
Paweł Czernik
Paweł Gilewski
Szymon Janikowski
Michał Przyłuski

Rozproszone Systemy Operacyjne

DOKUMENTACJA Z PRZEBIEGU REALIZACJI
PROJEKTU Z RSO PRZEZ ZESPÓŁ B

WARSZAWA 2008

Raport z realizacji projektu z RSO, w roku akademickim 2007/2008.

Skład i łamanie: Zespół, przy pomocy systemu składu tekstu $\text{\LaTeX}2\epsilon$.

Złożone czcionką *Adobe Minion*.

Spis treści

| | | |
|----------|----------------------------------------------------------------------------------------------------|----------|
| 1 | Etap wstępny | 3 |
| 1.1 | System kontroli wersji <i>subversion</i> | 3 |
| 1.1.1 | Zarys instalacji | 3 |
| 1.1.2 | Użytkownie | 4 |
| 1.1.3 | Pielęgnacja | 7 |
| 1.2 | Konfiguracja <i>ssh</i> | 7 |
| 1.2.1 | Generacja kluczy | 7 |
| 1.2.2 | Instalacja klucza publicznego RSA | 8 |
| 1.2.3 | Generowanie kluczy DSA | 8 |
| 1.2.4 | Instalacja klucza publicznego DSA | 9 |
| 1.2.5 | Program <i>ssh-agent</i> | 9 |
| 1.2.6 | Program <i>ssh-add</i> | 9 |
| 1.2.7 | Niedogodności używania programu <i>ssh-agent</i> | 10 |
| 1.2.8 | Program <i>keychain</i> | 10 |
| 1.3 | Dołączanie nowych węzłów | 11 |
| 1.4 | Upadek węzła | 12 |
| 1.5 | Obsługa standardowych operacji na bazie danych w strukturze pierścieniowej. Blokowanie. | 12 |
| 1.5.1 | Nawiązywanie połączenia | 12 |
| 1.5.2 | Blokowanie | 13 |
| 1.6 | Replikacja | 14 |
| 1.6.1 | Wykrywanie potrzeby tworzenia replik danych. | 15 |
| 1.6.2 | Tworzenie replik poszczególnych tabel | 15 |
| 1.6.3 | Uaktualnianie replik | 16 |
| 1.6.4 | Zapis danych do plików | 16 |
| 1.7 | Architektura klienta | 16 |
| 1.7.1 | Styk użytkownik-aplikacja | 17 |
| 1.7.2 | Styk aplikacja-sieć systemu bazy danych | 17 |
| 1.8 | Harmonogram Prac | 18 |
| 1.8.1 | Etap I | 18 |
| 1.8.2 | Etap II | 18 |

| | | |
|----------|--------------------------------------------------------|-----------|
| 2 | Faza II | 21 |
| 2.1 | Komentarz do opracowania etapu II | 21 |
| 2.2 | Architektura | 21 |
| 2.3 | Budowa | 22 |
| 2.3.1 | Synchronizacja wątków | 23 |
| 2.3.2 | Komunikacja | 23 |
| 2.4 | Realizacja | 24 |
| 2.4.1 | Najwyższy poziom | 24 |
| 2.5 | Logika serwera | 25 |
| 2.5.1 | Schemat logiki serwera | 26 |
| 2.6 | Mechanizmy blokowania | 27 |
| 2.6.1 | Umieszczenie i struktury danych | 27 |
| 2.6.2 | Przykład przebiegu obsługi klientów | 27 |
| 2.7 | Replikacje i reakcja na rozłączenia serwerów | 28 |
| 2.7.1 | Replikacje podczas normalnego funkcjonowania | 29 |
| 2.7.2 | Sytuacje awaryjne - awaria liścia | 29 |
| 2.7.3 | Sytuacje awaryjne - awaria serwera głównego | 30 |
| 2.8 | Komunikacja | 30 |
| 2.8.1 | Pakiety przychodzące do serwera głównego | 30 |
| 2.8.2 | Pakiety wychodzące od serwera głównego | 30 |
| 2.8.3 | Opakowanie XML | 31 |
| 2.9 | Aplikacja kliencka | 31 |
| 2.10 | Silnik Bazy Danych | 31 |
| 2.10.1 | Struktury danych | 32 |
| 2.10.2 | Algorytmy | 33 |
| 2.11 | Język zapytań | 34 |
| 2.11.1 | Klasa Baza | 35 |
| 2.12 | Harmonogram Prac | 35 |
| 2.12.1 | Etap I | 35 |
| 2.12.2 | Etap II | 36 |
| 3 | Etap końcowy | 39 |
| 3.1 | Komentarz do opracowania etapu III | 39 |
| 3.2 | Opis realizacji komunikacji klient - serwer | 39 |
| 3.3 | Klaster testowy | 40 |
| 3.4 | Harmonogram Prac | 40 |
| 3.4.1 | Etap I | 40 |
| 3.4.2 | Etap II | 41 |
| 3.4.3 | Etap III | 42 |

Wstęp

Dokument ten ma za zadanie prezentować postępy w pracach nad realizacją projektu z przedmiotu *RSO* przez zespół *B*.

Cel projektu

Celem projektu jest *zaprojektowanie, zaimplementowanie, uruchomienie i przetestowanie rozproszonej bazy danych*.^[1]

Rozdział 1

Etap wstępny

Celem etapu wstępnego jest nakreślenie planu działania dla wszystkich uczestników projektu oraz rozpoznanie dziedziny.

W początkowej fazie projektowania zespół próbował realizować koncepcję, w której synchronizacja miała być realizowana przy pomocy algorytmów token ring. Poniżej przedstawiona jest właśnie ta koncepcja.

1.1 System kontroli wersji *subversion*

W celu zapewnienia bezproblemowej współbieżnej pracy przez wielu uczestników projektu, rozsądnym się wydaje wdrożenie odpowiedniego narzędzia pracy grupowej. Oprogramowaniem dobrze spełniającym tę funkcję jest m. in. *subversion*, do którego będziemy się odwoływać także poprzez skrót *SVN*.

1.1.1 Zarys instalacji

Repozytorium jest dostępne poprzez protokół `svn+ssh`. Taki mechanizm został wybrany, ze względu na bezpieczeństwo, oraz możliwość wykorzystania `ssh` do logowania się do repozytorium bez konieczności każdorazowego wprowadzania hasła. Jest to znaczące ułatwienie pracy dla wszystkich członków zespołu. Taka konfiguracja została przetestowana przez jednego z uczestników projektu w wielu poprzednich projektach, i nie miała żadnych wad. Z tego względu została wybrana (ponad dostępem lokalnym, po WWW, albo po `svnserve`, bez pomocy `ssh`).

Przygotowanie repozytorium przebiegło według następującego planu działania. Jest to realizacja zgodna z zarysem przedstawionym w rozdziale 6.6 książki poświęconej *SVN* [2].

1. Tworzenie kont użytkowników systemu. Przewidziano 4 konta dla uczestników projektu, wg. tabeli 1.1, oraz jedno konto o charakterze administracyjnym dla repozytorium o nazwie `rs0`. Została także utworzona grupa o takiej nazwie, a uczestnicy projektu stali się jej członkami. Taka organizacja umożliwia efektywne współdzielenie plików przez odpowiednie osoby.

2. Utworzenie repozytorium, poprzez wywołanie `svnadmin create`. Repozytorium założono w katalogu domowym użytkownika `rso`. Wykonane zostało także dowiązanie symboliczne z głównego katalogu.
3. Zmianienie uprawnień do katalogu `/ścieżka do repozytorium/rso/db` i jego podkatalogów, tak aby był ustawiony bit SGID. Powoduje to, że członkowie odpowiedniej grupy jeśli będą tworzyć pliki (a będą to robić pośrednio, przy każdym `commit`ie), grupa zachowa uprawnienia do tych plików. Jest to istotne, gdyż użytkownicy mogą być jednocześnie członkami wielu grup.
4. Ponadto w systemie był już *SVN* zainstalowany uprzednio, wraz ze skryptami opakowanymi m. in. `svnserve`, tak aby użytkownicy zdalnie modyfikowali pliki w repozytorium z odpowiednią maską.

Dodatkowo, aby usprawnić współdziałanie wszystkich użytkowników, oraz zapewnić każdemu użytkownikowi informacje o uaktualnieniach w plikach projektu, został do repozytorium dodany wyzwalacz na zdarzenie `post-commit`. Skrypt ten, wykonywany jest po pomysły `commit`ie do repozytorium, i powoduje, poprzez szereg pośrednich akcji, wysłanie do uczestników projektu wiadomości systemu Gadu-Gadu z informacją, że ktoś właśnie wprowadził nową wersję. Pozwala to uniknąć „aktywnego oczekiwania” na zmiany w repozytorium przez uczestników.

Dla użytkowników sugerowaną metodą dostępu do repozytorium jest wykorzystanie kluczy SSH. Dzięki temu nie będzie konieczne wielokrotne wprowadzanie hasła, w czasie dostępu do zasobów repozytorium przy zachowaniu pełnego bezpieczeństwa. Warto w tym momencie dodać, iż wszyscy uczestnicy projektu wygenerowali swoje klucze SSH (na swoich komputerach), oraz przesłali administratorowi repozytorium ich publiczne części. Administrator przypisał klucze do użytkowników systemu poprzez umieszczenie ich w odpowiednich plikach `authorized_keys`.

1.1.2 Użytkownie

Dostęp do repozytorium jest możliwy z poziomu każdego systemu operacyjnego z zainstalowanym odpowiednim oprogramowaniem. W przypadku systemów UNIXo pochodnych, jest to zestaw oprogramowania „subversion”. Może być on dystrybuowany w formie źródłowej oraz binarnej, prekompilowanej na daną architekturę (np. jako `SMCsubv14` — ta paczka została zainstalowana na serwerze, lub jako pakiet `rpm` dla dystrybucji linuxa). Dostępne są także graficzne interfejsy użytkownika, w szczególności *TortoiseSVN* dla systemu Microsoft Windows.

Użytkownicy loginy służące do dostępu do repozytorium wybrali zgodnie z tabelą 1.1.

Podstawowe operacje, jakie wykonuje użytkownik systemu sprowadzają się do *pobrania* aktualnej zawartości repozytorium, dokonania zmian w swojej lokalnej kopii, a następnie *zatwierdzenia* (ang. `commit`) tych zmian, tak aby stały się widoczne dla wszystkich uczestników projektu. Ponadto istotnymi operacjami są *importowanie* nowych plików do repozytorium, *przeglądanie* zmian i różnic między wersjami, a także *rozwiązywanie konfliktów*.

W pierwszej kolejności użytkownicy powinni dokonać konfiguracji swojego środowiska *SVN*. Najłatwiej tego dokonać edytując plik zlokalizowany w `~/ .subversion/config` w taki sposób, aby znalazły się w nim linie:

| login | imię i nazwisko |
|---------|-------------------|
| apos | Paweł Gilewski |
| exile | Szymon Janikowski |
| jowisz | Paweł Czernik |
| mikylie | Michał Przyłuski |

Tabela 1.1: Nazwy użytkowników w repozytorium *SVN*.

```
[helpers]
editor-cmd = vi
```

```
[miscellany]
enable-auto-props = yes
```

```
[auto-props]
*.c = svn:eol-style=native;svn:keywords=Id
*.cpp = svn:eol-style=native;svn:keywords=Id
*.h = svn:eol-style=native;svn:keywords=Id
*.txt = svn:eol-style=native;svn:keywords=Id
*.tex = svn:eol-style=native;svn:keywords=Id,Rev
Makefile = svn:eol-style=native;svn:keywords=Id
```

Zakładamy, że wszyscy użytkownicy będą wykorzystywać system UNIX lub jego pochodne. Założenie takie wynika z faktu, iż docelową platformą, na której ma działać rozwiązanie, jest pewna liczba komputerów działających pod linuxem, a zatem programowanie projektu w systemie Microsoft Windows mogłoby przyczynić się do pewnych dodatkowych trudności związanych z, tylko częściową, przenośnością.

Zakładamy, że wszyscy uczestnicy projektu zapoznali się z dokumentacją na temat *SVN* wytworzoną przez poprzednie zespoły laboratoryjne. Z tego względu tutaj znajdzie się jedynie skrócony podręcznik użytkownika, opisujący najbardziej powszechnie zdarzające się operacje na repozytorium.

Podstawowe operacje

W celu pobrania całego repozytorium należy wykonać komendę następującej postaci.

```
$ svn co svn+ssh://tin.myszak-szyszak.pl/rso
```

Spowoduje to pobranie zawartości repozytorium, do nowoutworzonego w bieżącym katalogu, podkatalogu „rso”. Wszystkie dalsze operacje będą zakładały *przebywanie w tym właśnie katalogu*. Operację pobrania całego repozytorium zasadniczo wykonujemy raz — na komputerze, na którym będziemy edytować bądź dodawać pliki. Wszystkie dalsze operacje będą się sprowadzały do uaktualniania naszej kopii repozytorium.

Kolejną istotną operacją jest dodawanie plików do repozytorium. Przebiega to w prosty sposób, tj.

```
$ svn add README
A      README
```

Pozostaje jeszcze zatwierdzenie zmian w lokalnej kopii. Dopiero po zatwierdzeniu nasze zmiany staną się widoczne dla innych uczestników projektu.

```
$ svn commit
Adding      README
Transmitting file data .
Committed revision 1.
```

Gdy dokonamy pewnych zmian w lokalnej kopii, warto czasem sprawdzić jakie faktycznie są różnice, w porównaniu z ostatnio pobraną wersją. Można tego dokonać wykonując komendę `svn diff`. Wypisze one wynik UNIXowej komendy `diff` dla wszystkich zmienionych plików, względem ostatnio przez nas pobranej wersji repozytorium. Możliwe jest także porównanie zmian względem *aktualnej* wersji repozytorium, bez jego pobierania, dodając przełącznik `-u` do powyższej komendy.

Warta uwagi jest także komenda `svn status`, która pokazuje status każdego zmienionego (lub niewersjonowanego) pliku w bieżącym katalogu roboczym *SVN*. Znaczenie poszczególnych liter jest opisane szczegółowo przy opisie komendy `svn status` w rozdziale 9 [2].

Możliwe jest także tworzenie katalogów, przenoszenie plików oraz usuwanie. Pod żadnym pozorem nie należy tych obiektów (wersjonowanych) usuwać lub przenosić przy pomocy standardowych komend powłoki. Spowoduje to ich ponowne odtworzenie przy najbliższym uaktualnieniu kopii roboczej repozytorium. Do tego służą komendy `svn mv`, `svn mkdir` oraz `svn rm`.

Najistotniejszą, w pewnym sensie, jest komenda `svn update`. Służy ona do uaktualnienia naszej lokalnej kopii obecną zawartością repozytorium. Komenda ta, jeśli nie wystąpią konflikty, połączy zmienione przez nas pliki z ew. zmianami innych użytkowników.

Operacje zaawansowane

Jedną z najistotniejszych operacji zaawansowanych (tj. wykraczających poza podstawowy przepływ *pobierz-zmień-zatwierdź*) jest rozwiązywanie konfliktów. Warto szczegółowo opisać co należy zrobić w takiej sytuacji.

Do konfliktu dochodzi w sytuacji, gdy zmiany wprowadzone przez użytkowników się wykluczają. Rozważmy hipotetyczną sytuację, dwóch użytkowników *A* i *B*, oraz plik `README` w repozytorium. Niech plik `README` będzie aktualnie w rewizji 54. Zakładamy, że zarówno *A*, jak i *B* pobrali tę rewizję. Przyjmy, iż *A* wykasował pewną linię, a *B* dokonał pewnej zmiany w tej samej linii. Kluczowe dla zrozumienia tego problemu, jest uświadomienie sobie, że te zmiany się wykluczają. Z tego względu konflikty występują stosunkowo rzadko.¹ Niech *A* zatwierdzi swoją zmianę, repozytorium zmieniło swoją wersję do 55. Problem się pojawi, gdy *B* będzie chciał

¹W większości przypadków użytkownicy edytując ten sam plik zmieniają jego różne fragmenty, i *SVN* jest w stanie (w locie, podczas zatwierdzania zmian) połączyć je ze sobą.

zatwierdzić swoją zamianę. Należy podkreślić, że problem nie wynika ze zmiany wersji repozytorium, lecz wykluczających się modyfikacji w pliku. Gdy *B* spróbuje zatwierdzić swoje oprogramowanie zgłosi błąd, i do jego katalogu roboczego dołączą 3 nowe pliki, tj. `README.mine`, `README.r54` oraz `README.r55`. Ponadto, edytowany przez *A* plik `README` będzie zawierał zarówno fragmenty starej, nowej, oraz naszej wersji. Powinniśmy teraz *świadomie* wyedytować plik `README`, tak aby zawierał to co powinien zawierać ostatecznie. Należy rozważyć przy tym co inni do niego wprowadzili, i ew. się z nimi skonsultować. Gdy już plik uzyska swoją postać docelową należy wykonać `svn resolved README`, a następnie zatwierdzić zmiany jak zwykle.

Taki mechanizm postępowania sprawdza się w większości przypadków. Jednakże, w systemie *SVN* istnieje również możliwość blokowania plików. Realizowane jest to podkomendą `svn lock`. Zablokowanie pliku przez jednego użytkownika uniemożliwi jakiegokolwiek jego zmiany przez innych. Nie należy korzystać z tego rozwiązania bez zgody kierownika projektu, lub sytuacji wyższej konieczności. Blokada trwa do póki nie zostanie zdjęta przez jej autora. Możliwe jest także siłowe zdjęcie blokady przez innego użytkownika.

1.1.3 Pielęgnacja

W celu zapewnienia maksymalnego bezpieczeństwa zgromadzonych danych, repozytorium jest poddawane replikacji, codziennie w nocy ok. godziny 4:50. Zostało to zrealizowane poprzez następujący wpis w pliku `/var/spool/cron/crontabs/root`:

```
45 4 * * * [ -x /usr/local/bin/svnbkp ] && /usr/local/bin/svnbkp
```

oraz utworzenie pliku `svnbkp` zawierającego:

```
#!/bin/sh
DDD=`date +%d%m%y` \
/usr/bin/svnadmin hotcopy /rso /raid/svn-backup/backup$DDD
chmod 700 /raid/svn-backup/backup$DDD
```

Powyższe zapewnia utworzenie gorącej kopii repozytorium, oraz zapisanie jej na (zlokalizowanym na innym komputerze, montowanym po NFS'ie) systemie plików. Warto nadmienić, iż ten system plików jest wewnętrznie macierzą RAID-5. Wszystko to, w celu zapewnienia nieprzerwanej pracy, jest zasilane poprzez UPSa o znacznej mocy.

Podjęte zabezpieczenia wydają się być wystarczające wobec potencjalnych niebezpieczeństw wynikających z awarii sprzętu. Bezpieczeństwo danych ze względu na nieautoryzowany dostęp również zostało poddane analizie. Stwierdzono, że aktualnie wykorzystywana wersja SSH jest bezpieczna, i nie ma innych bezpośrednich zagrożeń zewnętrznych dla integralności danych na serwerze pełniącym funkcję serwera *SVN*.

1.2 Konfiguracja *ssh*

1.2.1 Generacja kluczy

Pierwszym krokiem w konfiguracji autoryzacji RSA jest wygenerowanie pary kluczy. Wygenerowanie pary kluczy wygląda następująco:

```
$ ssh-keygen
Generating public/private rsa1 key pair.
Enter file in which to save the key (/home/drobbins/.ssh/identity):
(wciśnij enter)
Enter passphrase (empty for no passphrase): (podaj passphrase)
Enter same passphrase again: (podaj je ponownie)
Your identification has been saved in /home/drobbins/.ssh/identity.
Your public key has been saved in /home/drobbins/.ssh/identity.pub.
The key fingerprint is:
a4:e7:f2:39:a7:eb:fd:f8:39:f1:f1:7b:fe:48:a1:09 drobbins@localbox
```

Podanie „passphrase” pozwala zabezpieczyć klucz prywatny przed nadużyciami, jednakże powoduje pewną niewygodę. Teraz przy każdym połączeniu z kontem drobbins@remotebox, ssh zapyta o to passphrase.

1.2.2 Instalacja klucza publicznego RSA

W tym celu należy zaakceptować domyślne położenie kluczy (.ssh/identity oraz .ssh/identity.pub) i podać bezpieczne passphrase. Następnie należy skonfigurować zdalne systemy z sshd aby mogły wykorzystać do autoryzacji utworzony klucz publiczny. Zatem należy skopiować klucz na serwer ssh na przykład za pomocą polecenia „scp”: scp .ssh/identity.pub drobbins@remotebox: Ponieważ autoryzacja RSA nie została jeszcze w pełni skonfigurowana, system zapyta o hasło. Następnie należy się zalogować na remotebox i dopisać utworzony klucz do pliku .ssh/authorized_keys:

```
ssh drobbins@remotebox
drobbins@remotebox's password: (podaj hasło)
Last login: Thu Jun 28 20:28:47 2001 from localbox.gentoo.org
```

Welcome to remotebox!

```
$ cat identity.pub >> ~/.ssh/authorized_keys
$ exit
```

Przy kolejnych próbach logowania do remotebox, klient ssh powinien już nie pytać o hasło, tylko o „passphrase” (jeśli „passphrase” nie została wprowadzona to już można uzyskać połączenie bez hasła – opcja ta nie jest polecana ze względu na bezpieczeństwo) :

```
ssh drobbins@remotebox
Enter passphrase for key '/home/drobbins/.ssh/identity':
```

1.2.3 Generowanie kluczy DSA

Podczas gdy klucze RSA są używane przez wersję 1 protokołu ssh, zaktualizowana wersja 2 korzysta z kluczy DSA. Każda w miarę aktualna wersja OpenSSH powinna obsługiwać oba rodzaje kluczy. Wygenerowanie kluczy DSA programem ssh-keygen wygląda podobnie do generowania kluczy RSA:

```
ssh-keygen -t dsa
```

Tutaj również program zada zapytanie o „passphrase” i o położenie nowych kluczy. Opcja domyślna, zazwyczaj `.ssh/id_dsa` oraz `.ssh/id_dsa.pub`, powinna wystarczyć. Po ukończeniu jednorazowej generacji, klucz publiczny trzeba umieścić na zdalnych systemach.

1.2.4 Instalacja klucza publicznego DSA

Instalacja klucza publicznego DSA wygląda niemal identycznie jak w przypadku RSA. Należy skopiować plik `.ssh/id_dsa.pub` na `remotebox` i umieścić go tam w `.ssh/authorized_keys2`. Tym razem plik ten ma inną nazwę niż dla RSA. Po zainstalowaniu klucza, do zalogowania na `remotebox` zamiast hasła, trzeba podać „passphrase”.

1.2.5 Program *ssh-agent*

`ssh-agent` (część pakietu OpenSSH), to program specjalnie zaprojektowany do pracy z kluczami RSA i DSA. „`ssh-agent`” jest demonem zaprojektowanym w celu buforowania odszyfrowanych kluczy prywatnych. `ssh` zawiera wbudowane wsparcie, umożliwiające mu komunikację z `ssh-agent` i pozwala na posiadanie odszyfrowanych kluczy prywatnych bez konieczności potwierdzania hasła przy każdym nowym połączeniu. W tym celu należy po prostu użyć programu „`ssh-add`” żeby dodać klucz prywatny do pamięci podręcznej agenta `ssh`. Jest to jednorazowy proces; po użyciu `ssh-add`, `ssh` przechwyci klucz prywatny od `ssh-agent`, zamiast czekać na potwierdzenie hasła. Kiedy `ssh-agent` się uruchamia, zwraca kilka ważnych zmiennych środowiskowych przed odłączeniem od powłoki i kontynuowaniem pracy w tle. Oto kilka przykładowych wydruków wygenerowanych przez `ssh-agent` w momencie uruchomienia:

```
ssh-agent
SSH_AUTH_SOCK=/tmp/ssh-XX4LkMJS/agent.26916; export SSH_AUTH_SOCK;
SSH_AGENT_PID=26917; export SSH_AGENT_PID;
echo Agent pid 26917;
```

Jednym ze sposobów uruchomienia `ssh-agent` jest zamieszczenie wpisu w pliku `.bash_profile`, co sprawia że wszystkie programy uruchomione w powłoce logowania zobaczą zmienne środowiskowe, przez co będą w stanie go zlokalizować i w razie potrzeby zapytać o klucze.

1.2.6 Program *ssh-add*

`ssh-agent` startuje z pustym buforem odszyfrowanych kluczy prywatnych. Zanim będzie można użyć `ssh-agent`, trzeba najpierw dodać klucze prywatne do bufora „`ssh-agent`”, co umożliwia polecenie „`ssh-add`”. W poniższym przykładzie, użyto `ssh-add` w celu dodania `.ssh/identity` klucza prywatnego RSA do bufora `ssh-agent`:

```
ssh-add ~/.ssh/identity
Need passphrase for /home/drobbins/.ssh/identity
Enter passphrase for /home/drobbins/.ssh/identity
(enter passphrase)
```

Czyli ssh-add zapytał o hasło po to aby gotowy do użycia klucz prywatny mógł być odszyfrowany i przechowany w buforze ssh-agenta.

1.2.7 Niedogodności używania programu *ssh-agent*

Zawarty w pliku `.bash_profile` wpis powoduje że nowa kopia ssh-agent jest uruchamiana dla każdej sesji logowania. Oznacza to że trzeba użyć ssh-add żeby dodać klucz prywatny dla każdej nowej kopii ssh-agent. Jeśli jest otworzony jeden terminal lub konsola, nie ma to wielkiego znaczenia, ale zazwyczaj otwartych jest kilka terminali i trzeba wpisać hasło za każdym razem gdy otwierana jest nowa konsola. Inny problem z domyślnym ustawieniem ssh-agent jest taki, że nie jest zgodny z żądaniami crona. Odkąd procesy są uruchomione przez crona, nie będą dziedziczyć zmiennej `SSH_AUTH_SOCK` z ich środowiska i przez to nie będą wiedziały, że proces ssh-agent jest uruchomiony albo jak się z nim skontaktować.

1.2.8 Program *keychain*

Keychain jest to skrypt napisany w bashu, który umożliwia używanie procesu ssh-agent jednego na system, a nie na sesje logowania. Skrypt ten pomaga również optymalizować proces ssh-add poprzez próbę dodania kluczy prywatnych, które nie są w buforze działającego ssh-agenta. Działania keychain polega na sprawdzeniu czy ssh-agent został już uruchomiony. Jeżeli nie, wtedy uruchomi ssh-agent i zapisze ważne zmienne `SSH_AUTH_SOCK` i `SSH_AGENT_PID` w pliku `.ssh-agent` dla bezpieczeństwa i do przyszłego użycia. Najlepszy sposób na uruchomienie keychain, tak jak w przypadku użycia czystego ssh-agent, jest umieszczenie odpowiednich ustawień wewnątrz `.bash_profile`:

```
#!/bin/bash
#example ~/.bash_profile file
/usr/bin/keychain ~/.ssh/id_rsa
#redirect ~/.ssh-agent output to /dev/null to zap the annoying
#"Agent PID" message
source ~/.ssh-agent > /dev/null
```

Teraz zmienna `SSH_AUTH_SOCK` jest zapisana w `.ssh-agent`, co sprawia że skrypty powłoki i zadania crona mogą w prosty sposób połączyć się z ssh-agent pobierając plik źródłowy `.ssh-agent`. Sam keychain również korzysta z tego pliku. W momencie uruchomienia sprawdza czy ssh-agent jest uruchomiony. Jeżeli tak, używa pliku `.ssh-agent` do pobrania właściwych ustawień `SSH_AUTH_SOCK`, umożliwiających mu użycie istniejącego agenta zamiast uruchamianie nowego. Keychain uruchomi nowy proces ssh-agent tylko jeżeli plik `.ssh-agent` jest przestarzały (wskazuje na nieistniejący ssh-agent) lub jeśli `.ssh-agent` nie istnieje.

Teraz kiedy keychain jest umieszczony w `/usr/bin/`, należy go dodać do `.bash_profile`, podając ścieżki do kluczy prywatnych jako argumenty. Oto standardowy plik z poprawnie aktywowanym keychain zawarty w pliku `.bash_profile`:

```
#!/bin/bash
#on this next line, we start keychain and point it to the private
```

```
#keys that we'd like it to cache
/usr/bin/keychain ~/.ssh/id_rsa ~/.ssh/id_dsa
source ~/.ssh-agent > /dev/null
#sourcing ~/.bashrc is a good thing
source ~/.bashrc
```

Po skonfigurowaniu pliku `.bash_profile` żeby wywoływał `keychain` przy każdym logowaniu, wylogowaniu i powtórnym logowaniu, `keychain` będzie uruchamiał `ssh-agent`, zapamiętywał ustawienia zmiennych środowiskowych agenta w pliku `.ssh-agent`, a potem prosił o podanie hasła dla każdego klucza prywatnego podanego jako argument wiersza poleceń w pliku `.bash_profile`. Po wpisaniu hasła klucze prywatne zostaną dodane do bufora i `keychain` zakończy działanie. Następnie pobierane są informacje z pliku `.ssh-agent` i zostaje zainicjowana sesja logowania do użycia przez `ssh-agent`. Teraz, jeżeli nastąpi wylogowanie i powtórne logowanie, `keychain` sam odnajdzie istniejący proces `ssh-agent`, który nie zakończył się gdy nastąpiło wylogowanie. Dodatkowo, `keychain` zweryfikuje to czy klucze prywatne które były wpisane znajdują się już w buforze `ssh-agent`. Jeżeli nie, wtedy trzeba będzie podać właściwe hasło, ale jeżeli wszystko pójdzie dobrze, istniejąca wersja `ssh-agent` wciąż będzie zawierać poprzednio dodane klucze prywatne. Dzięki temu nie trzeba ponownie podawać hasła. Po zalogowaniu będzie można kopiować (`scp`) i logować (`ssh`) się zdalnie. Nie trzeba korzystać z `ssh-add` po zalogowaniu się, również `ssh` i `scp` nie będą prosiły o podanie hasła. W rzeczywistości, dopóki początkowy proces `ssh-agent` jest uruchomiony, będzie można się zalogować i nawiązać połączenia `ssh` bez konieczności podawania hasła. `ssh-agent` może kontynuować swoje działanie do momentu, gdy komputer zostanie ponownie uruchomiony.

1.3 Dołączanie nowych węzłów

Proces uruchamiania systemu polega na uruchomieniu usługi przez administratora na dowolnym komputerze wchodzącym w skład systemu rozproszonego służącego do obsługi bazy danych. Zostanie to zrealizowane przy pomocy pewnej grupy komputerów pracujących pod kontrolą systemu linux. W katalogu w plikiem wykonywalnym serwera bazy danych znajdować się będzie także jego plik konfiguracyjny. Na jego podstawie demon usługi, jest w stanie ustalić jakie są adresy IP serwerów, na których powinny zostać uruchomione instancje bazy danych. Serwer uruchomiony na pewnym początkowym węźle będzie starał się zapewnić uruchomienie odpowiedniej liczby kopii bazy danych, w celu spełnienia wymagania minimalnej liczby aktywnych węzłów (replik danych).

Pierwszą czynnością, którą powinien wykonać demon usługi w trakcie uruchamiania, jest sprawdzenie, czy inny węzeł nie stara się równolegle uruchomić innej kopii systemu bazy danych. Aby to zapewnić przez pewien określony czas (np. 300ms) oczekuje na sygnał z sieci informujący o przebiegającym procesie uruchamiania systemu. W przypadku, gdy nie stwierdzi obecności innej kopii systemu przystąpi do właściwej procedury uruchamiania poszczególnych węzłów. Przebiegać to będzie z wykorzystaniem bezhasłowego logowania się na inne komputery przy pomocy `ssh`, w trybie wymiany kluczy. Klucze te nie będą chronione `passphrase`, aby umożliwić ich wykorzystanie bez interaktywnego udziału operatora.

W dalszej kolejności — po uruchomieniu wystarczającej liczby procesów bazy danych na odpowiednich węzłach — system rozpocznie swoją replikację. W tym momencie specjalna funkcjonalność demona startującego kończy się, i staje się on jednym ze zwyczajnych węzłów. Struktura sieci nie przewiduje specjalizowanych węzłów odpowiedzialnych za utrzymanie spójności sieci. Zapewnienie komunikacji między węzłami, tj. zachowanie formy pierścienia, zostanie zrealizowane w oparciu o algorytmy bazujące na strukturze pierścienia. Oznacza to w szczególności wykrywanie awarii w sieci na podstawie przerwy w komunikacji — tj. nieotrzymaniu żetonu przez określony okres czasu.

1.4 Upadek węzła

W przypadku, gdy jeden z węzłów utraci łączność z siecią uruchomiona zostanie procedura reorganizacji pierścienia. Jeśli z sieci zniknął węzeł, który był aktualnie w posiadaniu żetonu, pozostała część sieci przeprowadzi odwrotne tworzenie żetonu zgodnie z algorytmami stosowanymi w sieciach opartych na Token Ring. Istotną rolę w tym procesie ma ustalenie czy zachodzi taka konieczność, bowiem w przypadku, gdy odłączeniu ulegnie pojedynczy węzeł nieposiadający żetonu, wprowadzi to tylko pewne opóźnienie w działaniu sieci, gdyż pewien jego poprzednik będzie starał się przekazać żeton do węzła, który uległ awarii.

Gdy sieć ulegnie podziałowi na dwie lub więcej podsieci kluczową rolę w podjęciu decyzji o konieczności odtworzenia żetonu podejmuje się na podstawie informacji ilościowych o węzłach w odpowiednich odseparowanych sieciach. Obliczenia te mają jedynie charakter przybliżony, i nie są krytyczne dla samego działania sieci, ale wpływają na optymalność podjętej przez system decyzji o nowym kształcie systemu. Brane są pod uwagę także informacje o przechowywanych przez pozostałe przy życiu węzły informacjach w bazie danych. Podsieć, która uzna, że obejmuje najwięcej węzłów odtworzy token. Pozostałe węzły zawieszają swoje działania, ze względu na brak żetonu w swojej podsieci.

Po pewnym czasie dojdzie do przywrócenia łączności między węzłami. W takiej sytuacji algorytmy utrzymania spójności pierścienia pozwolą na ponowne włączenie węzłów do sieci z żetonem. Przeprowadzona zostanie procedura mająca na celu rozprzestrzenienie danych do nowo-podłączonych węzłów, oraz włączenie ich w strukturę sieci (tj. zapewnienie, że otrzymają żeton w odpowiednim momencie).

1.5 Obsługa standardowych operacji na bazie danych w strukturze pierścieniowej. Blokowanie.

1.5.1 Nawiązywanie połączenia

Zakłada się istnienie sieci serwerów zawierających kopie tabel. Komputery w sieci są zorganizowane w logiczny pierścień z krążącym znacznikiem zarządzającym („token”) jak to zostało opisane w poprzedniej części dokumentacji.

Aplikacja kliencka uzyskuje informacje o tym, z jakimi adresami ip może łączyć się klient. Będzie to robić na podstawie sztywnej listy.

W celu wykonania zapytania wybierany jest dowolny z uzyskanych adresów. Łączenie kończy się sukcesem w momencie kiedy serwer potwierdzi, że funkcjonuje w działającej sieci. Odpowiedź o pomyślnym nawiązaniu połączenia wysyłana jest gdy:

- w węźle do którego łączy się klient znacznik był relatywnie „niedawno” — zostanie tutaj ustalony odpowiedni limit czasowy
- gdy ustalony w powyższym punkcie czas od ostatniej obecności znacznika nie jest spełniony serwer wstrzymuje się z odpowiedzią do momentu w którym przyjdzie do niego znacznik.

Powyższe obwarowania są konieczne, ponieważ musimy mieć pewność, że połączyliśmy się z węzłem faktycznie będącym częścią funkcjonującego prawidłowo systemu. W przypadku, gdyby nawiązywanie połączenia zakończyło się niepowodzeniem wybierany jest w sposób losowy kolejny węzeł z listy posiadanej przez aplikację kliencką.

1.5.2 Blokowanie

W celu zachowania spójności i synchronizacji bazy danych w czasie operacji na niej konieczne jest wprowadzenie blokad zapisu (dla danych odczytywanych) jak i również blokad zapisu i odczytu (dla danych zapisywanych). Pojawiają się tu klasyczne problemy zagadnień synchronizacyjnych — w tym przypadku problem czytelników i pisarzy. Wobec częstego, relatywnie, występowania odczytów nad zapisami konieczne jest wprowadzenie priorytetu dla zapisujących.

Oprócz oczywistych blokad koniecznych do operowania na danych zgromadzonych w już istniejących tabelach, konieczna jest blokada na tworzenie nowych tabel, która obejmuje wszystkie węzły istniejące w sieci. Zabezpiecza to przed utworzeniem, przykładowo, dwóch tabel o tej samej nazwie a odmiennej strukturze atrybutów

Sytuacja usuwania tabeli da się dużo łatwiej zaklasyfikować jako zwykła modyfikacja wymagająca blokady zapisu i odczytu

Obsługa standardowego odczytu

Kiedy klient wie, że połączył się z funkcjonującym prawidłowo serwerem przesyła mu treść zapytania. Na tym etapie serwer dokonuje optymalizacji wydajnościowej. Do zrealizowania zapytania wykorzystany powinien zostać węzeł, który posiada daną tabelę i jest w obecnym momencie najmniej obciążony. Informacja o tym jest uzyskiwana w następujący sposób:

1. Serwer czeka na znacznik
2. Gdy znacznik się pojawi serwer wpisuje w strukturę znacznika informację o tym, że interesuje go, kto posiada poszukiwaną tabelę A i jeśli ją posiada — jak bardzo jest obciążony.
3. Następuje obieg znacznika dookoła pierścienia — każdy z serwerów przez który przechodzi jest odpytany o obecność danej tabeli.
4. Jeśli dana tabela znajduje się w danym węźle i nie jest zablokowana do odczytu inkrementowany jest semafor dla blokady zapisu. W znaczniku zapisywana jest para informacji serwer-obciążenie.

5. Serwer inicjujący odczytuje informacje i nawiązuje na oddzielnym porcie połączenie z najmniej obciążonym serwerem.
6. Następuje obsługa zapytania poprzez oddzielny kanał i przesłanie informacji do klienta
7. W podobny sposób następuje zdjęcie blokady zapisu.

Ze względu na to że nawiązywanie nowych połączeń za pomocą gniazd jest operacją dosyć kosztowną po zakończeniu obsługi danego zapytania połączenie takie nie jest rozwiązywane. Niemniej zawsze przed przesłaniem takim kanałem informacji przeprowadzana jest cała powyższa procedura

Obsługa zapisu

Procedura obsługi od procedury odczytu różni się przede wszystkim priorytetem wykonania. Wzorując się na klasycznym problemie synchronizacyjnym „czytników i pisarzy” należy wprowadzić dodatkowy semafor, który będzie oznaczał że od tej pory żaden nowy klient nie może dostać się do zasobu. Nie możemy jeszcze jednak przeprowadzić zapisu lub modyfikacji zasobu w trakcie gdy jest on odczytywany. Tak więc z założeniem pełnej, „prawdziwej” blokady należy się wstrzymać do momentu wycofania się „czytników”. Powinien to wykrywać krążący znacznik.

Algorytm zapisu będzie więc następujący:

1. Serwer obsługujący połączenie z klientem czeka na krążący znacznik
2. Gdy znacznik się pojawi dodaje do niego żądanie zablokowania danej tabeli do zapisu i odczytu
3. W czasie obiegu znacznika serwery posiadające tabelę zachowują się w dwojaki sposób:
 - jeśli aktualnie mają korzystających z tabeli "czytników" ustawiają znacznik blokujący wpuszczanie kolejnych i czekają aż odczyt zostanie zakończony wtedy nastąpi pełna blokada a węzły wpiszą do znacznika informację o sukcesie (przy którymś z kolejnych okrążeń)
 - jeśli aktualnie nie ma żadnych blokad blokowany jest odczyt i zapis i na bieżąco w tokenie jest zapisywana informacja o sukcesie
4. Serwer nadający czeka aż z którymś obiegiem tokena dostanie informację że wszystko zostało zablokowane - wtedy deleguje oddzielnymi kanałami zadanie zapis
5. Serwer oczekuje aż wszyscy zakończą transmisję i wysyła token z żądaniem zdjęcia blokady

1.6 Replikacja

W przypadku częstych i dużych odczytów (na przykład całych tabeli na raz), korzystne powinno być tworzenie połączeń między procesami nie sąsiadującymi ze sobą w kręgu. Jeśli w tokenie,

przy zapytaniu, przekazany będzie identyfikator maszyny i procesu, który zapytanie do pierścienia wprowadził, maszyny umiejące na pytanie odpowiedzieć prześlą potrzebne informacje bezpośrednio. Mogą przeszukać listę wcześniej stworzonych gniazd, sprawdzić, czy połączenie z autorem polecenia odczytu nadal jest aktywne, ewentualnie utworzyć z pytającym serwerem nowe połączenie. Następnie, używając identyfikatora (jak choćby numer id) zapytania, może przekazać w potrzebną tabelkę bezpośrednio na zainteresowany serwer. Tak powstałe powiązania poza pierścieniem systemu pełnią pomocniczą funkcję, od ich trwałości nie zależy stabilność całej bazy.

W przypadku modyfikacji tabel można pozwolić sobie na przeczekanie, aż cały pierścień zostanie okrążony przez token uzbrojony w polecenie modyfikacji. Powrót oznaczałby, że wszystkie repliki modyfikacji dokonały. Można jednak użyć kanałów przesyłu danych (tworzonych jak przy odczycie połączeń poza pierścieniem), aby wysłać potwierdzenia dokonania modyfikacji przez pierwszy węzeł, który faktycznie je zrealizuje.

1.6.1 Wykrywanie potrzeby tworzenia replik danych.

Dzięki temu, że w przyjętym rozwiązaniu każdy węzeł przez moment w cyklu jest mózgiem systemu, możemy podzielić odpowiedzialność za sprawdzanie konieczności tworzenia kopii tablic. Procedura sprawdzenia może być wszczynana cyklicznie (co określoną liczbę przekazów tokenu). Dodatkowo, w przypadku awarii pierścienia, proces który inicjalizował naprawę samej struktury systemu, może płynnie przejść w fazę sprawdzania co z danych powinno zostać naprawione. Procedura sama polega na dodaniu do tokenu odpowiedniego zapytania. W odpowiedzi na nie kolejne węzły uzyskujące token dopisują pary (nazwa_tabeli, ile_kopii) lub inkrementują liczniki przy już istniejących wpisach. Po obiegu całego pierścienia węzeł inicjujący zliczanie ocenia, czy i których tabeli jest za mało i reaguje stosownie do zapotrzebowania.

1.6.2 Tworzenie replik poszczególnych tabel

Polecenie tworzenia replik tabel przekazywane jest w systemie za pomocą tokenu. W przypadku wykrycia za małej ilości poszczególnych tabeli, serwer aktualnie posiadający token sprawdza, czy któryś z poprzedników już się problemem nie zajął, czy nie istnieje już podpisane pod token polecenie „niech ktoś zgłosi się po tabelę X”. Jeśli taka tabela nie znajduje się na aktywnym i uprzywilejowanym w danej chwili komputerze, zdejmuje on wspomniane polecenie z tokena i zgłasza zapytanie o całą tabelę.

W celu początkowego rozsypania tabeli po węzłach lub po prostu w sytuacji, kiedy wiele różnych tabeli musi zostać powielonych, serwer wykrywający tę sytuację reaguje w sposób bardziej skomplikowany. Tworzy listę poleceń „niech ktoś zgłosi się po tabelę X_i ”. Węzeł otrzymujący później taki token wyjmując z tej listy jednorazowo tylko jedną pozycję — tabelkę której nie posiada. Zgłasza zapytanie o treść całej tej tabeli. Reszty listy w tym obiegu już nie dotyka. Wykorzystując listę, możemy rozpropagować tabele po wszystkich węzłach systemu, unikając sytuacji, w której węzły posiadające wszystkie tabele zgrupowane są w pierścieniu tuż za węzłem inicjującym działanie aplikacji.

1.6.3 Uaktualnianie replik

W przypadku poprawnej pracy pierścienia polecenia modyfikacji tabeli przemieszczają się wraz z tokenem. Wszystkie repliki otrzymują sygnał do modyfikacji zawartości. W przypadku wypadnięcia któregoś procesu z pętli, po ponownym jego przyłączeniu, powinna istnieć możliwość uaktualnienia posiadanych przez węzeł tabel bez potrzeby przesyłania całej tabeli. W tym celu wszystkie kopie dowolnego zestawu danych wyposażone są w dziennik modyfikacji. Dziennik ten zapisuje co najmniej numery identyfikacyjne modyfikujących zapytań, ale także ich treści. Po powrocie z procesu przyłączania węzła ma on ustawiany na tabelach znacznik, rozkazujący węzłowi uaktualnić dane. W tym celu, po otrzymaniu tokena, węzeł zgłasza zapytania o dzienniki modyfikacji dla tabel. Wyraża też zainteresowanie wpisami po tych, które jemu wydają się ostatnimi. Dalsze węzły, jeśli posiadają sprawną kopię tabeli, którą ktoś chce tak uaktualniać, wykorzystują drogę taką samą jak w przypadku przesyłu całej tabeli danych. Nawiązują lub odkurzają połączenie z zainteresowanym dziennikami węzłem i przesyłają cały wpis o modyfikacjach a także aktualny stan blokad na tabeli.

1.6.4 Zapis danych do plików

System przewiduje możliwość zapisu całej bazy danych do plików. W celu określenia węzła, na którym dane mają być złożone, zapytanie przyjmuje argument, będący wskazaniem który proces lub na której maszynie proces ma się tym zapisem zająć. W metodach obsługi tego typu zapytań ze strony klienta można zaszyć zestaw maszyn które takiego backupu danych dokonają domyślnie. Serwer mający dokonać zapisu plików u siebie zgłasza prośby o przesłane mu konkretnych, całościowych tabel. Po ściągnięciu tabeli z wykorzystaniem mechanizmów tworzenia replik danych zapisywana jest ona do pliku tekstowego. Początkowe wiersze tego pliku stanowią nagłówki pozwalający odtworzyć tabelę z zapisanych następnie danych.

1.7 Architektura klienta

Z punktu widzenia użytkownika końcowego aplikacja klienta jest kluczowym elementem systemu. To ona odpowiada za komunikację użytkownika, z całą siecią pełniącą funkcję bazy danych. Z tego względu szczegółowemu dopracowaniu ten aplikacji należy poświęcić szczególną uwagę.

Z punktu widzenia systemu, jako układu wszystkich urządzeń i osób, oddziaływujących między sobą, powinniśmy wyodrębnić dwa punkty interakcyjne związane z aplikacją kliencką. Pierwszy z nich, to miejsce kontaktu użytkownika — aktora — w sensie inżynierii oprogramowania, z całym systemem bazy danych. Tu dochodzi do wymiany komunikatów użytkownika z aplikacją, które powinny być przeprowadzone w formie dialogu między nimi, i zakończyć się spełnieniem oczekiwań użytkownika. Drugim punktem interakcyjnym jest miejsce włączenia aplikacji klienckiej do struktury sieci. W tej warstwie aplikacja kliencka jest odpowiedzialna za odpowiednie zgłaszanie swoich żądań do sieci, odbieranie odpowiedzi, oraz zapewnienie trwałego połączenia z siecią.

1.7.1 Styk użytkownik-aplikacja

Aplikacja zostanie zrealizowana podobnie jak tzw. „interaktywne terminale” znanych systemów zarządzania bazą danych. Będzie to tekstowy interfejs użytkownika, Użytkownikowi zostanie udostępniony pewien podzbiór języka SQL, tak, aby mógł wykonywać wszystkie, przewidziane specyfikacją zadania, operacje na danych i ich strukturze. Aplikacja ta będzie także odpowiedzialna za prezentację danych pobranych z bazy danych.

W toku analizy potencjalnych rozwiązań odrzucony został pomysł napisania graficznego interfejsu użytkownika (przy pomocy Javy bądź .NETa) ze względu na istotny dodatkowy nakład pracy, przy faktycznej utracie ergonomii korzystania i wątpliwych innych korzyściach. Aplikacje służące do obsługi znacznych ilości danych nie powinny do swojej obsługi wymagać myszki, a standardowe operacje powinny być łatwo dostępne. Realizacja aplikacji graficznej uniemożliwiłaby także jakiegokolwiek automatyzację testowania. W przypadku aplikacji tekstowej, możliwe jest napisanie prostego skryptu (perl/powłoka), który zbada wiele przypadków testowych automatycznie, bez konieczności ręcznego ich kontrolowania.

Poza standarowym wejściem i wyjściem, aplikacja kliencka będzie dodatkowo korzystać z pliku konfiguracyjnego. Plik ten będzie dostępny dla użytkownika, i pozwoli m. in. na ustalenie adresu serwera, z którym aplikacja ma nawiązać połączenie. Możliwe będzie także określenie adresów serwerów zapasowych.

1.7.2 Styk aplikacja-sieć systemu bazy danych

W warstwie komunikacji z siecią implementującą system zarządzania bazą danych aplikacja kliencka jest bardziej złożona. Musimy tu wyróżnić jej istotne fragmenty.

Pierwszym z nich jest utrzymanie skutecznego połączenia z siecią bazy danych. Chodzi o uodpornienie aplikacji na chwilowe (bądź trwałe) problemy komunikacyjne z danym węzłem dostępowym. Pierwsze połączenie, dla świeżo uruchomionego klienta, zostanie wykonane na zapisany w konfiguracji adres. W przypadku niepowodzenia aplikacja losowo będzie próbować nawiązać połączenie z innymi wymienionymi w konfiguracji serwerami. Jeśli to się nie powiedzie aplikacja nie jest w stanie już nic zrobić i powinna zawiesić swoją działalność informując użytkownika przy tym o zainstalowanej sytuacji.

Założmy jednak, że aplikacji uda się połączyć z bazą. W tej sytuacji, wybrany przez nią serwer powinien poinformować aplikację o swoich ew. zastępnikach. Szczegóły tego protokołu zostaną zdefiniowane w innej części, opisującej strukturę sieci.

Gdy aplikacja dokona wstępnej komunikacji z serwerem może przejść do zadawania zapytań. W tym celu aplikacja przedstawia swoje zapytanie serwerowi, z którym ma już nawiązane połączenie, a on albo je realizuje, albo sugeruje aplikacji połączenie się z innym serwerem w celu jego wykonania. Takie rozwiązanie wynika z faktu, że utrzymywanie stałych połączeń (nawet tylko wewnątrz sieci bazy danych) jest trudne w realizacji. Problemy pojawiają się w momencie, gdy któryś węzeł zniknie z sieci. Jeszcze gorsze byłoby utrzymywanie stałych połączeń z wieloma węzłami przez aplikację kliencką, bowiem ona z założenia nie powinna bazować na znajomości całej, lub istotnie dużego fragmentu, sieci serwerowej. Z drugiej strony, każdorazowe nawiązywanie połączenia w ramach sieci lokalnej (serwerów) również jest kosztowne. Dodatkowo pojawia się problem pojedynczego punktu potencjalnego upadku systemu. Wprawdzie klient się przełączy na

inny węzeł, ale bardzo wiele operacji może zostać przerwanych, ze względu na upadek serwera, z którym bezpośrednio był połączony klient.

1.8 Harmonogram Prac

1.8.1 Etap I

11 marca

- wymiana kontaktów,
- wstępne zapoznanie się z dokumentacjami z poprzednich lat.

18 marca

- ustalenie podstawowych założeń projektu (uruchamianie poprzez ssh, komunikacja poprzez sockety).

1 kwietnia

- opracowanie pierwszej koncepcji projektu — serwer główny + serwery z danymi,
- po konsultacji z prowadzącym, zmiana pomysłu na sieć serwerów równorzędnych z pierścieniem z krążącym znacznikiem,
- podział pracy przy dokumentacji:
 - Paweł Gilewski — replikacja
 - Michał Przyłuski — aplikacja kliencka i svn
 - Paweł Czernik — uruchamianie i struktura token-ring (awaria, dołączanie węzłów)
 - Szymon Janikowski — mechanizm blokad i zachowanie spójności w bazie

1.8.2 Etap II

21 kwietnia

- Wyróżnienie czynności wstępnych jakich należy się podjąć, w celu przygotowania gruntu pod właściwą implementację
- Przydzielenie wyróżnionych działań do poszczególnych członków zespołu:
 - Paweł Czernik — Oprogramowanie realizujące uruchamianie systemu za pomocą zdalnego logowania
 - Paweł Gilewski — Zestawienie połączeń pomiędzy poszczególnymi komputerami klastra
 - Michał Przyłuski — Prototyp aplikacji klienckiej. Problem podłączania użytkownika do klastra.

- Szymon Janikowski — Zaprojektowanie protokołów komunikacyjnych dla poszczególnych warstw systemu tj. pseudo języka zapytań SQL i protokołu komunikacji wewnętrznej.

15 maja

- Omówienie koncepcji wielowątkowego serwera zakładającej podział na warstwy komunikujące się ze sobą za pomocą kolejek. Wyszczególnienie klas zajmujących się komunikacją na każdym z wyróżnionych poziomów.
- Przedstawienie skryptu mającego na celu delikatne uruchamianie aplikacji na poszczególnych węzłach sieci klastra. Omówienie szczegółów dopracowania tego elementu (Paweł Czernik)

22 maja

- Testowanie skryptu uruchomieniowego w laboratorium OpenSSI (Paweł Czernik)
- Testowanie nawiązywania połączeń poprzez gniazda BSD (Paweł Gilewski)
- Dyskusa na temat koncepcji projektu - postawienie pod znakiem zapytania słuszności koncepcji z równorzędnymi węzłami synchronizowanymi za pomocą Token Ring. Przedstawienie wad i zalet obu rozwiązań.
- Podział zadań zasadniczej fazy implementacyjnej:
 - Paweł Czernik — wykonanie parsera zapytań pseudo-języka SQL i silnika Bazy Danych
 - Michał Przyłuski — dopracowanie koncepcji klas komunikujących się za pomocą kolejek, mechanizmów komunikacji pomiędzy wątkami oraz zadań synchronizacyjnych na poziomie pojedynczego serwera
 - Paweł Gilewski — realizowanie komunikacji na poziomie serwera z wykorzystaniem wielopoziomowej struktury klas
 - Szymon Janikowski — analizowanie potencjalnych sposobów organizacji serwera w związku z powstałymi wątpliwościami. Testowanie powstających fragmentów aplikacji. Dokumentowanie dokonań.

29 maja

- Dokładne wyszczególnienie podziału serwera na warstwy. Omówienie zrealizowanych rozwiązań i wyjaśnienie sposobu organizacji, komunikacji i korzystania z hierarchii klas (Michał Przyłuski)
- Podjęcie decyzji o zmianie koncepcji i powrót do tej najbardziej pierwotnej (z której zespół zrezygnował po konsultacji z prowadzącym 1 kwietnia), czyli serwera głównego (+ ewentualny serwer zapsaowy) i serwery zawierające dane.

Rozdział 2

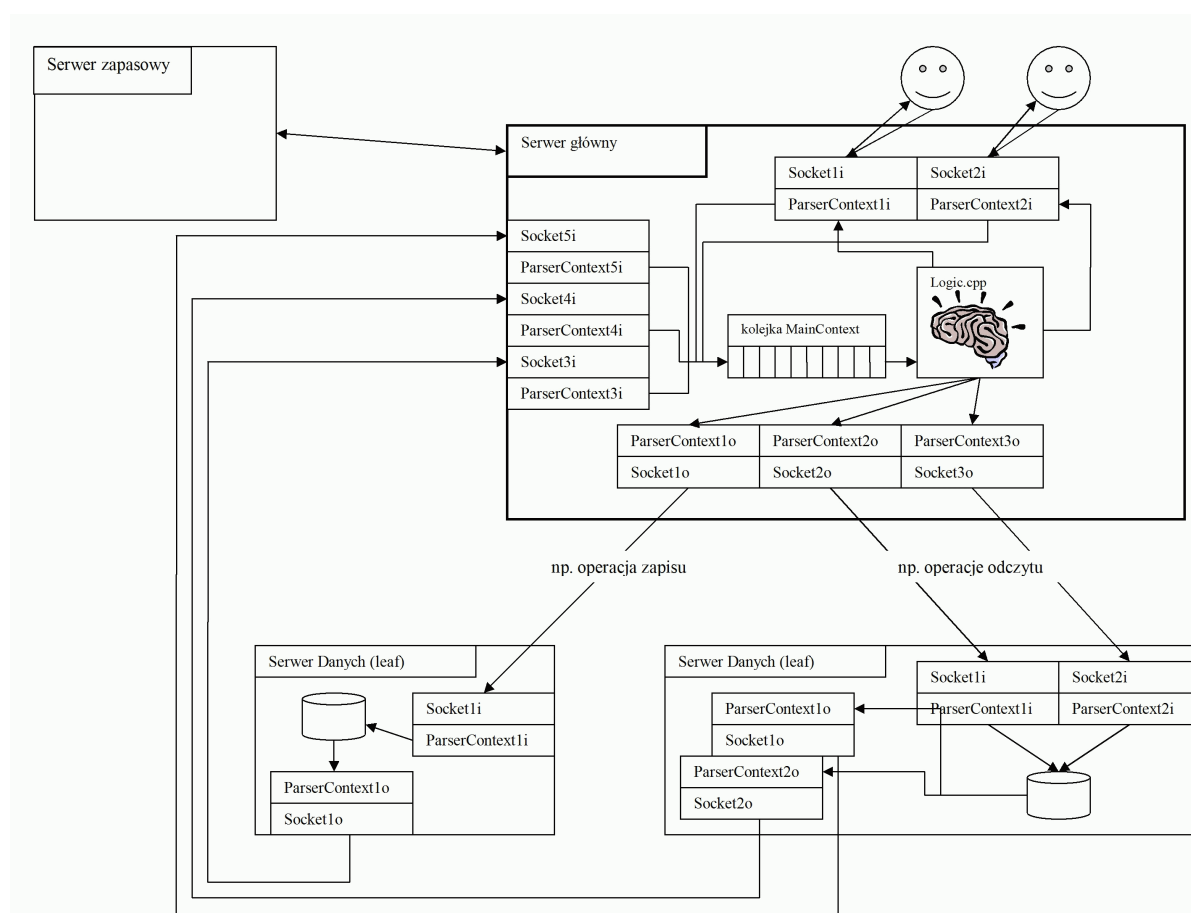
Faza II

2.1 Komentarz do opracowania etapu II

Jak to zostało zaznaczone wcześniej grupa podczas prac nad drugim etapem projektu zdecydowała się zmienić koncepcję jego wykonania na scentralizowaną. Algorytm pierścieniowy okazał się dużo bardziej skomplikowany w implementacji. Wiązało się to ze stratą czasu zainwestowanego w tworzenie pierwotnej koncepcji i nie pozwoliło na rozwinięcie działań implementacyjnych.

2.2 Architektura

Projekt planujemy zrealizować bazując na przetwarzaniu zdarzeń. Serwer będzie wielowątkowy.



2.3 Budowa

Najwyżej w hierarchii stoi warstwa logiki, która komunikuje się z warstwą parsera, a ta odpowiedzialna jest za odpowiednie sterowanie obsługą sieci.

W rozbiciu na klasy:

Obsługa sieci. Warstwa odpowiada za komunikację sieciową z innymi węzłami. Tłumaczy strumień TCP na struktury danych (klasy), które są analizowane przez warstwę parsującą. Klasa Socket może funkcjonować 2 kierunkowo. W przypadku inicjacji połączenia wychodzącego instancja klasy Socket jest tworzona w odmienny sposób niż w przypadku akceptacji połączenia przychodzącego.

Pliki: Socket.h/cpp

Parser. Parser również jest 2 kierunkowy.

Z jednej strony dokonuje on parsowania danych przekazanych przez warstwę sieciową (metoda `parse(Socket)`). Wtedy po przeanalizowaniu danych uzyskanych od warstwy sieciowej, i stwierdzeniu ich poprawności, wkłada odpowiednie zdarzenie z rodzaju `NetEvent` do kolejki komunikatów warstwy logiki, która (w innym wątku) zajmie się jego obsługą.

Symetrycznie, w przypadku gdy logika wyraża chęć wysłania pewnych danych do danego węzła; tworzy ona Parser, informując go do jakiego węzła należy wysłać dane. Następnie kolejka zdarzeń tego parsera jest napełniana zdarzeniem odpowiadającym danemu rodzajowi komunikatu, jaki chcemy przesłać, i parser jest uruchamiany w nowym wątku. Parser następnie autonomicznie zajmuje się realizacją przekazanego zlecenia.

Logika. Kontroluje wszystkie połączenia, podejmuje decyzje o wysłaniu wiadomości do innych węzłów. Komunikuje się ze światem zewnętrznym za pomocą parsera. Implementacja: główna klasa MainContext, która jest globalnym interfejsem danych dla zdarzeń przychodzących do logiki.

System jest wielowarstwowy. Wyodrębniono warstwę logiki, w której są przetwarzane zdarzenia najwyższego poziomu. Zdarzenia te wpływają na ogólny stan całego serwera.

2.3.1 Synchronizacja wątków

Serwer bazy danych jest aplikacją wielowątkową (wątki POSIX). Klasa Thread (Thread.h/cpp) tworzy interfejs C++ dla wywołań pthreads. Wątki komunikują się wyłącznie poprzez kolejki zdarzeń (Queue.h/cpp).

Kolejki te są w pełni zsynchronizowane poprzez zastosowanie odrębnej klasy synchronizującej, opierającej się na semaforach udostępnianych przez systemowe mechanizmy IPC. Kolejki te przechowują dowolne obiekty typu Event. Mogą to być zdarzenia adresowane do warstwy logiki (MainEvent), lub do warstwy parsera (ParserEvent).

Struktura aplikacji z punktu widzenia wielowątkowości:

- Najpierw startowany jest wątek warstwy logicznej.
- Jeden wątek przyjmuje połączenia przychodzące na ustalonym porcie.
- Każde połączenie przychodzące lub wychodzące jest obsługiwane w osobnym wątku. Wątki połączeń wychodzących tworzone są z wątku warstwy logicznej, dla połączeń przychodzących — w wątku nasłuchującym.

2.3.2 Komunikacja

Cała komunikacja pomiędzy różnymi wątkami opiera się na wysyłanych i odbieranych zdarzeniach. Wykorzystujemy w tym celu kolejki — klasa Queue.

Każde zdarzenie obsługiwane jest przez odpowiednią warstwę aplikacji i we właściwym kontekście (warstwa może różnie obsługiwać zdarzenia tego samego typu, zależnie od wybranych kryteriów — adres nadawcy, uzgodnione rozszerzenia protokołu, etc).

Klasy *Context zawierają interfejs potrzebny do obsłużenia wszystkich przychodzących zdarzeń. Rozbudowany interfejs kontekstu potrzebny jest w sytuacji kiedy następuje dużo interakcji pomiędzy przychodzącymi zdarzeniami - tak jest w przypadku klasy MainContext reprezentującej warstwę logiczną. Komunikacja pomiędzy klasami/obiektami w ramach jednego wątku odbywa się poprzez ustalone interfejsy (patrz: pliki nagłówkowe) - nie ma ryzyka związanego z dostępem przez wiele wątków do jednego obiektu, ponieważ każdy z nich jest wykorzystywany w danej chwili przez jeden tylko wątek.

2.4 Realizacja

Schemat działania. Z wizją implementacyjną.

2.4.1 Najwyższy poziom

1. Init.cpp

- Parsowanie konfiguracji z pliku do odpowiedniej klasy (Settings),
- Uruchomienie wątku klasy Logic (pkt. 2)
- Zapętlenie się, zapisując w pseudo-kodzie:

```
while(true){ fd = accept(); new Socket(fd); }
```

2. Logic.cpp

```
while(true){  
    MainEvent e = MainContext::queue.get();  
    e.process(*this);  
}
```

3. Socket.cpp

- W połączeń zakresie przychodzących: Nowy Socket(fd) tworzony jest tworzony w momencie pojawienia się nowego połączenia, zapewnia on komunikację z danym węzłem. Udostępnia swoją metodę recv. Wątek ten tworzy własny, nowy, ParserContext (przekazując aktualny Socket jako argument konstruktora), który gdy zostanie uruchomiony (metoda parse) prosi swoją klasę Socket o dane i dokonuje ich parsowania. Gdy parsowanie zakończy się sukcesem odpowiednie zdarzenie jest wkładane do kolejki warstwy logiki, i parser kończy swoją działalność.
- W zakresie połączeń wychodzących: Gdy warstwa logiki, w drodze wykonania odpowiedniej metody parse (dla danego zdarzenia), uzna konieczność utworzenia połączenia wychodzącego (a zatem jego inicjacji w komunikacji z drugą stroną) dojdzie do utworzenia nowego ParserContext. Kolejka zdarzeń tego (konkretnego) ParserContext'u zostajnie napełniona odpowiednim zdarzeniem (które jest zdarzeniem wychodzącym, tj. będzie przetwarzane (metoda process) w kontekście parsera), a następnie ten ParserContext zostanie odłączony (w nowy wątek) metodą run(). W swojej pętli głównej, wątek ten, pobierze ze swojej kolejki zdarzenie, które zostanie poddane przetwarzaniu, i utworzy instancję klasy Socket(NodeInfo). (Klasa NodeInfo jest kontenerem na adres IP i port.) Tak powstały Socket będzie, w wyniku parsowania komunikatu od logiki, poddawany kolejnym wywołaniom send, które będą przesyłały odpowiednie dane do drugiego węzła.
- W przypadku powstania błędu podczas transmisji, zostanie zwrócony wyjątek. Jego obsługa jest w gestii wyższej warstwy. Można ponownie spróbować transmisję, bądź też usunąć aktualną klasę, i czekać (dla połączeń przychodzących) lub utworzyć nowe (np. do innego węzła) połączenie.

4. Parser.cpp Klasa ta jest odpowiedzialna za, w metaforycznym sensie, na zamianie tekstu i woli, na zdarzenia. Parsowanie. Czyli odczytanie co przyszło w pakiecie od klienta/serwera, i wrzucenie odpowiedniego zlecenia do wyższej warstwy. (globalna kolejka) `MainContext::queue.put(*new TypZdarzenia(*this));`

Warto w tym miejscu zauważyć, że klasa `Socket` umożliwia komunikację zwrotną tym samym kanałem komunikacyjnym. Przykładowo po otrzymaniu od klienta komunikatu z prośbą o podłączenie do sieci, zostanie w ramach tego samego połączenia odesłana odpowiedź. Podobnie wygląda obsługa ping między serwerami. W ogólności jednak połączenia TCP/IP są tworzone na potrzeby każdego komunikatu niezależnie, przynajmniej w tej wersji protokołu.

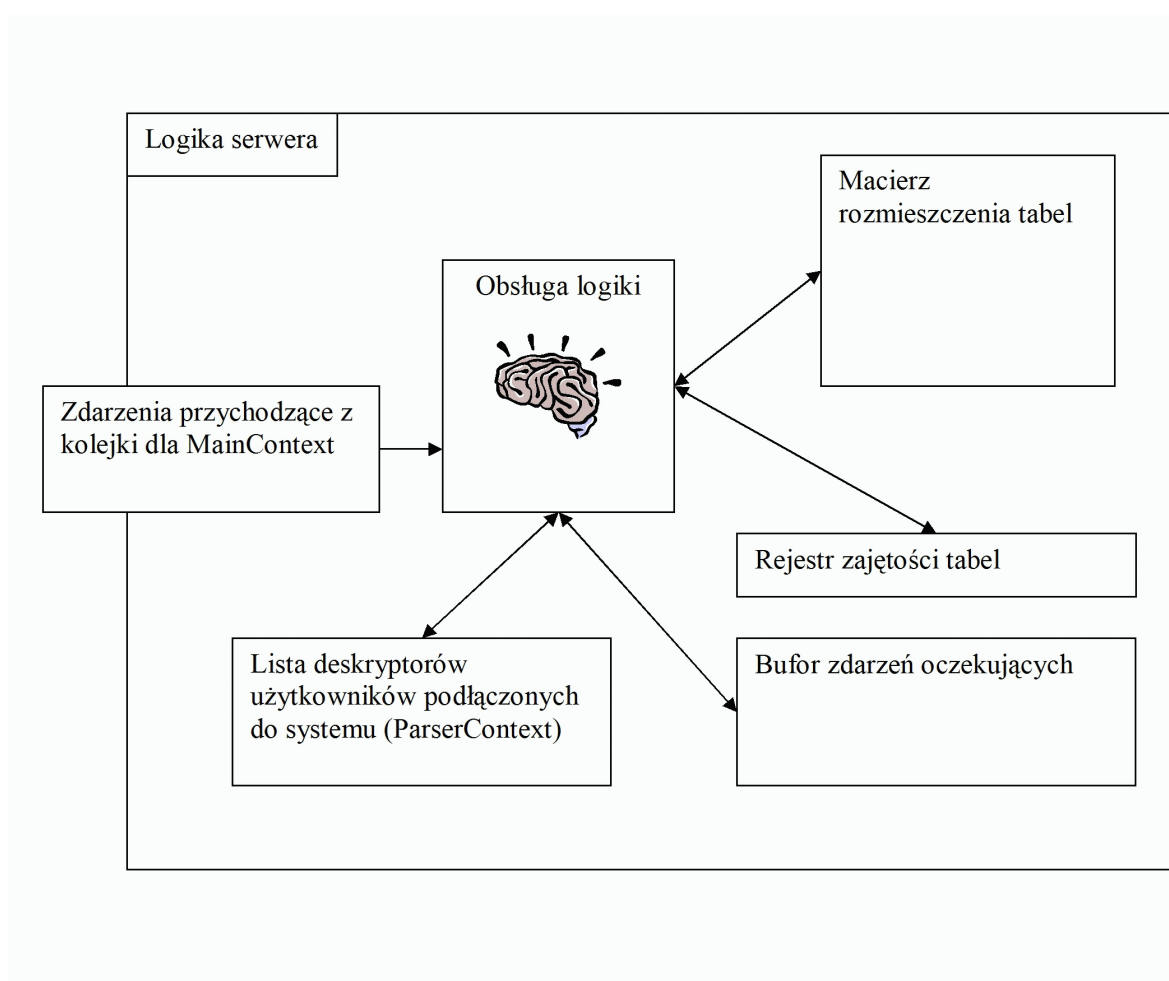
Ponadto powstało kilka klas o pomocniczym charakterze.

1. Kolejka zsynchronizowana,
2. Opakowanie wątek-klasa,
3. `NodeInfo` — przechowywanie i translacja adresów IP i portów na/z string'i, struct `in_addr`'y, itp.

2.5 Logika serwera

Warto przyjąć się bliżej organizacji logiki wewnątrz serwera głównego:

2.5.1 Schemat logiki serwera



Powyższy rysunek przedstawia struktury danych obecne w warstwie logicznej realizowanego rozwiązania. Na szczególną uwagę zasługuje lista obiektów klasy ParserContext, która pozwala serwerowi odesłać dane do odpowiedniego klienta. Mechanizm ten funkcjonuje w następujący sposób:

1. Do każdego nowopodłączonego klienta przypisywany jest krótki deskryptor, który będzie wędrował ze zleceniem złożonym przez klienta przez cały system. Natomiast ParserContext użytkownika jest składowany w wyżej wspomnianej strukturze
2. W momencie gdy do serwera przychodzi pakiet zleający odesłanie odpowiedzi do klienty znajduje odpowiedni ParserContext na podstawie deskryptora zawartego w pakiecie

Podsumowując - mechanizm ten działa podobnie do szatni z numerkami

2.6 Mechanizmy blokowania

W celu zapewnienia spójności rozproszonym środowisku danych konieczne jest wprowadzenie odpowiednich, sieciowych mechanizmów synchronizacji. Z racji wybrania rozwiązania z wyróżnionym węzłem nadrzędnym naturalne wydaje się uczynienie go odpowiedzialnym za ich realizację.

2.6.1 Umieszczenie i struktury danych

Warstwą serwera, która ma największą kontrolę nad wszystkimi operacjami podejmowanymi przez klienty i serwery oraz dysponuje największą ilością spójnych informacji na temat aktualnie podejmowanych działań jest warstwa logiczna. Sensowne wydaje się umiejscowienie mechanizmów synchronizacyjnych na poziomie tej warstwy.

W celu przechowywania informacji o używanych zasobach warstwa logiczna przechowuje **Rejestr zajętości tabel**. Jest to de facto rejestr semaforów przechowujących informacje o ilości i rodzaju procesów ubiegających się o zasoby poszczególnych tabel. Zakłada się dostęp nieograniczonej liczby czytelników do każdej z tabel oraz dopuszczenie tylko jednego piszącego dla wszystkich tabel (wszystkie repliki są w takiej sytuacji uzupełniane równolegle). Ponieważ istnieje realna groźba niedopuszczenia procesów piszących do zasobów bazy danych, procesy piszące i modyfikujące tabele mają wyższy priorytet.

Konieczne jest jednak wprowadzenie dodatkowego warunku, że jeśli w buforze zdarzeń oczekujących na realizację znajduje się więcej niż X (X np. równe 50) procesów żądających odczytów - bufor ten jest opróżniany. Zapytania wymagające pisania/modyfikacji bazy trafiają do bufora z powrotem, natomiast zapytania zawierające odczyty są natychmiast realizowane.

Wspomniany wyżej **Bufor zdarzeń oczekujących** jest kolejną strukturą wymaganą do realizacji blokad z wykorzystaniem warstwy logicznej. Trafiają tam zdarzenia które po pobraniu z kolejki zdarzeń nie mogą być od razu zrealizowane. Do realizacji bufora używamy standardowych bibliotek C++ - Standard Template Library (STL). Oprócz dość ewidentnych zalet wynikających z realizowanego w tej bibliotece dynamicznego zarządzania pamięcią możliwe jest swobodny dostęp i wyszukiwanie w takiej strukturze. Pozwala to odpowiednio reagować na zdarzenia zawierające informację o odblokowaniu zasobu.

2.6.2 Przykład przebiegu obsługi klientów

Przykładowy przebieg obsługi zapytań, w których biorą udział mechanizmy blokujące jest następujący:

1. Do logiki serwera głównego zgłasza się równolegle dwóch klientów z żądaniem odczytu z tabeli T_1
2. Serwer główny deleguje zapytania do serwerów danych i inkrementuje o 2 znacznik zajętości dla procesów czytających tabeli T_1
3. Do logiki serwera głównego zgłasza się klient z żądaniem modyfikacji/zapisu tabeli T_1

4. Serwer główny wykrywa zajętość tabeli T1 umieszcza zdarzenie T1 w buforze zdarzeń oczekujących. Ustawiany jest znacznik obecności zapytania z żądaniem pisania/modyfikacji w buforze w celu filtrowania kolejnych zdarzeń przychodzących zawierających żądania odczytu z T1 (zabezpieczenie przed zagłodzeniem w zlecenia pisania w buforze)
5. Serwer danych kończy obsługę jednego z klientów czytających. Przesyła żądane przez niego dane. Na końcu przesyła do serwera zdarzenie z informacją o możliwości zwolnienia blokady
6. Z kolejki zdarzeń serwera głównego odbierane jest zdarzenie zawierające informację o możliwości zwolnienia blokady dla tabeli T1. Semafor dla T1 jest dekrementowany o 1.
7. Do logiki serwera głównego zgłasza się zlecenie od nowego klienta zawierające żądanie odczytu z tabeli T1. Ponieważ serwer ma ustawiony znacznik obecności w buforze zlecenia pisania/modyfikacji tabeli T1 zdarzenie trafia do bufora zdarzeń oczekujących.
8. Serwer danych kończy obsługę drugiego z klientów czytających. Przesyła żądane przez niego dane. Na końcu przesyła do serwera zdarzenie z informacją o możliwości zwolnienia blokady dla T1.
9. Logika serwera odbiera zdarzenie ze zleceniem zdjęcia blokady - dekrementuje semafor dla T1 o 1, po czym zauważa że jego wartość wynosi 0. Ponieważ jest ustawiony znacznik istnienia w buforze zdarzeń oczekujących zdarzenia ze zleceniem pisania/modyfikacji zostaje ono odnalezione w buforze i wysłane do realizacji. Jednocześnie ustawiany jest semafor pisania/modyfikacji w Rejestrze zajętości tabel. Semafor ten jest inkrementowany o tyle ile jest replik danej tabeli w bazie danych.
10. Po zakończeniu zapisu do wszystkich replik przysyłane są przez liście zdarzenia ze zleceniem zwolnienia blokady.
11. Gdy blokada wynosi 0 logika serwera głównego odnajduje zdarzenie ze zleceniem odczytu przysłane przez nowego klienta. Zdarzenie to jest przesyłane do realizacji

2.7 Replikacje i reakcja na rozłączenia serwerów

Wartość wymaganej ilości replik każdej tabeli istniejącej w systemie przechowywana jest w klasie Settings i wpisywana tam z pliku konfiguracyjnego uruchamianego razem z całą aplikacją. Ze względu jednak na potencjalną możliwość wystąpienia awarii faktyczny rejestr replik jest przechowywany również w logice serwera głównego. W tym samym miejscu przechowywana jest informacja o rozmieszczeniu poszczególnych replik. Istotne jest że informacje te są na bieżąco aktualizowane i zapisywane również na serwerze zapasowym. Pozwala to na szybki powrót bazy do normalnego funkcjonowania po ewentualnej awarii serwera głównego.

Klasa Settings zawiera również informację o minimalnej ilości węzłów na której ma funkcjonować system (poniżej jakiej ilości węzłów należy uruchomić procedury przeszukiwania sieci w poszukiwaniu nowych węzłów).

2.7.1 Replikacje podczas normalnego funkcjonowania

Za odpowiednią ilość replik podczas normalnego, bezawaryjnego funkcjonowania bazy odpowiedzialne są procedury służące do tworzenia tabel na zlecenie klienta lub podczas ładowania tabeli z pliku. Podczas takiej procedury:

- Dodawana jest pozycja do rejestru wszystkich tabel w logice systemu (przechowującego semafora oraz informację o ilości replik danej tabeli)
- Losowane są węzły, na których zostaną ulokowane poszczególne repliki. Do wybranych węzłów wysyłane jest zlecenie utworzenia danej tabeli
- W logice serwera głównego zapamiętywana jest informacja o powstałym właśnie rozmieszczeniu poszczególnych replik
- Informacje o rozmieszczeniu przekazywane są do logiki serwera zapasowego za pomocą protokołu - szczegóły w części dokumentacji poświęconej komunikacji

2.7.2 Sytuacje awaryjne - awaria liścia

Jak to zostało opisane w części dokumentacji poświęconej strukturze projektu podczas awarii łączy klasa Socket zgłasza wyjątek do przypisanej do niej klasy ParserContext. Zakładamy że w reakcji na takie zdarzenie klasa parser context dokona kilkukrotnej próby ponownego nawiązania połączenia. Ponieważ jest to klasa wyposażona w kolejkę zdarzeń będzie ona buforować nadchodzące zdarzenia do momentu uzyskania połączenia. Będzie to dobrze funkcjonować w przypadku, gdy dany węzeł został odłączony od sieci tylko na chwilę.

W sytuacji jednak, gdy dany węzeł został odłączony od sieci permanentnie a bufor klasy ParserContext ulegnie przepełnieniu wstawi ona odpowiednie zdarzenie do kolejki logiki informując ją o stracie węzła. Wraz z tą informacją zostaną przekazane wszystkie istotne zdarzenia zgromadzone w buforze dla tego węzła. Gdy po takim zdarzeniu węzeł ponownie podłączy się do sieci nie będzie mógł pełnić tej funkcji co poprzednio. Konieczne będzie pełne odtworzenie danych, które się na nim znajdowały i potraktowanie go jako nowego węzła dołączającego się do sieci.

Logika po otrzymaniu informacji o stracie węzła i porzeanalizowaniu danych o rozmieszczeniu tabel może podjąć następujące działania:

- Gdy wykryje że liczba replik któreś ze straconych tabel spadła poniżej dopuszczalnego minimum wydaje polecenie liściowi, który zawiera daną tabelę z poleceniem przesłania do serwera głównego repliki. Replika ta jest przekazywana na inny niezajęty jeszcze przez tą tabelę węzeł
- Gdy wykryje że liczba węzłów na których powinna być uruchomiona usługa spadła poniżej dopuszczalnego minimum uruchamia procedurę przeszukiwania sieci w poszukiwaniu nowych węzłów - uruchamia skrypt ssh analogiczny do tego, który służy do rozruchu systemu. Jeśli zostanie znaleziony odpowiedni węzeł sam zgłosi się do serwera głównego poprzez odpowiedni port

2.7.3 Sytuacje awaryjne - awaria serwera głównego

W sytuacji, gdy dojdzie do awarii serwera głównego następuje przełączenie na serwer zapasowy. Informacje o zdarzeniach przechowywanych w kolejce zdarzeń serwera głównego przepadają. Ponieważ tam znajdują się wszystkie informacje o blokadach nie ma groźby sytuacji że na tabelach pozostaną niezdyktowane blokady co uniemożliwi korzystanie z nich. Aplikacja kliencka wykrywa awarię serwera głównego, łączy się do serwera zapasowego i informuje korzystającego zeń klienta o ewentualnym niepowodzeniu zleconej przez niego transakcji.

Możliwe usprawnienia:

W przypadku gdyby awarii uległ zarówno serwer główny jak i serwer zapasowy można rozważyć uruchomienie algorytmów elekcji i procedur odtwarzania informacji ogólnych na podstawie informacji zawartej w liściach

2.8 Komunikacja

W celu zapewnienia odpowiednich mechanizmów komunikacji konieczne jest wyspecyfikowanie rodzajów pakietów jakie będą przesyłane pomiędzy poszczególnymi elementami systemu.

2.8.1 Pakiety przychodzące do serwera głównego

Do pakietów przychodzących do serwera głównego i trafiających w kolejkę logiki można zaliczyć:

- Opakowane przez aplikację kliencką zapytania od klientów.
- Otrzymane od serwera danych zlecenia przesłania odpowiedzi do klienta
- Otrzymane od serwera danych zlecenia przesłania danych do innego serwera danych (np. w sytuacji awaryjnej)
- Informacje o zwolnieniach istniejących blokad
- Informacje o stracie węzła
- Informacje o strukturze sieci odbierane w sytuacji gdy dany serwer jest serwerem zapasowym

2.8.2 Pakiety wychodzące od serwera głównego

Serwer główny może nadawać następujące rodzaje pakietów:

- Sformatowane zapytania do serwerów danych
- Spakowane tabele w celu odtworzenia z nich repliki
- Informacje o strukturze sieci przesyłane do serwera zapasowego

2.8.3 Opakowanie XML

Jak to zostało zaznaczone wcześniej każdemu z wyżej wymienionych rodzajów pakietów zostanie przyporządkowany odpowiedni XML Header który będzie parsowany przez klasy ParserContext. Specyfikacja XML znajdzie się w dokumentacji końcowej projektu.

2.9 Aplikacja kliencka

Program kliencki jest prosty i efektywny w swojej budowie. Odpowiada on za przekazywanie zleceń od użytkownika do serwerów, oraz zapewnia stałe połączenie z siecią.

Służy on także do odbierania odpowiedzi od serwerów, które przyjdą na zapytania klienta.

klasa NodeInfo — opakowanie na informację o numerze IP i portu serwera. klasa Net — przechowuje dane o potencjalnych serwerach (zbiór NodeInfo) i aktualnym serwerze.

Przebieg wykonania w kliencie:

1. Wczytać konfigurację z pliku do klasy Net.
2. Połączyć się z pierwszym serwerem, wysłać pakiet typu HELLO.
3. Odebrać potwierdzenie od serwera, wraz z informacją o ew. zapasowych serwerach (pakiet WELCOME).
4. Odczytać komendę od użytkownika. Wstępnie zwalidować jej poprawność syntaktyczną.
5. Opakować komendę użytkownika w pakiet danych (typu USER_REQ) i wysłać do serwera przez gniazdo.
6. Jeśli powyższe się nie udało (gniazdo zwróciło błąd, serwer nie odesłał odpowiedzi) należy spróbować połączenia z innym serwerem, i ponowić operację. Serwer, który nie funkcjonuje oznaczyć jako zły w klasie Net.
7. Czekać na odpowiedź serwera (z danymi, lub przekierowanie na inny serwer) przez ustalony czas.
8. Jeśli udało się odebrać odpowiedź (pętla czekająca na select()), należy pakiet ten wyświetlić, po ew. konwersji na postać czytelną dla użytkownika. Jeśli komunikat miał charakter sterujący, program kliencki wykona przewidzianą w nim akcję (jak zmiana serwera).

2.10 Silnik Bazy Danych

Wykorzystane w projekcie rozwiązanie jest rozwiązaniem autorskim. Zgodnie z wymaganiami dane przechowywane w bazie znajdują się w całości w pamięci operacyjnej. Wymagana do tego celu pamięć jest alokowana i zwalniana dynamicznie w zależności od aktualnych potrzeb.

2.10.1 Struktury danych

W celu zorganizowania i uporządkowania przestrzeni pamięci przeznaczonej do przechowywania informacji wprowadzone zostały następujące struktury:

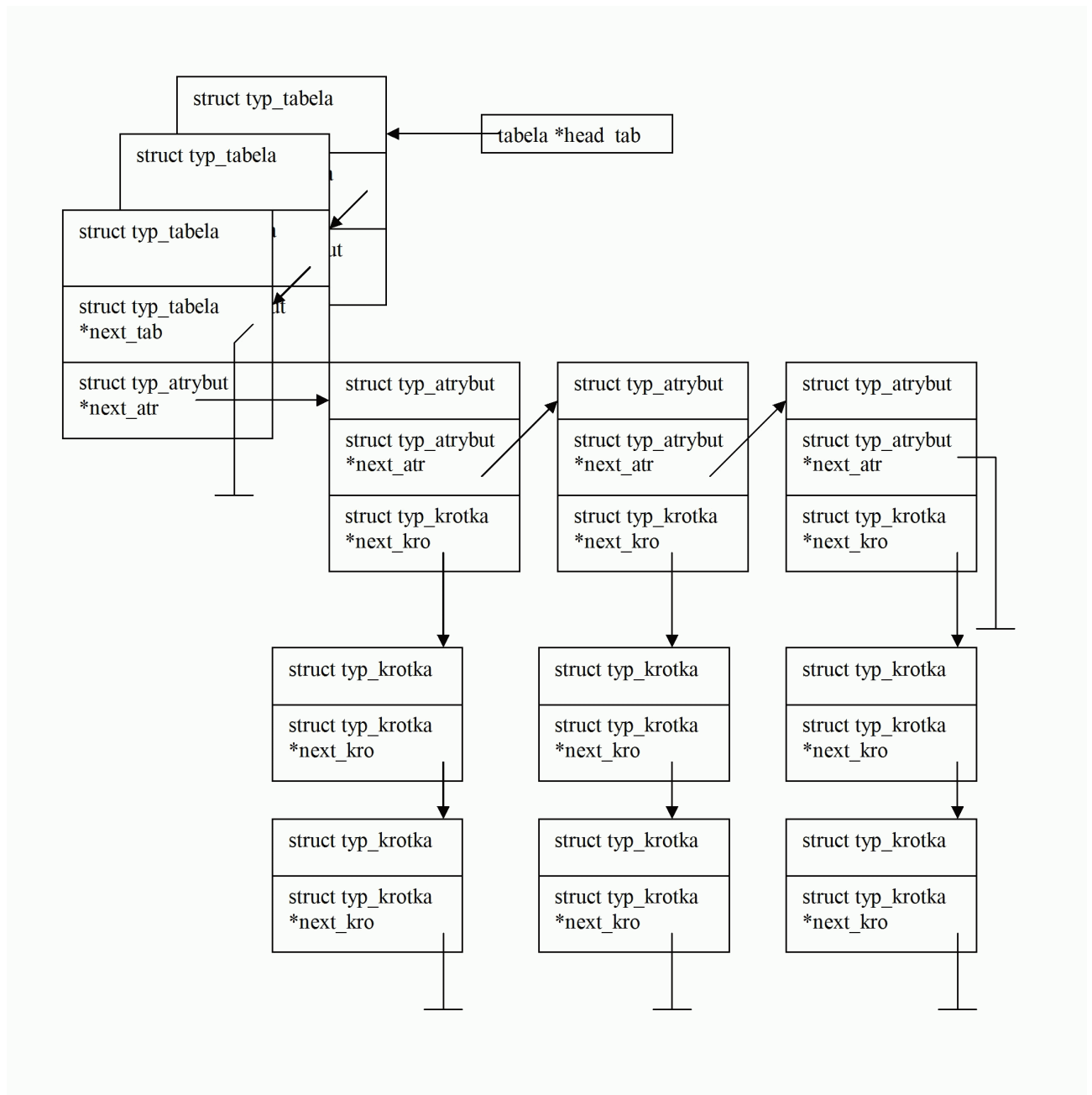
```
struct typ_krotka {  
    char wartosc[30];  
    struct typ_krotka *next_kro;  
};
```

```
struct typ_atrybut {  
    char atr_nazwa[30];  
    struct typ_atrybut *next_atr;  
    struct typ_krotka *next_kro;  
};
```

```
struct typ_tabela {  
    char tab_nazwa[30];  
    struct typ_tabela *next_tab;  
    struct typ_atrybut *next_atr;  
};
```

```
typedef struct typ_tabela tabela;  
typedef struct typ_atrybut atrybut;  
typedef struct typ_krotka krotka;
```

Sposób uporządkowania struktur przedstawia rysunek:



Zorganizowanie struktur w wyżej przedstawiony sposób umożliwia uporządkowany dostęp do danych przechowywanych w bazie oraz manipulacje nimi na satysfakcjonującym poziomie.

2.10.2 Algorytmy

Podczas odczytów, w najgorszym przypadku, przeglądana jest lista tabel, lista atrybutów znalezionej tabeli i lista wszystkich rekordów. Również podczas dokonywania zapisu odpowiednia

lista jest w całości przeglądana w celu znalezienia jej końca i umieszczenia tam nowotworzonego elementu. Nie są to rozwiązania optymalne wydajnościowo. Istnieje zatem duże pole do optymalizacji na następnym etapie projektowania.

Możliwe ulepszenia:

- Wprowadzenie dodatkowych struktur, w których przechowywane by były wskaźniki do ostatnich elementów w danej liście. Umożliwiłoby to dodawanie nowego elementu do takiej listy bez potrzeby przeszukiwania całej listy.
- Modyfikacja listy wszystkich tabel na podstawie gromadzonych statystyk użycia. Przesuwanie bliżej początkowego wskaźnika tych, które są częściej używane

2.11 Język zapytań

Specjalnie na potrzeby projektu, w celu umożliwienia wykonywania na danych operacji DML (Data Management) oraz wykonywania operacji DDL (Data Definition) został skonstruowany prosty język zapytań. Obejmuje on następujące operacje:

- Tworzenie nowej tabeli:

```
CREATE nazwa_tabeli (atrybut_1, atrybut_2...);
```

- Dodawanie wiersza do tabeli:

```
INSERT nazwa_tabeli (wart_atr_1, wart_atr_2...);
```

- Wyszukiwanie rekordów pasujących do pewnego wzorca:

```
SELECT kolumna_1, kolumna_2... FROM nazwa_tabeli WHERE  
atrybut_1=wartosc_1, atrybut_2=wartosc_2...;
```

- Wypisanie całej zawartości tabeli:

```
SELECT * FROM nazwa_tabeli WHERE **;
```

- Aktualizowanie zawartości rekordów pasujących do pewnego wzorca:

```
UPDATE nazwa_tabeli SET wartosc_atr1, wartosc_atr2 WHERE  
atrybut_1=wartosc_1, atrybut_2=wartosc_2...;
```

- Dodawanie nowego rekordu do tabeli

```
DELETE nazwa_tabeli WHERE  
artybut_1=wartosc_1, atrybut_2=wartosc_2...;
```

- Usuwanie całej tabeli

```
DROP nazwa_tabeli;
```

2.11.1 Klasa Baza

Obudowanie przedstawionego wyżej mechanizmu klasą Baza ma za zadanie wyodrębnienie interfejsu przeznaczonego do komunikacji ze obiektem bazy danych, który znajduje się na każdym węźle podrzędnym (liściu) rozproszonego serwera.

Metody udostępniane przez klasę Baza:

- bool load_content(String *sciezka_dost_do_pliku*);
- bool block_table(String *nazwa_tabeli*);
- String doQuery(String query);

2.12 Harmonogram Prac

Prace nad projektem wykonano wieloetapowo.

2.12.1 Etap I

11 marca

- wymiana kontaktów,
- wstępne zapoznanie się z dokumentacjami z poprzednich lat.

18 marca

- ustalenie podstawowych założeń projektu (uruchamianie poprzez ssh, komunikacja poprzez sockety).

1 kwietnia

- opracowanie pierwszej koncepcji projektu — serwer główny + serwery z danymi,
- po konsultacji z prowadzącym, zmiana pomysłu na sieć serwerów równorzędnych z pierścieniem z krążącym znacznikiem,

- podział pracy przy dokumentacji:
 - Paweł Gilewski — replikacja
 - Michał Przyłuski — aplikacja kliencka i svn
 - Paweł Czernik — uruchamianie i struktura token-ring (awaria, dołączanie węzłów)
 - Szymon Janikowski — mechanizm blokad i zachowanie spójności w bazie

2.12.2 Etap II

21 kwietnia

- Wyróżnienie czynności wstępnych jakich należy się podjąć, w celu przygotowania gruntu pod właściwą implementację
- Przydzielenie wyróżnionych działań do poszczególnych członków zespołu:
 - Paweł Czernik — Oprogramowanie realizujące uruchamianie systemu za pomocą zdalnego logowania
 - Paweł Gilewski — Zestawienie połączeń pomiędzy poszczególnymi komputerami klastra
 - Michał Przyłuski — Prototyp aplikacji klienckiej. Problem podłączania użytkownika do klastra.
 - Szymon Janikowski — Zaprojektowanie protokołów komunikacyjnych dla poszczególnych warstw systemu tj. pseudo języka zapytań SQL i protokołu komunikacji wewnętrznej.

15 maja

- Omówienie koncepcji wielowątkowego serwera zakładającej podział na warstwy komunikujące się ze sobą za pomocą kolejek. Wyprecyzowanie klas zajmujących się komunikacją na każdym z wyróżnionych poziomów.
- Przedstawienie skryptu mającego na celu delikatne uruchamianie aplikacji na poszczególnych węzłach sieci klastra. Omówienie szczegółów dopracowania tego elementu (Paweł Czernik)

22 maja

- Testowanie skryptu uruchomieniowego w laboratorium OpenSSI (Paweł Czernik)
- Testowanie nawiązywania połączeń poprzez gniazda BSD (Paweł Gilewski)
- Dyskusa na temat koncepcji projektu - postawienie pod znakiem zapytania słuszności koncepcji z równorzędnymi węzłami synchronizowanymi za pomocą Token Ring. Przedstawienie wad i zalet obu rozwiązań.
- Podział zadań zasadniczej fazy implementacyjnej:

- Paweł Czernik — wykonanie parsera zapytań pseudo-języka SQL i silnika Bazy Danych
- Michał Przyłuski — dopracowanie koncepcji klas komunikujących się za pomocą kolejek, mechanizmów komunikacji pomiędzy wątkami oraz zadań synchronizacyjnych na poziomie pojedynczego serwera
- Paweł Gilewski — realizowanie komunikacji na poziomie serwera z wykorzystaniem wielpoziomowej struktury klas
- Szymon Janikowski — analizowanie potencjalnych sposobów organizacji serwera w związku z powstałymi wątpliwościami. Testowanie powstających fragmentów aplikacji. Dokumentowanie dokonań.

29 maja

- Dokładne wyspecyfikowanie podziału serwera na warstwy. Omówienie zrealizowanych rozwiązań i wyjaśnienie sposobu organizacji, komunikacji i korzystania z hierarchii klas (Michał Przyłuski)
- Podjęcie decyzji o zmianie koncepcji i powrót do tej najbardziej pierwotnej (z której zespół zrezygnował po konsultacji z prowadzącym 1 kwietnia), czyli serwera głównego (+ ewentualny serwer zapasowy) i serwery zawierające dane.

Rozdział 3

Etap końcowy

3.1 Komentarz do opracowania etapu III

Z racji tego że prace w momencie etapu II nie były z racji wcześniej podanych przyczyn odpowiednio zaawansowane, a tydzień na który musiałoby przypaść nadrobienie zaległości okazał się być najgorętszym czasem naszej dotychczasowej kariery naukowej, prace projektowe ograniczyły się do nawiązania kontaktu klienta z serwerem w ramach zaprojektowanej na etapie II architektury klas. Poza tym wykonano szereg prób środowiska testowego (którego to opis znajduje się w niniejszej części dokumentacji) oraz przygotowano silnik bazy danych do włączenia w rozwiązanie rozproszone poprzez odpowiednie opakowanie go w klasę.

3.2 Opis realizacji komunikacji klient - serwer

1. Nowe połączenie przychodzi
2. Socket::main tworzy ParserContext do tego połączenia rejestruje się jemu, i zmusza do przeparsowania bieżącego socket'a
3. Parser wchodzi w ::parse
4. Parser pobiera XMLHeader (87), ze swojego socketa
5. Parser wykrywa typ XML'a, wiadomość rozpoznawana teraz to: Przyjęty format XML'a jest następujący: <toad v="1"> <msg type="query">TRESC ZAPYTANIA</msg> </toad> Jak program stwierdzi, że otrzymał takie coś wywoła funkcję, która przeparsuje argument(y)
6. Funkcja ParserContext::analyzerPing(xmlXPathContextPtr) zbiera /TRESC ZAPYTANIA/ i tworzy obiekt typu NetQuery(...) i wstawia go do kolejki.
7. Teraz, parser wraca do 163 linii, i czeka na sygnał z logiki czeka na odpowiedź. Komunikacja wygląda następująco .. klient - nowe połączenie out, serwer odbiera połączenie przychodzące klient coś wysłał serwer to odbiera, serwer coś odsyła klient to odbiera koniec ..

8. Logika przetwarza otrzymany komunikat wywołując `NetQuery::process(..)` W wyniku działania tej metody, zwracany jest komunikat `OutReplyQuery()`. metoda `reply()`. A `reply` wkłada to zdarzenie do kolejki obecnego `ParserContext'u`.
9. `ParserContext` znowu się budzi
10. Wywołuje `OutReplyQuery::process`, który de facto wywołuje dla swojego `socket'a` `sendHeader` z tekstem który przekazała nam logika
11. Koniec. 164/`Parser.cpp` wraca, zamykane są połączenia

3.3 Klaster testowy

W skład klastra wchodzi 5 wirtualnych maszyn z systemem operacyjnym Fedora 8. Katalog `/home` jest współdzielony imitując warunki panujące w laboratorium 25.

Aby połączyć się z klastrem należy:

1. Połączyć się przez ssh na `tin.myszak-szyszak.pl:5022`. Powoduje to połączenie z maszyną `rs01`.

Struktura klastra:

- `rs01` – 192.168.0.21 (Węzeł główny - tutaj znajduje się `/home`)
 - `rs02` – 192.168.0.22
 - `rs03` – 192.168.0.23
 - `rs04` – 192.168.0.24
 - `rs05` – 192.168.0.25
2. Jak zwykle poleca się korzystanie z klucza publicznego, aby wszystko samo się autoryzowało.

3.4 Harmonogram Prac

Prace nad projektem wykonano wieloetapowo.

3.4.1 Etap I

11 marca

- wymiana kontaktów,
- wstępne zapoznanie się z dokumentacjami z poprzednich lat.

18 marca

- ustalenie podstawowych założeń projektu (uruchamianie poprzez ssh, komunikacja poprzez sockety).

1 kwietnia

- opracowanie pierwszej koncepcji projektu — serwer główny + serwery z danymi,
- po konsultacji z prowadzącym, zmiana pomysłu na sieć serwerów równorzędnych z pierścieniem z krążącym znacznikiem,
- podział pracy przy dokumentacji:
 - Paweł Gilewski — replikacja
 - Michał Przyłuski — aplikacja kliencka i svn
 - Paweł Czernik — uruchamianie i struktura token-ring (awaria, dołączanie węzłów)
 - Szymon Janikowski — mechanizm blokad i zachowanie spójności w bazie

3.4.2 Etap II**21 kwietnia**

- Wyróżnienie czynności wstępnych jakich należy się podjąć, w celu przygotowania gruntu pod właściwą implementację
- Przydzielenie wyróżnionych działań do poszczególnych członków zespołu:
 - Paweł Czernik — Oprogramowanie realizujące uruchamianie systemu za pomocą zdalnego logowania
 - Paweł Gilewski — Zestawienie połączeń pomiędzy poszczególnymi komputerami klastra
 - Michał Przyłuski — Prototyp aplikacji klienckiej. Problem podłączania użytkownika do klastra.
 - Szymon Janikowski — Zaprojektowanie protokołów komunikacyjnych dla poszczególnych warstw systemu tj. pseudo języka zapytań SQL i protokołu komunikacji wewnętrznej.

15 maja

- Omówienie koncepcji wielowątkowego serwera zakładającej podział na warstwy komunikujące się ze sobą za pomocą kolejek. Wyprecyzowanie klas zajmujących się komunikacją na każdym z wyróżnionych poziomów.
- Przedstawienie skryptu mającego na celu delikatne uruchamianie aplikacji na poszczególnych węzłach sieci klastra. Omówienie szczegółów dopracowania tego elementu (Paweł Czernik)

22 maja

- Testowanie skryptu uruchomieniowego w laboratorium OpenSSI (Paweł Czernik)
- Testowanie nawiązywania połączeń poprzez gniazda BSD (Paweł Gilewski)
- Dyskusa na temat koncepcji projektu - postawienie pod znakiem zapytania słuszności koncepcji z równorzędnymi węzłami synchronizowanymi za pomocą Token Ring. Przedstawienie wad i zalet obu rozwiązań.
- Podział zadań zasadniczej fazy implementacyjnej:
 - Paweł Czernik — wykonanie parsera zapytań pseudo-języka SQL i silnika Bazy Danych
 - Michał Przyłuski — dopracowanie koncepcji klas komunikujących się za pomocą kolejek, mechanizmów komunikacji pomiędzy wątkami oraz zadań synchronizacyjnych na poziomie pojedynczego serwera
 - Paweł Gilewski — realizowanie komunikacji na poziomie serwera z wykorzystaniem wielpoziomowej struktury klas
 - Szymon Janikowski — analizowanie potencjalnych sposobów organizacji serwera w związku z powstałymi wątpliwościami. Testowanie powstających fragmentów aplikacji. Dokumentowanie dokonań.

29 maja

- Dokładne wyspecyfikowanie podziału serwera na warstwy. Omówienie zrealizowanych rozwiązań i wyjaśnienie sposobu organizacji, komunikacji i korzystania z hierarchii klas (Michał Przyłuski)
- Podjęcie decyzji o zmianie koncepcji i powrót do tej najbardziej pierwotnej (z której zespół zrezygnował po konsultacji z prowadzącym 1 kwietnia), czyli serwera głównego (+ ewentualny serwer zapasowy) i serwery zawierające dane.

3.4.3 Etap III

08 czerwca - Podsumowanie działań

- Skonstruowanie działającego klienta, który potrafi podłączyć się do serwera
- Próby skonstruowania prawidłowo funkcjonującego serwera
- Obudowanie klasą Baza silnika bazy danych

Bibliografia

- [1] Wymagania projektowe w postaci pliku *rso_projekt2007.txt*, wg. wersji 2007.03.11,
WEiTI PW, RSO.B, tjk
- [2] Collins-Sussman B., Fitzpatrick B. W., Pilato C. M.: *Version Control with Subversion*, rev.
2866, dostępna na <http://svnbook.red-bean.com/en/1.4/index.html>