plików, MS wysyła komunikat `DOMERGE`, jako parametry podając m.in. listę plików do scalenia. Scalać można jedynie **ciągłe** podzbiory plików, tj. zawierające dane odpowiadające operacjom zatwierdzania z ciągłego przedziału czasu. Oczywiście raz scalone pliki można później znowu scalać.

Pliki, które były używane do stworzenia scalonego pliku wynikowego **nie** są usuwane zaraz po zakończeniu operacji scalania. Są potrzebne by zapewnić odpowiednią synchronizację transakcji. Więcej o tym będzie mowa w rozdziale dotyczącym zapewniania niezawodności.

Po zakończeniu operacji scalania, TS odsyła MS komunikat `MERGECOMPLETED`.

REMOVEOUTDATEDFILES

Operacja usunięcia przestarzałych plików. Przed chwilą powiedzieliśmy, że po wykonaniu operacji scalenia pliki używane do stworzenia scalonego pliku wynikowego nie są usuwane. W końcu jednak może nadejść moment gdy pliki te będą bezużyteczne, wtedy można zlecić usunięcie ich. W tym celu M przesyła do TS komunikat `REMOVEOUTDATEDFILES`, w odpowiedzi otrzymując `OUTDATEDFILESREMOVED`.

2.7.3. Operacje administracyjne

ADDTABLETYPE

GBT jest przystosowany do obsługi wielu tabel różnych typów, umożliwiając tworzenie nowych typów w czasie działania systemu. Nie można więc obsługiwanych typów wkompilować statycznie, potrzebne jest rozwiązanie dynamiczne.

Obecnie dodawanie nowych typów jest dla użytkownika nieco skomplikowane i w przyszłości zostanie uproszczone. Oparte będzie jednak na podobnej idei.

Aby dodać nowy typ, trzeba najpierw napisać klasę implementującą odpowiedni interfejs, następnie skompilować ją do postaci biblioteki dynamicznej. Tak powstały plik należy umieścić w odpowiednim katalogu, zarówno po stronie MS, jak i wszystkich TS (w przypadku TS katalog znajduje się we wspólnej przestrzeni dyskowej MFS), w podkatalogu o nazwie równej ID typu. Następnie należy wysłać MS komunikat `ADDTABLETYPE` sparametryzowany nazwą typu i jego identyfikatorem. Od tej pory można już używać tego typu przy tworzeniu nowych tabel.

LISTTABLES

W wyniku tej operacji MS przesyła komunikat zawierający listę wszystkich tabel stworzonych w systemie.

LISTTABLETSERVERS

W wyniku tej operacji MS przesyła komunikat zawierający listę wszystkich serwerów tabletów działających w ramach systemu.

CHANGETABLETMODIFIER

Umożliwia, dla danej tabeli, zmianę jej polityki podziału tabletu.

CHANGEMERGEPOLICY

Umożliwia, dla danego tabletu, zmianę jego polityki scalania plików.

Rozdział 3

Zapewnianie niezawodności

W tym rozdziale opiszemy na czym polega zapewnianie niezawodności w systemie rozproszonym (podrozdział 3.1) oraz przyjrzymy się w jaki sposób jest to zrealizowane w Gemius BigTable.

Całość opisanych tu prac koncepcyjnych i implementacyjnych jest mojego autorstwa.

3.1. Obszary niezawodności

Niezawodność w rozproszonych systemach komputerowych to szerokie pojęcie, na które składają się między innymi (za [Birm05]):

- **Odporność na awarie (ang. *fault tolerance*):** Zdolność systemu do samodzielnego naprawienia się po awarii jednego lub kilku jego komponentów, podtrzymując poprawność jego działania.
- **Wysoka dostępność (ang. *high availability*):** W kontekście systemu odpornego na awarie jest to jego zdolność do poprawnego odtworzenia operacji, pozwalającego na wznowienie dostarczania przez niego usług.
- **Przywracanie sprawności (ang. *recoverability*):** Także w kontekście systemów odpornych na awarie, jest to zdolność uszkodzonych komponentów do zrestartowania się i ponownego przyłączenia się do systemu po tym jak awaria zostanie naprawiona.
- **Spójność (ang. *consistency*):** Zdolność systemu do koordynowania działań wielu komponentów, w kontekście poprawnej współbieżności i odporności na awarie. Dla systemów rozproszonych można to rozumieć jako umiejętność symulowania pracy systemu nierozproszonego.
- **Skalowalność (ang. *scalability*):** Umiejętność systemu do kontynuowania poprawnego działania nawet w przypadku, gdy niektóre z jego elementów osiągają duże rozmiary. Dla przykładu, gdy zwiększa się liczba węzłów w systemie, zwiększa się również liczba połączeń między nimi, a co za tym idzie — częstotliwość awarii sieci. System skalowalny musi umieć sobie poradzić z takimi sytuacjami. Zwiększać możemy m.in. liczbę serwerów, użytkowników, obciążenie — skalowalność należy więc rozpatrywać wielowymiarowo. Specyfikacja systemu powinna definiować jak zmienia się jego wydajność pod wpływem skalowania go w każdym z wymiarów.

Wiele z wymienionych pojęć ma bezpośredni związek z *awariami*. Te z kolei również mogą mieć wiele znaczeń (również za [Birm05]):

- **Awarie kontrolowane (ang. *fail-stop failures*):** sytuacja, gdy komponent sam wykrywa u siebie awarię, informuje o tym pozostałe części systemu i kontynuuje swoją pracę według awaryjnego scenariusza (który, być może, polega na zakończeniu się). Awaria tego typu nie narusza spójności systemu — każdy z komponentów dalej kontynuuje swoją pracę według wcześniej założonego scenariusza.
- **Awarie nagłe (ang. *halting failures*):** sytuacja, gdy praca komponentu zostaje przerwana w sposób nagły, uniemożliwiając mu eleganckie zakończenie aktualnie wykonywanych operacji, zwolnienie pobranych zasobów itp. W tym przypadku może dość do sytuacji, że stan systemu będzie niespójny — rolą systemu odpornego na awarie będzie tę spójność przywrócić. Zauważmy, że w przeciwieństwie do poprzedniego przypadku, tutaj wyzwaniem jest też wykrycie awarii — zazwyczaj nie jest to możliwe w inny sposób niż poprzez zaobserwowanie, że dany komponent przestaje odpowiadać (timeout).
- **Awarie sieci (ang. *network failures*):** sytuacja, gdy komponenty działają w sposób prawidłowy, ale ulega awarii połączenie między nimi. W takim przypadku komponenty bez możliwości komunikacji między sobą muszą skoordynować działania mające na celu przywrócenie sprawności systemowi.

W kolejnych podrozdziałach zajmiemy się omówieniem mechanizmów zapewniających poszczególne obszary niezawodności.

3.2. Spójność — poprawna synchronizacja operacji

Operacje udostępniane przez GBT zostały szerzej opisane w podrozdziale 2.7. Jak wiadomo, nie wszystkie z nich mogą wykonywać się współbieżnie bez żadnych ograniczeń, np. kiedy jeden użytkownik zleci usunięcie danych w tabeli (operacja `TRUNCATETABLE`), a inny pobranie informacji o tabletach danej tabeli (operacja `QUERYRANGE`), to "zły" przeplot tych operacji mógłby doprowadzić do przesłania niespójnej informacji. Dlatego potrzebny jest pewien system zapewniający odpowiednią synchronizację operacji.

Każda operacja, która wymaga odebrania więcej niż 1 komunikatu i/lub zsynchronizowania jej z innymi, po stronie serwera głównego obsługiwana jest przez **koordynatora**. Koordynator implementuje protokół obsługujący daną operację (deterministyczny automat skończony) oraz część zapewniającą synchronizację.

Podczas projektowania systemu przeanalizowaliśmy dwie metody synchronizacji operacji. Opiszemy je dalej, stosując jednak pewne uproszczenia. Po pierwsze, nie będziemy rozpatrywali obsługi transakcji (o tym będzie mowa w rozdziale 3.3). Po drugie, założymy, że w czasie pracy systemu nie dojdzie do awarii (więcej o obsłudze awarii w rozdziale 3.4).

3.2.1. Czytelnicy i pisarze

Sytuacja koordynatorów w systemie jest analogiczna do problemu czytelników i pisarzy. Przypomnijmy — jest to problem synchronizacji dostępu do jednego zasobu dwóch rodzajów procesów — dokonujących i niedokonujących w nim zmian.

W naszym przypadku zasobem jest **zbiór tabletów**. Możemy koordynatorów podzielić na dwie grupy: modyfikujących zasób i niemodyfikujących go. Koordynatorami modyfikującymi zasób (pisarzami) są `CREATETABLE`, `TRUNCATETABLE`, `DROPTABLE`, `MODIFYTABLETS` oraz `TAKETABLET`, pozostali są czytelnikami. Zauważmy, że `TAKETABLET` nie modyfikuje zbioru tabletów w ścisłym znaczeniu, ale do czasu jego zakończenia nie wszystkie tablety z tego zbioru są obsługiwane, więc naturalne jest, że powinna być to operacja blokująca.

Rozwiązanie to jest poprawne i proste w implementacji, posiada jednak poważne ograniczenia. Pierwsze związane z wydajnością — dopuszczenie do działania w systemie na raz tylko jednego koordynatora będącego pisarzem to poważne "wąskie gardło" systemu mającego obsługiwać wiele tabel i transakcji jednocześnie. Warto więc pojęcie zasobu rozdrobnić.

Po drugie, synchronizacja operacji to nie tylko kwestia dostępu do tabletów. Na przykład wymogiem systemu jest to, by operacje zatwierdzenia transakcji (`COMMIT`) były wykonywane po kolei, tzn. ta o niższym numerze `CommitID` była zatwierdzona wcześniej od tej o wyższym numerze. Ciężko w naturalny sposób sformułować ten wymóg w języku czytelników i pisarzy.

3.2.2. Zakładanie blokad

Procesy działające w systemie operacyjnym również wymagają dostępu do pewnych zasobów (np. plików), modyfikując je lub nie. Jednak procesu nie nazywa się odgórnie czytelnikiem lub pisarzem, pozwala mu się samemu zdecydować czy dany plik chce otworzyć z prawami do odczytu czy zapisu. Względem jednego z zasobów (plików) może być on więc pisarzem, a względem innego czytelnikiem.

Podobny model zastosowaliśmy w systemie. Podstawową jednostką będącą zasobem jest **tablet**, a koordynatorzy zakładają na nim blokadę zapisu/odczytu w zależności od tego, czy chcą dokonać na nim modyfikacji czy nie. Przyjrzymy się bliżej temu rozwiązaniu.

W systemie obecna jest klasa `ResourceManager` (zarządca zasobów, w skrócie RM) udostępniająca koordynatorom metody `readLock`, `writeLock`. Na początku swojej pracy koordynator wywołuje jedną z nich, jako parametr podając listę tabletów, na które ma być założona blokada. Blokada zakładana jest w sposób atomowy — albo na wszystkie żądane tablety (a jeśli w czasie od złożenia żądania założenia blokady któreś z nich zostaną podzielone — także na nowo powstałe w wyniku tej operacji), albo na żadne. Po przydzieleniu blokady koordynator może kontynuować pracę.

Wspomniany w poprzednim podrozdziale wymóg synchronizacji operacji zatwierdzenia transakcji również da się wyrazić w języku rywalizacji o zasoby — RM posiada jeden "bilet" (ang. *ticket*) na wykonanie operacji zatwierdzania, który po kolei przydzielany jest koordynatorom operacji zatwierdzania. Więcej o tym jak realizowana jest ta operacja napisane jest w rozdziale 3.3.1.

```
1  class ResourceManager {
2  public:
3      void readLock(Coordinator *, std::list<BTTabletID>);
4      void writeLock(Coordinator *, std::list<BTTabletID>);
5      void unlock(Coordinator *);
6
7      void suspendLock(Coordinator *, std::list<BTTabletID>);
8
9      void getCommitMutex(CommitCoordinator *);
10     void releaseCommitMutex(CommitCoordinator *);
11 }
```

Listing 3.1: Interfejs klasy `ResourceManager`

Jak widać, zarządca zasobów oprócz założenia blokad do czytania/pisania umożliwia też założenie blokady zawieszającej (ang. *suspend lock*). Więcej o tej blokadzie powiemy w rozdziale 3.4.2.

Metoda ta jest dużo wydajniejsza od wcześniej zaproponowanej — wyłączny dostęp jest przyznawany koordynatorom jedynie do tych tabletów, które rzeczywiście są im potrzebne. I tak np. operacja podziału tabletu teraz wymaga założenia blokady do zapisu jedynie na jeden tablet.

3.3. Poprawna obsługa transakcji

Operacją elementarną w bazie danych nazwiemy najmniejszą pojedynczą operację jaką może wykonać w niej użytkownik — czyli dodanie, aktualizacja wartości, bądź usunięcie rekordu.

Transakcją nazywamy najmniejszą logiczną operację na bazie danych. Dla przykładu, realizacja przelewu miedzy dwoma kontami bankowymi (zmniejszenie salda konta źródłowego oraz zwiększenie salda konta docelowego) jest transakcją składającą się z kilku operacji elementarnych.

ACID (akronim słów *atomicity, consistency, isolation, durability* — atomowość, spójność, izolacja oraz trwałość) to zbiór zasad gwarantujących przetwarzanie transakcji w sposób niezawodny. Zasady te zostały sformułowane przez J. Graya pod koniec lat 70-tych, a nazwa pochodzi od A. Reutera i T. Haerdera (1983).

W tym rozdziale omówimy sposoby realizacji tych zasad w GBT. Zanim jednak do tego przejdziemy, sformułujemy kilka niezbędnych definicji. Każdej transakcji w systemie przypisany jest **identyfikator transakcji** (ang. *transaction id*) — nieujemna liczba całkowita, jednoznacznie identyfikująca transakcję. Przypomnijmy, że transakcja może albo zostać **wycofana** (ang. *rollback*) — wtedy wszystkie wprowadzone przez nią zmiany są anulowane, albo **zatwierdzona** (ang. *commit*) — wtedy przypisywany jest jej unikatowy **identyfikator zatwierdzenia** (ang. *commit id*), jednoznacznie identyfikujący zatwierdzoną transakcję. Identyfikator zatwierdzenia również jest nieujemną liczbą całkowitą, charakteryzującą się tym, że transakcje zatwierdzone wcześniej mają identyfikator o mniejszej wartości niż te zatwierdzone później (w sensie czasu rzeczywistego; jak za chwilę się przekonamy, jednocześnie nie może być zatwierdzanych wiele transakcji na raz).

Często będziemy używać sformułowania **identyfikator transakcji** (bądź, zamiennie, **numer transakcji**) w kontekście transakcji już zatwierdzonych — wtedy oznaczać to będzie de facto jej numer zatwierdzenia. Z kontekstu zawsze wynikać będzie, o który z identyfikatorów nam chodzi. Samo wprowadzenie dwóch identyfikatorów ma na celu jedynie prowadzenie osobnej numeracji transakcji już zatwierdzonych i właśnie trwających.

3.3.1. Atomowość

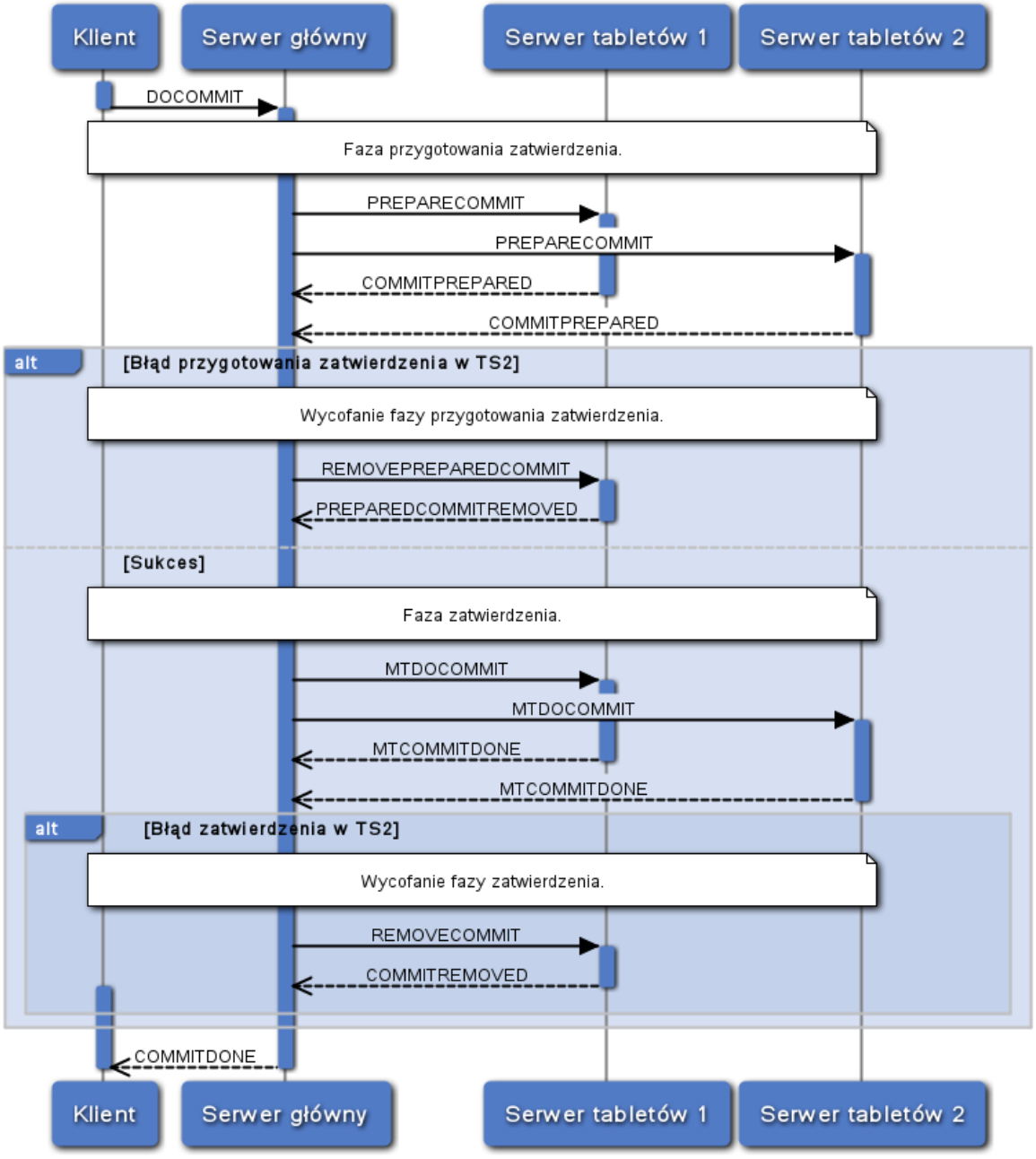
Atomowość to zdolność systemu do zapewnienia, że albo *wszystkie* operacje elementarne związane z transakcją zostaną wykonane, albo *żadna* z nich.

W celu zapewnienia atomowości transakcji należy prześledzić przebieg operacji wycofania (`ROLLBACK`) i zatwierdzenia transakcji (`COMMIT`). Wycofanie powinno anulować wszystkie modyfikacje wykonane przez transakcję, natomiast zatwierdzenie powinno albo zaakceptować, albo odrzucić wszystkie.

Zatwierdzenie transakcji jest realizowane za pomocą algorytmu wzorowanego na *Two-phase commit protocol* (2PC). Jest to algorytm gwarantujący atomowość operacji zatwierdzania w środowisku rozproszonym.

Algorytm ten składa się z dwóch faz. Podczas pierwszej z nich, **fazy przygotowania zatwierdzenia** (ang. *commit-request phase*), serwer główny wysyła do serwerów tabletów żądanie przygotowania zatwierdzenia, tj. zapisania wszystkich rekordów przesłanych podczas danej transakcji do tymczasowego pliku. Po zebraniu kompletu potwierdzeń wchodzimy w kolejną fazę, **fazę zatwierdzania** (ang. *commit phase*), podczas której serwer główny przesyła serwerom tabletów żądanie zatwierdzenia transakcji. Obsługa tej fazy jest już po stronie TS szybka, polega na zmianie nazwy plików tymczasowych. Po odebraniu przez MS kompletu potwierdzeń, transakcja zostaje zatwierdzona.

Oczywiście każda z powyższych faz może po stronie TS zakończyć się niepowodzeniem, wtedy operację zatwierdzenia transakcji należy wycofać. Przyjrzymy się teraz bliżej w jaki sposób operacja ta jest realizowana. Należy zaznaczyć, że nie będziemy się w tym rozdziale zajmować sytuacją, kiedy podczas wykonywania operacji dojdzie do awarii któregoś z węzłów — ten obszar zapewniania niezawodności jest omówiony w rozdziale 3.4.



Rysunek 3.1: Diagram przepływu ilustrujący operację zatwierdzenia transakcji

Jak widać na rys. 3.1, klient przesyła zlecenie zatwierdzenia obecnie wykonywanej transakcji poprzez przesłanie wiadomości `DOCOMMIT`. Serwer główny zaczyna realizację od fazy przygotowania zatwierdzenia. Dla każdego tabletu biorącego udział w transakcji przesyła obsługującemu go serwerowi tabletów wiadomość `PREPARECOMMIT`. TS zapisuje wtedy wszystkie

dane przesłane podczas tej transakcji do tymczasowego pliku w MFS. Po zakończeniu tej operacji przesyła wiadomość `COMMITPREPARED` z odpowiednim statusem informującym o powodzeniu, bądź nie, wykonania tej fazy.

MS zbiera potwierdzenia, jeżeli na którymkolwiek tablecie operacja się nie powiodła, to cała transakcja jest wycofywana — dla każdego tabletu wysyłana jest obsługującemu go TS wiadomość `REMOVEPREPAREDCOMMIT`. W jej wyniku TS usuwa plik tymczasowy oraz wszystkie metadane związane z transakcją, przywracając stan sprzed rozpoczęcia transakcji. Po tym odsyła potwierdzenie `PREPAREDCOMMITREMOVED`. Po zebraniu kompletu potwierdzeń MS przesyła klientowi wiadomość `COMMITDONE` z informacją o tym, że operacja zatwierdzenia zakończyła się niepowodzeniem.

Jeśli faza przygotowania zatwierdzenia zakończy się powodzeniem, to system przechodzi do realizacji fazy zatwierdzenia. Koordynator obsługujący operację pobiera wtedy od zarządcy zasobów bilet na wykonanie operacji zatwierdzania. Zarządca dysponuje tylko jednym takim biletem co gwarantuje, że naraz wykonywana będzie co najwyżej jedna faza zatwierdzenia. Po pobraniu biletu generowany jest unikatowy **identyfikator zatwierdzenia** (ang. *commit identifier*), jednoznacznie identyfikujący zatwierdzoną transakcję. Identyfikator ten jest nieujemną liczbą całkowitą większą od każdego dotychczas wygenerowanego identyfikatora zatwierdzenia. Bilet na wykonanie fazy zatwierdzenia zostanie zwolniony dopiero po jej zakończeniu, co gwarantuje, że zatwierdzone transakcje numerowane są chronologicznie — ta zatwierdzona później ma większy identyfikator od tych zatwierdzonych wcześniej.

Po wygenerowaniu identyfikatora zatwierdzenia serwer główny dla każdego tabletu biorącego udział w transakcji przesyła obsługującemu go TS wiadomość `MTDOCOMMIT`, sparametryzowaną m.in. identyfikatorem zatwierdzenia. Po odebraniu jej TS zmienia nazwę pliku tymczasowego powstałego podczas fazy przygotowania zatwierdzenia na zgodną z konwencją nazywania plików z danymi. Odsyła MS potwierdzenie `MTCOMMITDONE`.

Podobnie jak w poprzedniej fazie, MS zbiera potwierdzenia od TS i jeśli któreś z nich nadejdzie z informacją o błędzie, to całą transakcję należy wycofać. Dla każdego tabletu biorącego udział w transakcji MS przesyła obsługującemu go TS wiadomość `REMOVECOMMIT`. TS usuwa wtedy odpowiednie pliki z danymi i wszystkie wpisy dotyczące ich i odpowiadającej im zatwierdzonej transakcji. Serwerowi głównemu odsyłany jest komunikat `COMMITREMOVED`.

Gdy faza zatwierdzenia zakończy się sukcesem, sukcesem zakończy się również cała operacja zatwierdzenia. Klientowi odsyłany jest wtedy komunikat `COMMITDONE` sparametryzowany identyfikatorem zatwierdzenia `commit_id`. Dopiero w tym momencie transakcję uznaje się za zatwierdzoną. W rozdziale 3.3.3 napiszemy o tym, że w momencie tworzenia nowej transakcji, przypisywana jest jej wartość `snapshot_id` równa numerowi najpóźniej zatwierdzonej transakcji. Oczywiście od tej pory wartość ta przypisywana nowym transakcjom będzie nie mniejsza niż `commit_id`.

Natomiast w przypadku wycofania transakcji sytuacja jest prosta. Do czasu zakończenia wykonywania na obecnej transakcji operacji `COMMIT` żadne dane tej transakcji nie zostaną zwrócone podczas pobierania danych z tabeli. Dzieje się tak, ponieważ operacja `SELECT` pobiera dane jedynie z plików zawierających dane z zatwierdzonych transakcji (patrz rozdział 2.5.2). W związku z tym, w przypadku wycofywania transakcji nie istnieje ryzyko "nie wycofania jej w całości".

3.3.2. Spójność

W standardowych SZBD określa się więzy spójności lub oczekiwania dotyczące związków między elementami danych (np. saldo na koncie nie powinno być ujemne). **Spójność** to cecha określająca gwarancję, że transakcje nie naruszą spójności danych przechowywanych w

bazie.

Jednakże w GBT zrezygnowano z zapewnienia możliwości zdefiniowania więzów spójności — priorytetem projektowym GBT jest szybkość, więc to na użytkowników przeniesiono odpowiedzialność za wykonywanie transakcji nienaruszających wcześniej przyjętych oczekiwań w zakresie spójności danych.

3.3.3. Izolacja

Systemy bazodanowe są przystosowane do współbieżnego przetwarzania wielu transakcji na raz. **Izolacja** transakcji to cecha określająca jak bardzo poszczególne transakcje są od siebie odseparowane.

Jak pośrednio może wynikać z lektury rozdziału 2.3, w odróżnieniu od standardowych systemów bazodanowych, w GBT transakcje składają się albo z samych operacji modyfikujących stan bazy (dodania, usunięcia bądź aktualizacji rekordów) (listing 2.1), albo z samych operacji odczytu danych z bazy (listingi 2.2 i 2.3). Jak widać na ww. listingach, pojęcie *transakcji* jest jawnie wykorzystywane przez klienta jedynie przy dodawaniu danych do bazy; jak się jednak za chwilę przekonamy, odczyty także wykonywane są w ramach transakcji.

Standardowe SZBD wspierają cztery poziomy izolacji: SERIALIZABLE, REPEATABLE READ, READ COMMITED, READ UNCOMMITED, w kolejności od najbardziej do najmniej izolujących. W standardowych SZBD wraz ze wzrostem poziomu izolacji maleje niestety współbieżność bazy — na bazie częściej trzeba zakładać blokady wyłącznego dostępu, np. przy poziomie SERIALIZABLE zakładane są **blokady zakresu** (ang. *range locks*), uniemożliwiające do czasu jej zwolnienia dodania do bazy rekordów o kluczach należących do odpowiadającego blokadzie zakresu. Jednak jak za chwilę się przekonamy, realizacja izolacji transakcji w GBT opiera się na innym pomyśle i nawet zastosowanie SERIALIZABLE nie odbija się negatywnie na współbieżności systemu. W związku z tym transakcje w GBT są izolowane na poziomie SERIALIZABLE. Więcej o poziomach izolacji w standardowych systemach bazodanowych można przeczytać w [Ullm00].

Izolacja transakcji to tak naprawdę zdefiniowanie, jakie dane mogą się pojawiać jako wynik operacji `SELECT`. W związku z tym przyjrzymy się w jaki sposób wykonywana jest operacja odczytu danych.

Każde zapytanie jest wykonywane w ramach transakcji. W momencie tworzenia transakcji, przypisywany jest jej **identyfikator migawki** (ang. *snapshot id*), równy identyfikatorowi ostatniej zakończonej sukcesem operacji `COMMIT`. Wszystkie operacje odczytu wykonane w ramach tej transakcji będą zwracały jako wyniki jedynie dane umieszczone w bazie do czasu zatwierdzenia transakcji o numerze `snapshot_id`. Tak więc do czasu zakończenia tej transakcji GBT musi zapewniać możliwość odczytania jej stanu z chwili gdy zatwierdzana była transakcja o numerze `snapshot_id`.

Pozornie bardzo kosztowne i skomplikowane zadanie okazuje się nie być trudne w obliczu przyjętego projektu systemu. Dane z kolejnych transakcji przechowywane są w osobnych plikach, więc jeśli chcemy pobrać dane jedynie z transakcji o numerach nie większych niż k, to wystarczy pobrać je jedynie z plików zawierających dane z transakcji o numerach nie większych niż k. Dlatego też każda wiadomość `SELECT` sparametryzowana jest numerem `snapshot_id` — oznaczającą największy numer transakcji, z których dane chcemy pobrać.

Jak wspomnieliśmy, co pewien czas wykonywana jest operacja scalania plików z danymi. W wyniku scalenia gubimy jednak informację o stanie bazy z przeszłości — np. ze scalonego pliku zawierającego dane z transakcji 1 i 2 nie będziemy w stanie wyłuskać danych jedynie z transakcji 1. W związku z tym, po scaleniu pliki źródłowe nie są usuwane — ta operacja odroczona jest do czasu aż pliki te "nie będą już potrzebne".

Kiedy potrzeba użycia ww. plików minie? Wyznaczmy minimum z wartości `snapshot_id` z każdej z działających obecnie transakcji. Oznaczmy ją jako `minSnapshot_id`. Oznacza to, że GBT nie potrzebuje już dostępu do bazy sprzed transakcji o numerze `minSnapshot_id`. Można więc już usunąć wszystkie pliki przechowujące dane sprzed transakcji o numerze, które brały już udział w operacji scalania. Takie pliki będziemy nazywać **przestarzałymi** (ang. *outdated*).

Aby zlecić usunięcie przestarzałych plików, dla każdego tabletu posiadającego przestarzałe pliki, MS przesyła obsługującemu go TS komunikat `REMOVEOUTDATEDFILES`, sparametryzowany aktualną wartością `minSnapshot_id`. Po pomyślnym wykonaniu tej operacji TS odsyła potwierdzenie `OUTDATEDFILESREMOVED`.

Wróćmy jednak do opisu wykonywania operacji odczytu danych. Jak możemy zauważyć w listingach 2.2 oraz 2.3, w celu wykonania operacji `SELECT` najpierw należy *utworzyć* opis zapytania — kreując obiekt `Select<Record>` oraz konfigurując go. Następnie można *wykonać* zapytanie wywołując metodę `execute` — albo z użyciem wcześniej stworzonej migawki bazy danych, albo bez.

Utworzenie migawki bazy danych to po prostu utworzenie transakcji. Wykonanie zapytania na danej migawce to po prostu wykonanie operacji `SELECT` w ramach odpowiadającej jej transakcji. Natomiast wykonanie zapytania na aktualnej wersji bazy polega na stworzeniu nowej transakcji, wykonania w ramach niej zapytania, a następnie usunięciu tej transakcji.

3.3.4. Trwałość

Trwałość to własność systemu bazodanowego gwarantująca, że dane z zatwierdzonych transakcji nie zostaną utracone nawet po awarii systemu.

W przypadku GBT oznacza to, że:

1. Dla każdego tabletu: pliki z danymi tego tabletu oraz informacja o tym jakie pliki wchodzą w skład danego tabletu przechowywane są w sposób trwały.

2. Informacja o tabletach obecnych w systemie przechowywana jest w sposób trwały.

Ad 1. Jak już było powiedziane w rozdziale 2.5.2, wszystkie pliki z danymi (oraz metapliki przechowujące informacje o plikach z danymi) przechowywane są przez serwer tabletów jako zasób w systemie plików MooseFS. MFS gwarantuje odporność na awarie poprzez zastosowanie replikacji zasobów. Każdy z plików przechowywanych przez MFS ma przydzielony **współczynnik replikacji** (ang. *goal*), który oznacza w ilu kopiach na różnych serwerach dany plik ma być przechowywany.

Przyjrzyjmy się temu dokładniej: każdy plik przechowywany przez MFS dzielony jest na **kawałki** (ang. *chunks*) o wielkości do 64MB. Każdy z kawałków jest przechowywany przez kilka (dokładnie tyle ile wynosi współczynnik `goal` dla tego pliku) **serwerów kawałków** (ang. *chunkservers*, CS). W przypadku awarii CS przechowującego kawałek pliku posiadającego co najmniej 2 kopie dane będą nadal dostępne z innego serwera, a po pewnym czasie zostaną po raz kolejny zreplikowane, aby zapewnić wymaganą liczbę kopii. Zauważmy, że gdy liczba CS w systemie jest mniejsza od wartości `goal` niektórych plików, to wymagana liczba kopii nie będzie zapewniona. W tym przypadku należy jak najszybciej podłączyć kolejny serwer do systemu.

Przechowywane kawałki są wersjonowane, więc nie ma ryzyka utraty spójności w przypadku podłączenia się CS z nieaktualną wersją kawałka.

Domyślną wartością `goal` dla MFS wykorzystywanego przez GBT jest 3.

Ad. 2. Informacje o tabletach obecnych w systemie są przechowywane przez serwer główny w sposób trwały. W swoim katalogu roboczym tworzy on identyczną strukturę katalogów jak TS (por. rozdział 2.5.2), jednak w każdym katalogu odpowiadającym tabletowi znajduje się jedynie plik `tabletinfo.xml` zawierający następujące informacje o danym tablecie:

- granice kluczy obsługiwane przez ten tablet,
- polityka scalania plików w tym tablecie.

Natomiast w każdym katalogu odpowiadającym tabeli przechowywany jest plik `table-info.xml` zawierający następujące informacje o tabeli:

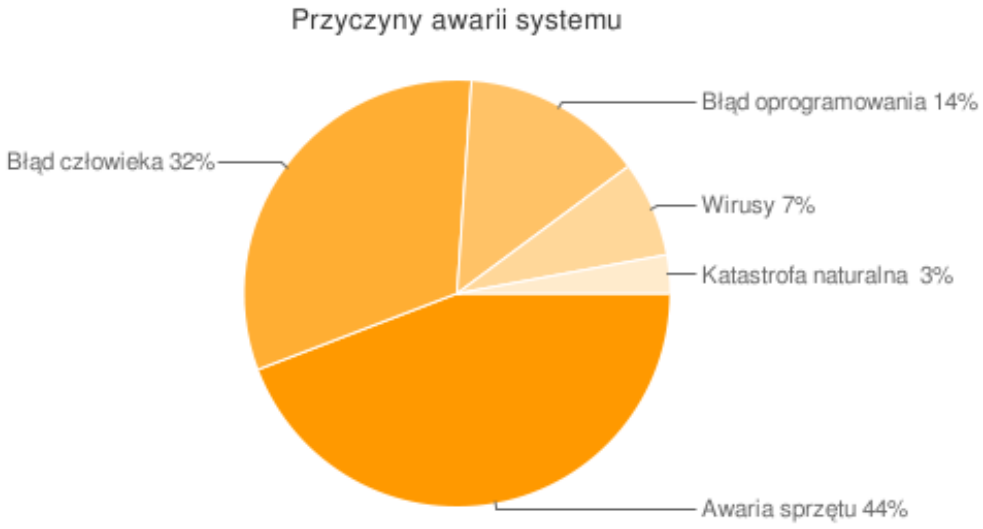
- nazwa tabeli,
- typ tabeli,
- polityka podziału tabletów w tej tabeli.

Należy zaznaczyć, że MS ze względów wydajnościowych nie zapisuje do systemu plików MFS, jednak można w tym przypadku zastosować np. macierz dyskową (RAID), co także zapewni trwałość przechowywanych danych w wyniku awarii.

3.4. Odporność na awarie, wysoka dostępność i przywracanie sprawności

3.4.1. Najważniejsze definicje i postulaty projektowe

Przed rozpatrywaniem odporności na awarie należy zastanowić się czym są one spowodowane.



Rysunek 3.2: Przyczyny awarii systemu wg [Marc00]

Najczęściej przyczyną są awarie sprzętu — 44%, 14% przypadków spowodowanych jest przez błąd oprogramowania, 7% przez wirusy, a 3% przez katastrofy naturalne. Niezależnie od stopnia uodpornienia systemu na czynniki zewnętrzne, i tak aż 32% awarii powodowanych

jest przez błąd człowieka. Nie da się więc (np. poprzez zakup niezawodnego sprzętu) odizolować systemu od czynników powodujących awarie i projektując system niezawodny należy zaopatrzyć go w mechanizmy radzenia sobie z awariami.

W kolejnych podrozdziałach opiszemy scenariusze reagowania na awarie obydwu komponentów systemu: serwera tabletów oraz serwera głównego.

3.4.2. Odporność na awarie serwera tabletów

Podstawową konsekwencją awarii serwera tabletów jest niedostępność serwowanych przez niego tabletów, a co za tym idzie — niedostępność części danych obsługiwanych przez aplikację. Tablety serwowane przez serwer, który uległ awarii, będziemy nazywać **osieroconymi**.

Założeniem projektowym systemu jest obecność wielu serwerów tabletów, więc ich awarie będą się zdarzać stosunkowo często. W związku z tym, oraz z wymogiem wysokiej dostępności, system musi być wyposażony w mechanizmy automatycznego przywracania sprawności i wznowienia udostępnianych usług.

Jak wspomniano wcześniej, system musi być odporny także na awarie nagłe — a co za tym idzie, nie możemy nic zakładać ani o momencie wystąpienia awarii ani o stanie w jakim zostaną pozostawione osierocone tablety. Taka mnogość scenariuszy wymusiła na projekcie komponentu obsługującego awarię jego możliwie dużą ogólność. Posłużono się tu podobnym podejściem jak przy systemach plików z księgowaniem (ang. *journalised file system*).

W systemach plików tego typu dane nie są od razu zapisywane na dysk, tylko zapisywane w swoistym dzienniku/kronice (ang. *journal*). Dzięki takiemu mechanizmowi działania zmniejsza się prawdopodobieństwo utraty danych: jeśli utrata zasilania nastąpiła w trakcie zapisu — zapis zostanie dokończony po przywróceniu zasilania; jeśli przed — stracimy tylko ostatnio naniesione poprawki, a oryginalny plik pozostanie nietknięty ([WikiJFS]).

Podobny pomysł zdecydowano się wykorzystać projektując odporność na awarie TS. Niestety, najprostsza jego realizacja okazuje się być nieefektywna: zapisywanie przez TS w sposób trwały informacji o planowanych działaniach wiązałoby się z potrzebą częstego zapisywania niewielkich porcji danych w MFS, co nie jest efektywne. W związku z tym zdecydowano się na przechowywanie tej informacji po stronie MS. Zostanie to opisane szerzej w kolejnych podrozdziałach.

Opiszemy teraz ogólnie procedurę obsługi awarii serwera tabletów, a w kolejnych podrozdziałach szerzej przyjrzymy się poszczególnym jej krokom. Po wykryciu awarii MS wstrzymuje wszystkie operacje wykonujące się na osieroconych tabletach, następnie przydziela tablety pozostałym TS. Każdy z serwerów przywraca spójność przydzielonych mu tabletów. Następnie serwer główny wznawia wstrzymane operacje. Nie wszystkie będą mogły się zakończyć powodzeniem (w szczególności, jeśli do awarii dojdzie podczas wykonywania operacji COMMIT, to z dużym prawdopodobieństwem utracimy część danych z bieżącej transakcji, a więc operacja zakończy się z błędem), ale system będzie w stanie powrócić do spójnego stanu i kontynuować pracę.

W kolejnych podrozdziałach bliżej opiszemy tą procedurę. Skupimy się na omówieniu awarii nagłej, w wyniku której TS zakończył pracę w dowolnym momencie.

Wykrycie awarii

Wykrycie awarii jest pierwszym, często nieoczywistym w realizacji krokiem obsługi awarii. Jeśli dojdzie do awarii TS, to mamy do czynienia z dwoma przypadkami:

1. **Awaria kontrolowana** — gdy TS sam wykryje u siebie awarię (np. poprzez niespełnienie którejś z asercji w swoim kodzie), wtedy sam poinformuje MS o błędzie i zakończy

pracę.

2. **Awaria nagła** — gdy TS zakończy pracę niespodziewanie (np. przez nagłą awarię zasilania) lub awarii ulegnie połączenie sieciowe między MS a TS. W obydwu tych przypadkach niemożliwa będzie komunikacja między obydwoma komponentami. Ten fakt został wykorzystany przy wykrywaniu awarii. MS i TS wysyłają między sobą wiadomości typu `PING` i jeśli okres oczekiwania na odpowiedź drugiej strony przekroczy ustaloną wartość, to sytuacja interpretowana jest jako awaria. TS kończy pracę, a MS przechodzi do procedury obsługi awarii.

Wstrzymanie operacji wykonujących się na osieroconych tabletach i przydzielenie ich pozostałym serwerom tabletów

Awaria mogła nastąpić w trakcie wykonywania się operacji na tabletach. Oczywiście dopóki tablet nie zostanie przydzielony nowemu TS, dopóty operacja nie może być kontynuowana — należy więc ją wstrzymać i zamiast niej wykonać operację przydziału tabletów nowemu TS.

W tym celu został wprowadzony specjalny rodzaj blokady udostępnionej przez `ResourceManager` — **blokada zawieszająca** (ang. *suspend lock*, w skrócie SL). Do założenia jej jest upoważniony jedynie `TakeTabletCoordinator` powołany w celu przydzielenia osieroconych tabletów nowym TS. Blokada ta ma tą właściwość, że po zgłoszeniu żądania założenia jej na dany zbiór tabletów T natychmiastowo przydzielony jest wyłączny dostęp do T i, do czasu jej zwolnienia, zawieszone są wszystkie inne blokady założone na tablety ze zbioru T (również zawieszające).

Każdy koordynator, który założył na dany tablet blokadę i w wyniku przydzielenia SL ją utracił, jest o tym fakcie informowany. Jego praca nie jest jednak zawieszana — dzięki temu może bez przeszkód kontynuować pracę na pozostałych tabletach, na które posiada założoną blokadę. Po zwolnieniu SL z powrotem przydzielane są zawieszone wcześniej blokady.

Po przydzieleniu blokady zawieszającej `TakeTabletCoordinator` rozdziela osierocone tablety między aktywne TS. Każdy z TS przywraca spójny stan otrzymanych tabletów, a następnie MS wznawia wstrzymane operacje.

Przywracanie spójnego stanu tabletu

Na **stan tabletu** składają się:

1. Zbiór plików z danymi znajdujących się w katalogu odpowiadającemu tabletowi.
2. Zawartość pliku `fileinfo.xml` (będziemy nazywać go **metaplikiem**).

Plik `fileinfo.xml` zawiera informacje o plikach z danymi odpowiadającemu tabletowi:

1. Nazwę pliku.
2. Numer pierwszej i ostatniej operacji `COMMIT` (ew. jej części), z których dane są przechowywane w pliku.
3. Rozmiar pliku.

Stan tabletu nazwiemy **niespójnym** jeśli zawartość metapliku nie odpowiada rzeczywistemu stanowi plików na dysku. Tak może się stać np. podczas wykonywania operacji `COMMIT`, gdy zostanie utworzony plik z danymi, ale dojdzie do awarii zanim informacja o tym zostanie dopisana do metapliku.

Zmiany w pliku `fileinfo.xml` wprowadzane są w taki sposób, aby nigdy nie doszło do sytuacji, że znajduje się w nim wpis dotyczący nieistniejącego pliku. To znaczy:

1. Jeśli plik jest dodawany — najpierw jest on tworzony na dysku, dopiero później umieszcza się wpis o nim w metapliku.

2. Jeśli plik jest usuwany — najpierw jest usuwany wpis w metapliku, dopiero później plik usuwa się fizycznie z dysku.

Przypomnijmy, że pliki z danymi są niemodyfikowalne, więc jedyne operacje na plikach, to właśnie dodawanie i usuwanie.

Po przejęciu osieroconego tabletu TS przywraca w nim spójność, tj. usuwa wszystkie pliki które nie mają swojego wpisu w metapliku.

Wznawianie wstrzymanych operacji

W momencie awarii w systemie mogły się wykonywać operacje, które miały założoną blokadę na osierocony tablet. Po zwolnieniu blokady zawieszającej uprzednio przydzielona tym operacjom blokada jest przywracana, a operacja jest wznawiana.

Przywracanie operacji zostanie przedstawione na podstawie przywracania operacji `COMMIT`.

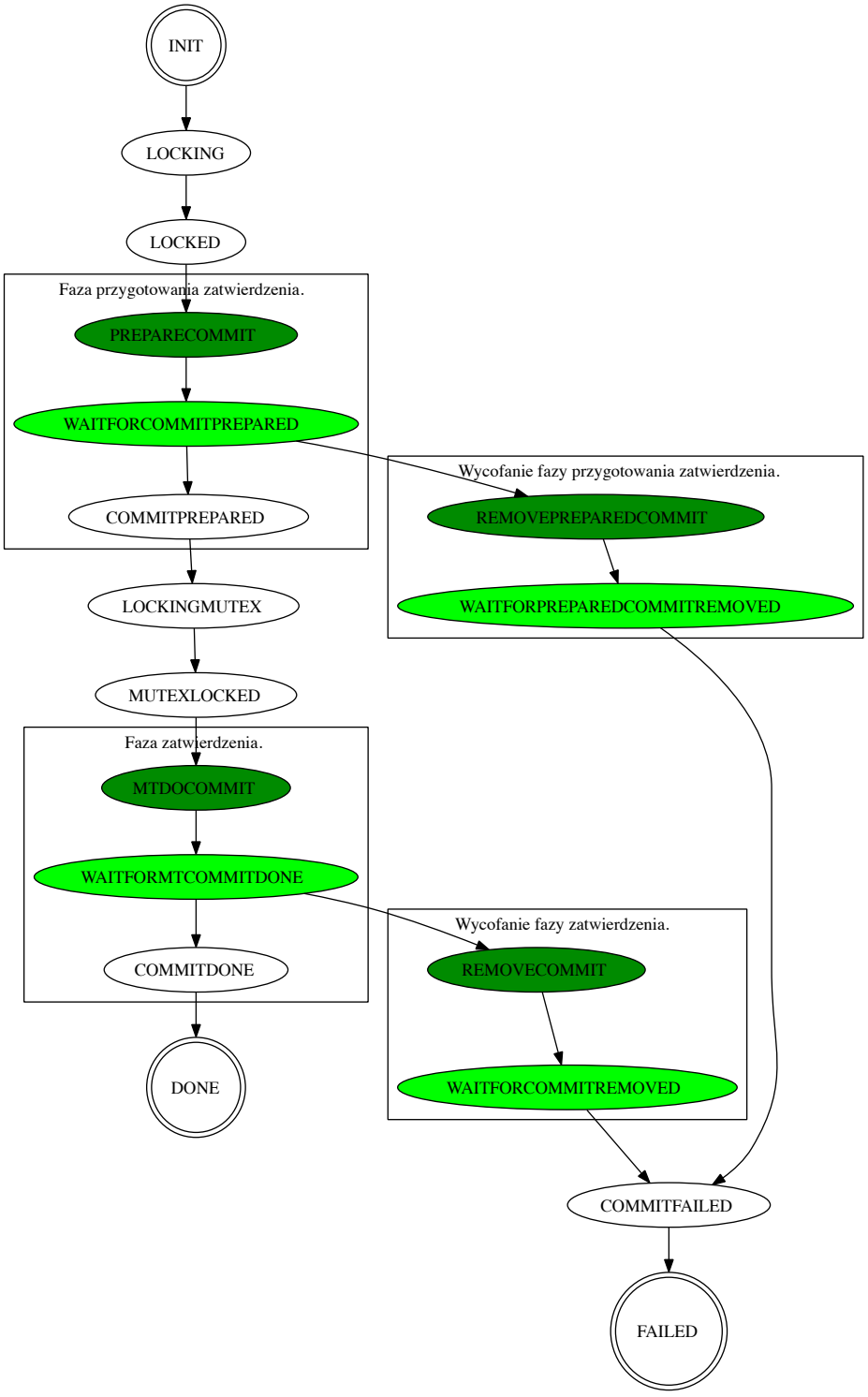
Na rys. 3.3 przedstawiony jest diagram stanów koordynatora operacji zatwierdzenia transakcji. W prostokątne ramki ujęto stany powiązane z daną fazą operacji. Kolorem ciemnozielonym zaznaczono stany, w których następuje wysłanie wiadomości, natomiast jasnozielonym te, w których następuje odczytanie wiadomości.

Schemat realizacji operacji zatwierdzenia transakcji przedstawiono w rozdziale 3.3.1. Najpierw omówimy ją w kontekście stanów koordynatora, następnie przedstawimy sposób reagowania MS gdy podczas wykonywania tej operacji dojdzie do awarii.

Stanem początkowym koordynatora jest `INIT`. Pierwszym etapem działania koordynatora jest zdobycie blokad do odczytu na potrzebne zasoby. Zasobami w tym przypadku są oczywiście tablety biorące udział w zatwierdzanej transakcji. Zgłoszenie żądania ma miejsce w stanie `LOCKING`, gdy zostanie przydzielone, koordynator przechodzi do stanu `LOCKED`. Po zakończeniu procedury zdobywania zasobów, zaczyna się realizacja fazy przygotowania zatwierdzenia. Koordynator przechodzi do stanu `PREPARECOMMIT`, w którym przesyła wszystkim TS wiadomość `PREPARECOMMIT`. Następnie przechodzi do stanu `WAITFORCOMMITPREPARED`, w którym oczekuje na potwierdzenia od TS — wiadomości `COMMITPREPARED`.

Jeśli nie wszystkie operacje po stronie TS zakończyły się sukcesem, następuje realizacja wycofania fazy przygotowania zatwierdzenia. Koordynator przechodzi do stanu `REMOVEPREPAREDCOMMIT`, wysyłając wiadomości o tej samej nazwie, oczekując w stanie `WAITFORPREPAREDCOMMITREMOVED` na odpowiedzi `PREPAREDCOMMITREMOVED`. Następnie przechodzi do stanu `COMMITFAILED`, w końcu do stanu końcowego — `FAILED`.

Jeśli faza przygotowania zatwierdzenia zakończy się sukcesem, to koordynator przechodzi do stanu `COMMITPREPARED`. Następnym krokiem jest zdobycie biletu na wykonanie operacji zatwierdzenia — żądanie uzyskania zgłoszone jest w stanie `LOCKINGMUTEX`, gdy bilet ten zostanie przydzielony, koordynator przechodzi do stanu `MUTEXLOCKED`. Wtedy generowany jest identyfikator tego zatwierdzenia, następnie zaczyna się realizacja fazy zatwierdzenia. Koordynator przechodzi wtedy do stanu `MTDOCOMMIT`, gdzie wysyła TS wiadomości o tej samej nazwie. Następnie w stanie `WAITFORMTCOMMITDONE` oczekuje na odpowiedzi `MTCOMMITDONE`. Jeśli nie wszystkie operacje po stronie TS zakończyły się sukcesem, to następuje realizacja wycofania fazy zatwierdzenia — analogiczna jak w przypadku wycofania fazy przygotowania zatwierdzenia. Natomiast gdy wszystkie operacje po stronie TS zakończą się sukcesem, koordynator przechodzi do stanu `COMMITDONE`, a następnie do finalnego `DONE`.



Rysunek 3.3: Diagram stanów koordynatora obsługującego operację zatwierdzenia (`CommitCoordinator`)

Teraz przejdziemy do opisu jak koordynator obsługuje awarie TS. Poczyńmy kilka obserwacji.

Po pierwsze, informowanie koordynatora, że na danym TS doszło do awarii wyrażone jest w języku rywalizacji o zasoby — jak już wspomnieliśmy, koordynator przydziału tabletu pobiera, a następnie zwalnia blokadę zawieszającą. Reakcja na awarie sprowadza się więc do reagowania na wywłaszczenie/ponowny przydział blokady ówcześnie pobranej przez koordynatora.

Po drugie, reakcja ta jest nietrywialna jedynie w przypadkach gdy do awarii dojdzie w chwili gdy koordynator znajduje się w stanach, w których dochodzi do komunikacji na linii MS – TS (zaznaczonych na zielono rys. 3.3).

Reakcja polega na przesłaniu TS, który przejął tablet **wiadomości retransmitowanej** — podobnej do tej, która wysłana była w momencie awarii (nazywać ją będziemy **wiadomością bazową**). Nazwa wiadomości retransmitowanej jest nazwą wiadomości bazowej z sufiksem `ENSURE` — np. dla `MTDOCOMMIT` będzie to `ENSUREMTDOCOMMIT`. Obsługa tej wiadomości po stronie TS jest podobna do obsługi wiadomości bazowej, z tą różnicą, że sprawdzane jest również czy operacja ta (bądź jej część) nie została już wcześniej wykonana przez TS, na którym doszło do awarii.

Dla przykładu, obsługa wiadomości `ENSUREMTDOCOMMIT` będzie polegać na sprawdzeniu, czy docelowy plik z danymi z tej transakcji już nie istnieje. Jeśli nie — kontynuowana jest normalna obsługa wiadomości `MTDOCOMMIT`. Analogicznie, obsługując wiadomość `ENSUREPREPARECOMMIT`, TS sprawdzi czy docelowy plik tymczasowy nie został już utworzony. Jeśli nie, to kontynuowana jest normalna obsługa wiadomości bazowej (która oczywiście w tym przypadku zakończy się błędem — polega ona na zapisaniu do tymczasowego pliku danych zapisanych w sposób nietrwały, które zostały utracone; tę transakcję należało będzie wycofać). Natomiast obsługując wiadomość `ENSUREREMOVEPREPAREDCOMMIT`, TS sprawdzi czy docelowy plik tymczasowy został usunięty, jeśli nie, to kontynuować będzie obsługę wiadomości bazowej.

Zauważmy, że format wiadomości będącej odpowiedzią na wiadomość retransmitowaną jest taki sam jak wiadomości bazowej — zarówno po wysłaniu `MTDOCOMMIT`, jak i `ENSUREMTDOCOMMIT` MS spodziewać się będzie wiadomości `MTCOMMITDONE` jako odpowiedzi. Tak więc już w momencie wysłania wiadomości retransmitowanej obsługa sytuacji awaryjnej po stronie MS się kończy.

Osiągnięto tym samym stawiane cele — obsługę sytuacji awaryjnych zarówno po stronie MS, jak i TS, sprowadzono do wykonania bardzo niewielu czynności, a następnie kontynuowaniu działania jak w sytuacji bezawaryjnej. Poza tym TS w kontekście wykonywanych operacji jest *bezstanowy* — wszystkie informacje o aktualnie wykonywanych operacjach są przechowywane jedynie po stronie MS.

3.4.3. Odporność na awarie serwera głównego

W odróżnieniu od serwerów tabletów, serwer główny jest scentralizowanym elementem systemu. W systemie nie istnieje węzeł, który za zadanie będzie miał przejęcie jego obowiązków w razie awarii. Awaria ta nie będzie mogła więc przebiec w sposób zupełnie transparentny dla użytkownika. Jedynym wymogiem będzie to, aby system powrócił do spójnego stanu i by spełnione były warunki ACID.

Zakłada się, że do awarii MS będzie dochodzić rzadko — aplikacja MS będzie uruchomiona na komputerze o podwyższonej niezawodności, podłączonego do sieci o dużej przepustowości i małej awaryjności.

Obsługa awarii MS nie jest jeszcze wspierana, dotychczas powstał jedynie zarys pomysłu na jej realizację, który poniżej zostanie przedstawiony.

MS gromadzi informacje o swoim **stanie**, w skład którego wchodzą:

- informacje o typach tabel: mapowanie $id\_typu \rightarrow nazwa\_typu$
- informacja o tabelach obecnych w systemie: identyfikator, nazwa, typ, id ostatniej operacji zatwierdzenia, z której dane znajdują się w tabeli, liczba tabletów wchodzących w skład tabeli, polityka podziału tabletu oraz czy na danej tabeli wykonywana jest właśnie operacja `DROP/TRUNCATE`.
- informacja o tabletach obecnych w systemie: identyfikator, granice obsługiwanych kluczy, id ostatniej operacji zatwierdzenia z której dane znajdują się na tym tablecie, polityka scalania plików

Informacje te są przechowywane w sposób trwały. Ze względów wydajnościowych (szybkość zapisu) nie planuje się przechowywania ich w MFS, ale np. na dyskach twardych połączonych w macierz RAID.

W przypadku awarii, z powyższych danych będzie można odtworzyć cały stan systemu, z wyjątkiem stanu trwających obecnie operacji. Wykonywanie wszystkich operacji zostanie anulowane, z wyjątkiem tych, w przypadku których nie da się przywrócić systemu do stanu sprzed tej operacji (są to `DROP`, `TRUNCATE` oraz `REMOVEOUTDATEDFILES` — wszystkie polegają na usunięciu plików). W szczególności oznacza to, że anulowane zostaną wszystkie trwające w czasie awarii transakcje.

Po ponownym uruchomieniu, schemat działania systemu będzie następujący:

- MS czeka na zgłoszenie się TS i rozdystrybuuje pomiędzy nie tablety oraz zleci dokończenie wszystkich wykonujących się w czasie awarii operacji `DROP` lub `TRUNCATE`;
- każdy z TS przywróci spójność w każdym ze swoich tabletów (dzięki czemu usunie wszystkie pliki będące pozostałościami po niezakończonych operacjach) i prześle MS informacje o plikach wchodzących w skład tabletu;
- MS zleci każdemu z TS usunięcie przestarzałych plików z każdego z tabletów — ponieważ w tym czasie w systemie nie działa żadna transakcja, to usunięte zostaną wszystkie pliki, które kiedyś były scalane;
- MS wyśle każdemu z TS id ostatniej zakończonej operacji zatwierdzenia, z której dane znajdują się w tabeli i zleci usunięcie wszystkich ewentualnych plików zawierających dane z późniejszych transakcji

.

W wyniku tego system zostanie przywrócony do spójnego stanu.

Rozdział 4

Testy poprawnościowe

W tym rozdziale opiszemy metody testowania GBT. Testowanie niezawodności jest o tyle trudniejsze, że dotyczy zachowania programu w przypadkach, które w środowisku produkcyjnym występują stosunkowo rzadko (np. reakcja na awarie). W związku z tym elementem testów jest również stworzenie środowiska symulującego zajście tych sytuacji.

W tym celu stworzyłem aplikację `failuretester` — środowisko symulujące awarie sieci oraz `testClient` — testową aplikację kliencką. W kolejnych podrozdziałach przyjrzymy się im bliżej, by na końcu przeanalizować wyniki testów.

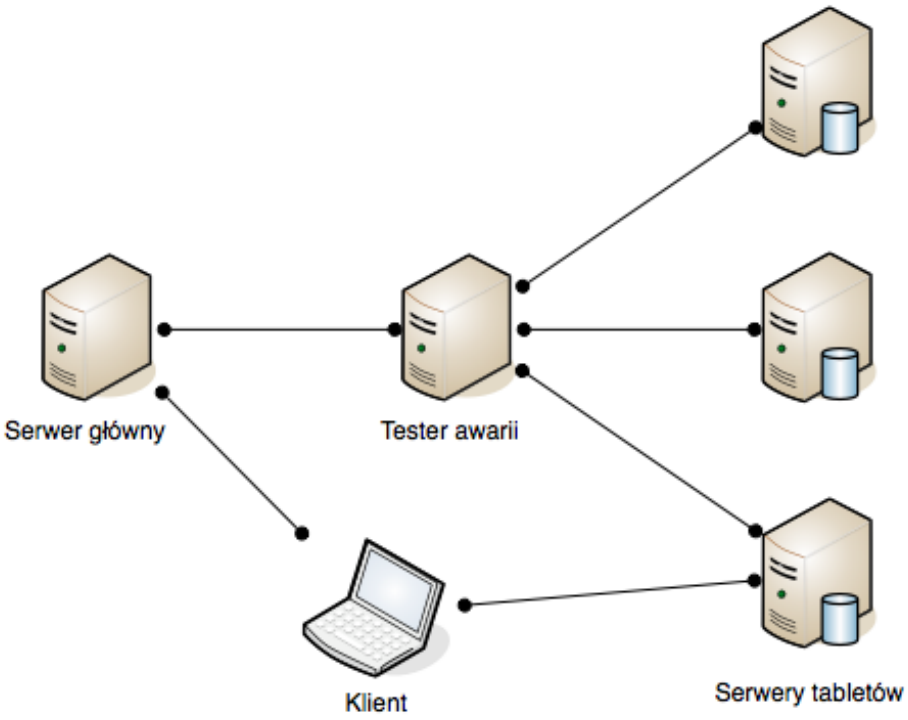
4.1. `failuretester` — aplikacja symulująca awarie

Tester awarii (`failuretester`, FT) to aplikacja symulująca awarie. Uruchamiając FT definiujemy komunikaty, które mogą zostać **odrzucone** — podajemy listę typów wiadomości m i prawdopodobieństw p, które należy rozumieć jako "po przyjściu wiadomości m z prawdopodobieństwem p dojdzie do awarii TS". Dzięki wylistowaniu typów wiadomości możemy sprecyzować jakiego typu awarie dopuszczamy w danej sesji testowej; natomiast zastosowanie prawdopodobieństwa sprawia, że poszczególne sesje nie realizują dokładnie tych samych scenariuszy.

Aplikacja `failuretester` działa jako pośrednik między MS a TS i analizuje ruch sieciowy między nimi, natomiast klient łączy się z systemem bezpośrednio. Dla klienta systemy z i bez testera awarii są więc nierozróżnialne (rys. 4.1). Jak łatwo zauważyć, aplikacja ta nie służy więc do testowania odporności na awarie klienta (lub połączenia na linii klient – TS lub klient – MS), ale ten przypadek, w przeciwieństwie do awarii MS i TS, jest prosty w obsłudze.

```
$ ./failuretester -h
Allowed options:
  -p [ --port ] arg (=6667)                    listening port
  -H [ --masterhost ] arg (=localhost)         master host
  -P [ --masterport ] arg (=6664)              master port
  -l [ --loglevel ] arg (=debug)               log level
  -t [ --immunets ] arg (=1)                   number of tabletservers which
                                               will be immune to failures
  -m [ --modifier ] arg (=0.90000000000000002) probability modifier
  -M [ --rejectable-messages ] arg             list of message types and
     their rejection probability (0-100)
  -h [ --help ]                                produce help message
```

Listing 4.1: Parametry aplikacji `failuretester`



Rysunek 4.1: Umiejscowienie testera awarii w topologii systemu

Gdy FT wykryje nadejście którejś z wiadomości podanej w parametrze, z odpowiadającym jej prawdopodobieństwem odłącza TS, który jest nadawcą/adresatem tej wiadomości.

Pozostałe parametry wywołania aplikacji `failuretester` znajdują się na listingu 4.1.

Parametr `immunets` określa liczbę serwerów tabletów, które mają nie być podatne na awarie. Z kolei jako parametr `modifier` aplikacja przyjmuje dodatnią liczbę wymierną, przez którą mnożone jest prawdopodobieństwo odrzucenia wiadomości, w momencie gdy ta wiadomość jest odrzucana. Daje to możliwość modyfikacji prawdopodobieństwa z biegiem życia programu. Jest to związane z faktem, że wraz z przyrostem danych w bazie wzrasta też liczba tabletów, a co za tym idzie — liczba wymienianych przez GBT komunikatów. Zmniejszenia prawdopodobieństwa awarii w kontekście *komunikatu* ma na celu zachowanie podobnego prawdopodobieństwa awarii w kontekście *aplikacji*. Na przykład, uruchamiając GBT z aplikacją `failuretester` z parametrem `PREPARECOMMIT 20`, w sytuacji, gdy w systemie jest jeden tablet, prawdopodobieństwo awarii podczas wykonywania operacji `COMMIT` wyniesie 20%, przy dwóch tabletach jest to 36%, trzech – 49%, czterech – 59%. Parametr `modifier` ma na celu zapobiec takiemu przyrostowi prawdopodobieństwa awarii.

4.2. `testClient` — aplikacja kliencka

Aplikacja kliencka `testClient` (TC) symuluje zachowanie klienta w systemie produkcyjnym: łączy się z bazą, tworzy kilka transakcji, w ramach których umieszcza dane, następnie je zatwierdzając (ew. odrzucając). Jednocześnie wraz z umieszczaniem danych w GBT aplikacja ta przechowuje je lokalnie, na końcu pracy porównując je ze sobą w celu sprawdzenia poprawności.

Parametry wywołania aplikacji przedstawione są na listingu 4.2

```
$ ./testClient -h
Allowed options:
  -H [ --masterhost ] arg (=localhost) master host
  -P [ --masterport ] arg (=6664)      master port
  -l [ --loglevel ] arg (=info)        log level
  -t [ --tablename ] arg (=tfailure)   tablename
  -N [ --numcommits ] arg (=40)        number of commits
  -n [ --numrecords ] arg (=7368107)   number of records (must be a prime
     number)
  -r [ --rollback ] arg (=0.0)         rollback probability
  -c [ --create ]                      create table at the beginning
  -u [ --truncate ]                    truncate table at the end
  -d [ --drop ]                        drop table at the end
  -h [ --help ]                        produce help message
```

Listing 4.2: Parametry aplikacji `testClient`

Priorytetem projektowym aplikacji było to, aby w ramach każdej transakcji umieszczane były dane z jak największego zakresu (aby jak najwięcej tabletów brało w niej udział) oraz by testy były szybkie (tj. należało np. unikać sprawdzania za każdym razem czy rekord, który ma zostać dodany, znajduje się już w bazie). Zdecydowałem się więc na następujące rozwiązanie: w ramach jednego testu umieszczam w bazie p rekordów, gdzie p jest liczbą pierwszą podaną jako parametr `numrecords`. Baza przechowuje rekordy o typie uint64 $\rightarrow$ (uint64, uint64). Kluczem są elementy grupy cyklicznej $\mathbf{Z}_p$. Jak wiadomo, każdy element tej grupy oprócz elementu neutralnego (0) jest jej generatorem, a co za tym idzie dodawanie go modulo p będzie generować nam parami różne elementy (przez p kroków). Jako generator przyjmowany jest $\lfloor \frac{p}{100} \rfloor$, a do bazy dodawane są elementy kolejno przez niego generowane. Dzięki temu osiągnięte zostały dwa priorytetowe założenia.

Wartością rekordu jest para liczb $(nr\_transakcji, nr\_rekordu)$, dzięki czemu w przypadku wykrycia niespójności między GBT a lokalną bazą będzie można łatwo dojść, która transakcja spowodowała problemy.

Opcjonalnie każda transakcja może zakończyć się wycofaniem, prawdopodobieństwo zajścia tego zdarzenia określony jest w parametrze `rollback`.

4.3. Środowisko testowe

Środowisko testowe składa się z 7 komputerów klasy PC, każdy wyposażony w dwu- lub czterordzeniowy procesor Intel Xeon o częstotliwościach zegara od 2,66 do 2,83 GHz, 4 GB pamięci RAM. Komputery te połączone są siecią lokalną o prędkości 1 Gb/s. Jeden z komputerów pełni rolę serwera głównego GBT, jeden — serwera głównego MooseFS, natomiast pozostałe 5 komputerów pełnią rolę zarówno serwerów tabletów GBT, jak i serwera kawałków MooseFS.

4.4. Testy odporności na awarie

Podstawowym celem wykonywania testów odporności na awarie jest sprawdzenie *poprawności* implementacji, tj. tego, czy system jest w stanie powrócić po awarii do spójnego stanu i wznowić dostarczanie przez niego usług. Wyniki tego typu testów są jednoznaczne i nie wymagają szerszego komentarza. Oprócz tego można zbadać wpływ awarii na wydajność systemu i temu właśnie poświęcimy niniejszy rozdział.

4.4.1. Warunki przeprowadzenia testów

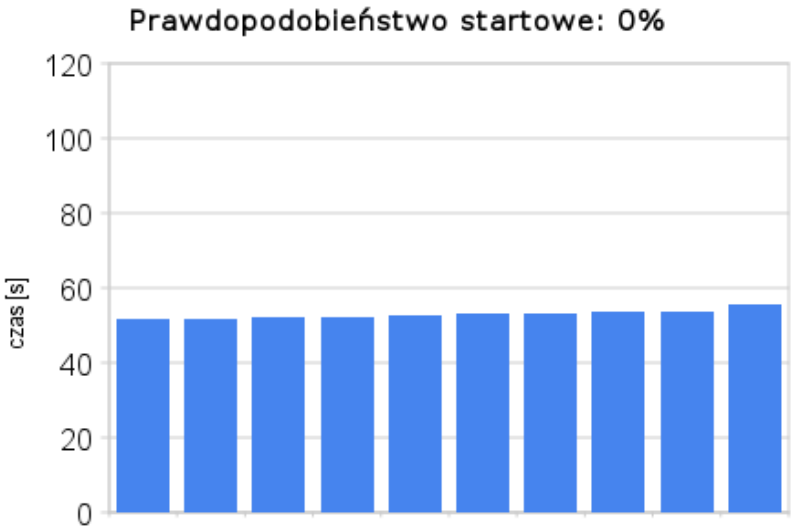
Wykonanych zostało 50 pomiarów, po 10 dla każdej z pięciu konfiguracji środowiska. Poszczególne konfiguracje różniły się częstością występowania awarii.

Za każdym razem test polegał na utworzeniu tabeli i wykonaniu na niej 10 transakcji, w sumie dodając 7368100 rekordów. Gdy w systemie dochodziło do awarii TS, po pewnym czasie w jego miejsce przyłączał się nowy TS.

Na wykresach zostały przedstawione wyniki wszystkich pomiarów. Na osi odciętych przedstawione są wyniki pomiarów poszczególnych instancji testów.

4.4.2. Wyniki testów

Pierwszy z testów był zrealizowany w bezawaryjnym systemie. Czas dodawania rekordów do bazy wyniósł średnio 52,99 s, przy odchyleniu standardowym 1,21 s (rys. 4.2). Różnice między poszczególnymi pomiarami były więc w tym przypadku niewielkie.



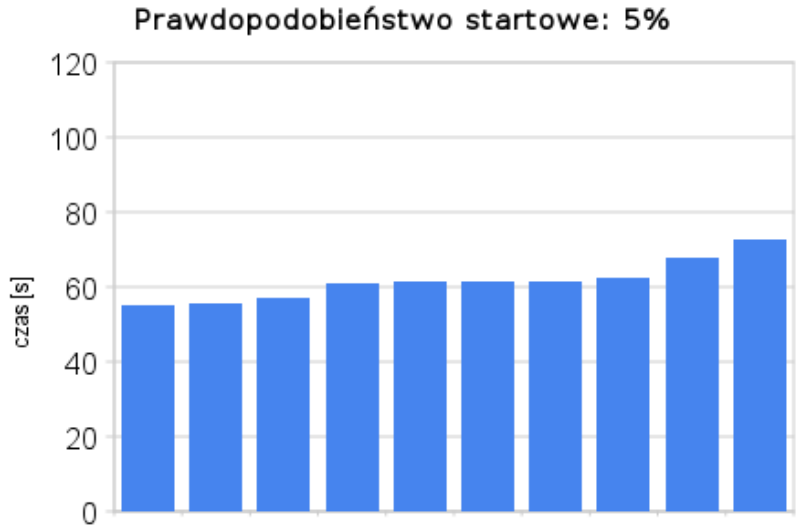
Rysunek 4.2: Wyniki testów w systemie bezawaryjnym

Kolejne pomiary dotyczyły systemu, w którym podczas jego pracy dochodziło do awarii — **prawdopodobieństwo startowe** wystąpienia awarii po odebraniu wiadomości miały wartość niezerową. Za każdym razem uruchamiano tester awarii z parametrem `modifier` ustawionym na 0,9 (a więc prawdopodobieństwo wystąpienia awarii zmniejszało się wraz z biegiem życia systemu).

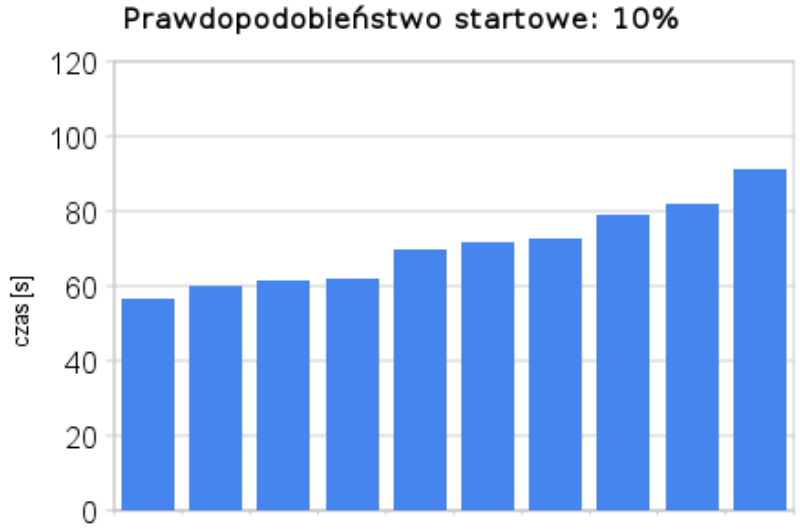
W przypadku, gdy prawdopodobieństwo startowe wynosiło 5% (rys 4.3), średni czas wykonania testu wynosił 61,57 s. Wyniki poszczególnych instancji testów nie były tak do siebie podobne jak w przypadku testów systemu bezawaryjnego; odchylenie standardowe wyniosło 5,36 s.

Gdy prawdopodobieństwo startowe zostało ustalone na 20% (rys. 4.4), średni czas przesyłu danych wyniósł 70,69 s (33% dłużej niż w przypadku systemu bezawaryjnego). W tym przypadku zanotowano też największe odchylenie standardowe — 11,01 s.

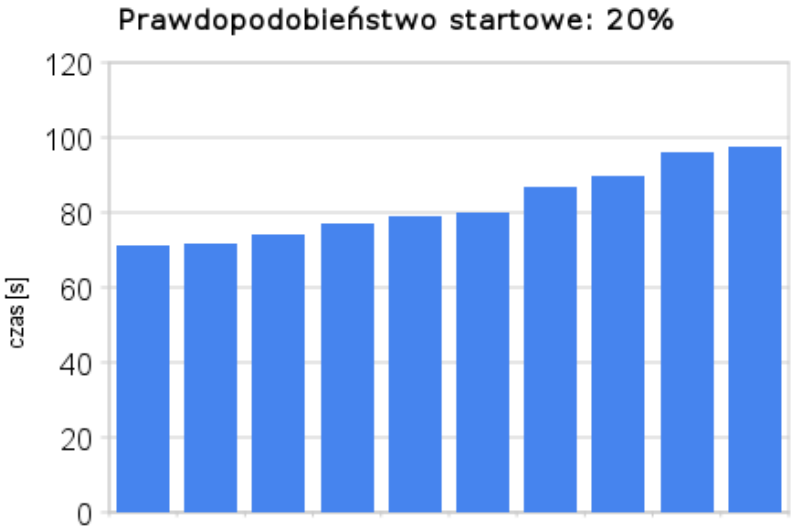
Przypadek prawdopodobieństwa startowego równego 20% (rys. 4.5) przynosi dalszy wzrost czasu potrzebnego na zapis danych (średnio 82,36 s, co jest o 55% większe niż czas wyjściowy), ale spadek odchylenia standardowego (9,7 s).



Rysunek 4.3: Wyniki testów w systemie awaryjnym o prawdopodobieństwie startowym wynoszącym 5%



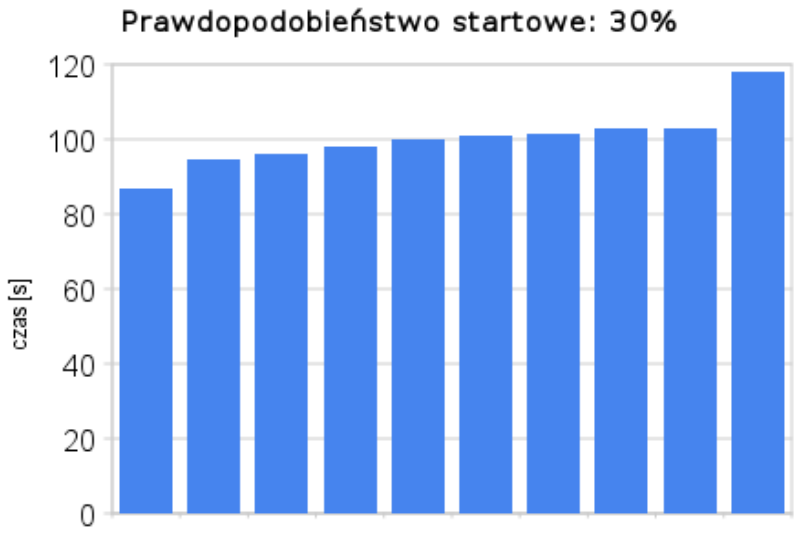
Rysunek 4.4: Wyniki testów w systemie awaryjnym o prawdopodobieństwie startowym wynoszącym 10%



Rysunek 4.5: Wyniki testów w systemie awaryjnym o prawdopodobieństwie startowym wynoszącym 20%

Testy w środowisku bardzo awaryjnym, bo wynoszącym początkowo 30% (rys. 4.6) oznaczają dalszy wzrost czasu potrzebnego na zapis danych (100,20 s, co oznacza wzrost w stosunku do systemu bezawaryjnego o 89%) oraz dalsze zmniejszenie odchylenia standardowego — 7,97 s.

Na wykresie 4.7 przedstawiono wykresy pokazujące związek między awaryjnością systemu a czasem potrzebnym na umieszczenie w nim danych. Oczywiście im większa awaryjność systemu tym ten czas jest większy — awarie zmniejszają liczbę węzłów w systemie (spadek mocy obliczeniowej), a także powodują wymóg wykonania pewnych dodatkowych czynności (np. wykonanie procedury obsługi awarii). Natomiast początkowy wzrost, a następnie spadek odchylenia standardowego można wytłumaczyć następująco: jeśli dojdzie do awarii TS obsługującego tablety biorące udział w transakcji, to zostanie ona wycofana przez GBT (a klient zmuszony do jej powtórzenia). Tak więc im później w trakcie transakcji dojdzie do awarii, tym więcej czasu zostanie "zmarnowanego". Oczywiście im rzadziej dochodzi do awarii, tym różnice w czasie wykonania poszczególnych instancji testów są mniejsze. Z kolei im większa awaryjność, tym bardziej równomierny rozkład awarii względem czasu życia transakcji.



Rysunek 4.6: Wyniki testów w systemie awaryjnym o prawdopodobieństwie startowym wynoszącym 30%

Rysunek 4.7: Czas potrzebny do zapisu danych w systemach charakteryzujących się różną awaryjnością

Rozdział 5

Podsumowanie

W ramach pracy stworzono część zapewniającą niezawodność w Gemius BigTable. Edsger W. Dijkstra pisał, że *"prostota jest niezbędna dla niezawodności"* i tej zasady starano się trzymać. Metoda synchronizacji operacji jest w pełni anonimowa (operacje aktualnie wykonywane nie muszą wiedzieć jakie inne operacje znajdują się w systemie) i generuje mały narzut (dzięki rozdrobnieniu definicji zasobu). Reakcja na awarie jest bardzo uogólniona, wymaga wykonania tylko niewielkiej pracy związanej z obsługą awarii, dalej system kontynuuje swe normalne działanie. Przywracanie spójnego stanu systemu jest proste i nie wymaga rozpatrywania wielu szczególnych przypadków (a spektrum stanów w jakich mógł pozostać system po nagłej awarii jest duże). Także realizacja warunków ACID dzięki wykorzystaniu wiedzy o sposobie przechowywania fizycznych danych w GBT jest przejrzysta i prosta.

Trudno prorokować, w którą stronę pójdzie informatyka w przeciągu kolejnych kilkudziesięciu lat, ale w dobie systemów obsługujących miliony użytkowników i zapewniających ciągłą dostępność usług, systemy rozproszone, charakteryzujące się wysoką dostępnością i wydajnością, mogą odegrać znaczną rolę. Ponieważ już w swojej naturze są bardziej skomplikowane od systemów scentralizowanych, wydaje mi się, że sztuka programowania rozproszonego długo będzie stawiała nacisk na opisaną wyżej prostotę.

W momencie oddawania tej pracy Gemius BigTable przestawał być projektem prototypowym, więc dopiero za jakiś czas będzie można realnie ocenić jak zastosowane rozwiązania sprawdziły się w warunkach produkcyjnych. Na pewno jednak ogrom zagadnień związanych z projektem wykracza poza temat jednej pracy magisterskiej i dostarczy wyzwań na kolejnych kilka lat. Szczególnie interesujące wydają się zagadnienia opracowania wydajnych polityk podziału tabletu i scalania plików czy równoważenia obciążenia w systemie poprzez dynamiczne zmiany przydziału tabletów do TS.

Bardzo dziękuję wszystkim osobom, które przyczyniły się do powstania tej pracy.

Dodatek A

Zawartość płyty CD

Na załączonej płycie CD znajdują się:

- `tweksej_mgr.pdf` — praca magisterska w formacie PDF,
- `BigTable/` — źródła Gemius BigTable
- `CommonLib/` — źródła biblioteki CommonLib
- `MooseFS/` — źródła Moose File System

Bibliografia

[BMDiff99] J. Bentley, D. McIlroy, *Data compression using long common strings*, AT&T Bell Labs., Murray Hill, NJ, 1999.

[BTab06] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber, *Bigtable: A Distributed Storage System for Structured Data*, Google Inc., 2006.

[Bhar85] Bharat Bhargava, *Building distributed database systems*, Computer Science Department Purdue University West Lafayette, Indiana, 1985.

[Birm05] Kenneth P. Birman, *Reliable distributed systems: technologies, Web services, and applications*, Springer, 2005.

[Chub06] Mike Burrows, *The Chubby lock service for loosely-coupled distributed systems*, Google Inc., 2006.

[Defa04] Xavier Defago, Peter Urban, Naohiro Hayashibara, Takuya Katayama, *The* Φ *accrual failure detector*, Japan Advanced Institute of Science and Technology, 2004.

[Elma03] Ramez Elmasri, Shamkant B. Navathe, *Fundamentals of Database Systems (4th Edition)*, Addison Wesley, 2003.

[GFS03] Sanjay Ghemawat, Howard Gobioff, Shun-Tak Leung, *The Google File System*, Google Inc., 2003.

[Laks09] Avinash Lakshman, Prashant Malik, *Cassandra - A Decentralized Structured Storage System*, Facebook Inc., 2009.

[MFS] Moose File System, `http://www.moosefs.com/`

[Marc00] Evan Marcus, Hal Stern, *Blueprints for High Availability: Designing Resilient Distributed Systems*, Wiley Computer Publishing, 2000.

[Purd06] Bharat Bhargava, *Distributed database systems*, materiały do wykładów na Wydziale Informatyki Purdue University (`http://www.cs.purdue.edu/homes/bb/cs542-06Spr/`), semestr wiosenny 2006.

[Ullm00] Garcia-Molina Hector, Ullman Jeffrey D., Widom Jennifer, *Podstawowy wykład z systemów baz danych*, Wydawnictwa Naukowo-Techniczne, 2000.

[WikiBT] Wikipedia: BigTable, `http://en.wikipedia.org/wiki/Bigtable`.

[WikiJFS] Wikipedia: Journaling file system, `http://en.wikipedia.org/wiki/Journaling_file_system`.