Implementacja MySQL w środowisku klastra OpenSSI

Opracowanie dla projektu z przedmiotu RSO

Zespół RSO5

Marcin Koziej
mkoziej@elka.pw.edu.pl

Michał Skrzędziejewski
docent@gmail.com

Łukasz Andrzej Bartnik
L.Bartnik@gmail.com

Krzysztof Jamróz
K.Jamroz@elka.pw.edu.pl

Michał Podsiadłowski
budyn@o2.pl

20 czerwca 2006

Spis treści

Część I

Faza I - opracowania wstępne

Rozdział 1

Tworzenie zwykłej paczki MySQL

Łukasz Andrzej Bartnik

1.1 Informacje wstępne

Podczas tworzenia paczki skonfigurować można wiele opcji, które zostaną „zaszyte" w plikach binarnych. Konfiguracji dokonuję się przy użyciu dostarczonego wraz ze źródłami MySQL programu *configure*.

1.2 Przygotowanie źródeł

Jeśli katalog ze źródłami nie został jeszcze pobrany z repozytorium SubVersion, należy (najlepiej w swoim katalogu domowym) wydać polecenie

```
svn checkout \
   file:///home/rso/mskrzedz/rso_rep/trunk/mysql_rso
```

Utworzy ono w bieżącym katalogu podkatalog *mysql_rso* zawierający bieżący stan repozytorium.

Jeśli źródła MySQL zostały pobrane wcześniej należy wyczyścić dane z poprzednich kompilacji wydając w katalogu mysql_rso polecenie

```
make distclean
```

1.3 Konfiguracja

W katalogu ze źródłami znajduje się plik *configure*, za pomocą którego wykonywana jest konfiguracja ustawień oraz sprawdzenie poprawności środowiska, w którym ma być skompilowany MySQL.

Należy podać wartości dla poniższych opcji:

- *–prefix=PREFIX* - jest to ścieżka instalacji dla plików niezależnych od architektury sprzętowej; z tą opcją wiąże się następna:
- *–exec-prefix=EPREFIX* - ścieżka dla plików zależnych od architektury; domyślnie wartość ta sama, co PREFIX
- *–localstatedir=DIR* - pliki modyfikowalne, dla danej maszyny (pliki z bazą danch), domyślnie PREFIX/var
- *–with-innodb* - włącza kompilację silnika bazy danych InnoDB, z którego domyślnie powinna korzystać instalacja

Dodatkowo podać można następujące opcje:

- *–with-ndbcluster* - włącza rozszerzenie MySQL Cluster
- *–with-debug* - włącza debug; dostajemy tutaj „bezpieczny alokator pamięci", patrz dokumentacja MySQL **Note debugging-server::*
- *–enable-assembler*
- *–with-mysqld-ldflags=-all-static* - dwie powyższe opcje mają wpływ na szybkość pracy serwera bazy danych

1.4 Kompilacja i instalacja

Jeśli program *configure* zakończy się poprawnie, należy wykonać polecenia:

```
make
```

które skompiluje wymagane pliki i utworzy pliki wykonywalne oraz

```
make install
```

które zainstaluje MySQL w katalogu podanym w opcji *–prefix* dla configure.

Z zawartość tego katalogu można utworzyć archiwum, które będzie „paczką binarną" wymaganą jako jeden z elementów fazy 1.

1.5 Instalacja plików bazy danych

Po instalacji MySQL w katalogu docelowym przed uruchomieniem serwera konieczna jest instalacja plików bazy danych. Wykonuje się to za pomocą skryptu *mysql_install_db*, którego parametrem jest nazwa katalogu dla plików bazy danych.
Aby użyć katalogu nie znajdującego się w poddrzewie instalacji MySQL, należy podać opcję *–datadir*=[pełna ścieżka], np. /home/rso/uzytkownik/data (wcześniej mkdir /data).

1.6 Automatyzacja

W repozytorium znajduje się skrypt magetgz.sh automatyzujący wykonanie powyższych poleceń. Jako wynik pracy w katalogu z instalacją binarną powstaje plik mysql.tgz.

1.7 Konfiguracja czasu wykonania

Wszystkie opcje MySQL można skonfigurować w momencie uruchamiania serwera. Służą do tego opcje linii komend, których odpowiedniki można zapisać w pliku konfiguracyjnym. Dla opcji postaci *–opcja=wartość* w pliku konfiguracyjnym powinna znaleźć się linia *opcja=wartość*.
Poniżej zamieszczono przykładowy plik konfiguracyjny który zmienia katalog z bazą danych

```
[client]
socket=/home/rso/lbartnik/mysql.sock

[mysqld]
datadir=/home/rso/lbartnik/data
socket=/home/rso/lbartnik/mysql.sock
```

Plik konfiguracyjny wskazuje się serwerowi MySQL poprzez opcję *–defaults-file*.

Więcej o korzystaniu plików konfiguracyjnych w punkcie 4.3.1 dokumentacji MySQL.

1.8 Problemy

Zauważyłem, że przy uruchamianiu mysqld_safe nie tworzy się plik mysql.sock w katalogu /tmp. Co więcej, podanie opcji –socket=/tmp/mysql.sock też nie pomaga. Stąd moja rada: w pliku konfiguracyjnym jak powyżej (lub poprzez opcję –socket) należy podać ścieżkę do pliku np. w swoim katalogu domowym.
Nie wiem, skąd wynika ten problem - dla mysqld_multi wszytko działa poprawnie.

Aby zakończyć pracę serwera należy użyć polecenie

```
mysqladmin -u root shutdown --defaults-file=[konfiguracja]
```

Rozdział 2

Praktyczne wprowadzenie do systemu Subversion

Michał Skrzędziejewski

2.1 Kontrola wersji

2.1.1 Ogólne wiadomości o systemach kontroli wersji

System kontroli wersji (kodu źródłowego) jest niezastąpionym narzędziem przy pracy nad oprogramowaniem. Głównym elementem tego typu systemów jest repozytorium - swoista baza danych zawierająca nie tylko kod źródłowy i/lub dokumentację projektu, ale także kompletną historię wszystkich wprowadzonych zmian. Niezależnie od tego, czy mamy do czynienia z zespołem programistów pracujących nad dużym komercyjnym projektem, czy z grupką rozsianych po świecie developerów tworzących oprogramowanie open-source, kontrola wersji jest praktycznie nieodzowna. Nawet dla pojedynczego programisty (np. studenta piszącego projekt na uczelnię) użycie systemu kontroli wersji takiego jak CVS czy Subversion może okazać się przydatne i pozwoli uniknąć potencjalnych kłopotów i niepotrzebnych frustracji. Oto podstawowe zalety systemu kontroli wersji:

- Repozytorium działa jak *globalny przycisk UNDO*. Żadne zmiany nie są ostateczne a każdy błąd można szybko poprawić. Nie jest problemem odzyskanie skasowanego przed tygodniem pliku, który okazał się niezwykle ważny i został usunięty bez potrzeby.
- Wielu programistów może pracować na tym samym kodzie, nie przeszkadzając sobie wzajemnie. Co więcej, różne osoby mogą pracować na tym samym pliku, nie martwiąc się o to, iż zmiany wprowadzone przez jedną z nich zostaną utracone. W wypadku zmian wprowadzonych do tego samego pliku system albo inteligentnie połączy to co wprowadziły obie osoby, albo da jednej z nich szanse scalenia zmian wprowadzonych przez siebie i przez drugą osobę pracującą na tym samym pliku.
- Wszelkie informacje o zmianach wprowadzanych przez poszczególnych użytkowników są zapisywane. Można łatwo ustalić kto i kiedy wprowadził daną zmianę.

- Repozytorium działa jak ”wehikuł czasu”. Zawsze istnieje możliwość ściągnięcia dowolnej wersji pliku/projektu. Bardzo proste jest konstruowanie poleceń typu: Pokaż repozytorium w takim stanie w jakim znajdowało się 2 miesiące temu.
- Programiści mogą jednocześnie pracować na kilku różnych gałęziach (ang. *branches*) oprogramowania. Dodatkowo, poprawki czy ulepszenia wprowadzone w jednej gałęzi mogą zostać wprowadzone do pozostałych.
- Trzymanie kodu źródłowego w centralnym repozytorium ułatwia tworzenie kopii zapasowej całego projektu.
- W wypadku oprogramowania open-source, posiadanie repozytorium umożliwia łatwe udostępnianie użytkownikom najnowszych wersji programów, np kompilowanych co noc (tzw. *nightly builds)*

2.1.2 Elementy systemu kontroli wersji

- **Repozytorium i kopia robocza:** Większość systemów kontroli wersji składuje projekt w centralnym repozytorium. Repozytorium może być po prostu strukturą katalogów (jak w przypadku CVS) lub bazą danych plików (Subversion). Istotny jest fakt, iż programiści nie operują bezpośrednio na plikach znajdujących się w repozytorium, lecz na kopii roboczej dostępnej na własnym komputerze, gdzie mogą wprowadzać lokalne modyfikacje.

  Kopia robocza umożliwia programistom lokalną pracę z kodem, dodawanie i testowanie zmian w razie potrzeby. Kiedy praca nad zmianami zostaje ukończona, są one zatwierdzane (ang. *commited*) i ładowane na serwer, gdzie zostają wprowadzone do projektu. Po zatwierdzeniu zmian, inni programiści podczas aktualizacji swoich wersji roboczych otrzymają najnowsze wersje plików. Mogą oni szybko przetestować wprowadzone przez nas zmiany na ich własnej kopii roboczej.

- **Rewizje:** Systemy kontroli wersji nie utrzymują jedynie najświeższej wersji plików projektu, lecz historię zmian do niego wprowadzonych. Kiedy programista zatwierdza swoje zmiany w projekcie, zostają one zachowane w rewizji (ang. *revision*). Rewizje mogą dotyczyć pojedynczych plików lub całego repozytorium. W wypadku Subversion, numer rewizji dotyczy całego repozytorium. Oznacza to, iż dana rewizja odzwierciedla stan w jakim znajdowało się całe repozytorium po wprowadzeniu przez kogoś zmiany. Np “rewizja 5” oznacza stan, w jakim znajdowało się repozytorium po wykonaniu 5 operacji *commit*.
- **Logi:** Każdy system kontroli wersji jest wyposażony w mechanizm logowania operacji wykonywanych na repozytorium. Podczas każdej operacji *commit* obowiązkowe jest wprowadzenie komentarza odnośnie wprowadzonych zmian. Stosowanie się do tej zasady ma tę zaletę, iż dowolny programista może zidentyfikować osobę i powód wprowadzenia zmian do pliku źródłowego, który nagle zaczął powodować problemy. Log pliku może też posłużyć do wygenerowania listy zmian (ang. *changelog*) przy wydawaniu kolejnej wersji oprogramowania.
- **Tagi:** Większość systemów kontroli wersji pozwala na oznaczenie (ang. *tag*) rewizji. W ten sposób można łatwo odnosić się do niej w przyszłości. Oznaczanie pozwala uniknąć odnoszenia się do rewizji poprzez jej numer lub datę.

Zamiast tego operuje się nazwą nadaną podczas operacji tagowania. Możliwość ta jest zwykle wykorzystywana przy okazji wydawania kolejnych wersji oprogramowania.

- **Gałęzie:** Tworzenie gałęzi pozwala na niezależną i równoległą pracę nad wieloma wersjami oprogramowania. Prace nad nimi toczą się niezależnie i zmiany wprowadzone do jednej gałęzi nie wpływają na pozostałe. Jeśli jednak zapadnie decyzja, iż zmiany wprowadzone do jednej gałęzi przydadzą się gdzie indziej, istnieje możliwość ich scalenia.

2.2 System Subversion

2.2.1 Prezentacja systemu Subversion

Subversion (SVN) jest darmowym, zaawansowanym systemem kontroli wersji. Jest wydawany na licencji typu open-source. Został stworzony przez grupę programistów świadomych ograniczeń starzejącego się systemu CVS. Twórcy Subversion postawili sobie za cel stworzenie systemu podobnego do CVS i mającego w założeniu go zastąpić, lecz nie powielającego jego wad. Oto najważniejsze cechy systemu Subversion:

- **Interfejs:** Interfejs Subversion jest zbliżony do interfejsu CVS. Podstawową metodą pracy jest użycie konsolowego klienta, którego składnia jest podobna do tej znanej z CVS. Istnieje wiele klientów GUI ułatwiających pracę z Subversion (np TortoiseSVN). Także coraz więcej środowisk IDE posiada zintegrowaną obsługę svn lub wtyczki zapewniające tego typu funkcjonalność.

- **Niepodzielne zatwierdzanie zmian(*atomic commits*):** Subversion działa zgodnie z zasadą "wszystko albo nic". Wszystkie zmiany przypadające na daną operację *commit* wykonywane są jednym niepodzielnym ciągiem. Przykładowo, jeśli podczas zatwierdzania 5 plików wystąpi błąd (np. zerwanie połączenia sieciowego), to repozytorium pozostanie w takim stanie w jakim znajdowało się przed rozpoczęciem operacji. Nie ma więc możliwości, że przerwanie operacji *commit* pozostawi repozytorium w niespójnym stanie.

- **Szybkość działania:** Ilość informacji transmitowanych przez sieć jest mniejsza niż w przypadku CVS. W Subversion wiele operacji może być wykonanych w trybie offline, bez potrzeby kontaktowania się z serwerem. Należy do nich np. sprawdzenie zmian wprowadzonych do kopii roboczej znajdującej się na komputerze programisty.

- **Obsługa katalogów:** W Subversion katalogi także podlegają kontroli wersji. Oznacza to iż usunięcie lub zmiana nazwy katalogu jest zwykłą operacją poprawnie obsługującą pliki znajdujące się wewnątrz katalogu, a wszystkie zmiany zostają zachowane w historii. Problemy z przenoszeniem i usuwaniem katalogów są jedną z głównych bolączek systemu CVS.

- **Tanie kopiowanie:** Utworzenie nowej gałęzi (*branch*) kodu nie nastręcza trudności, gdyż sprowadza się do wykonania polecenia kopiowania (*copy*). Dodatkowo operacja ta jest szybka (stały czas wykonania) oraz nie powoduje zwiększenia ilości zajmowanego miejsca na dysku (przechowywane są jedynie zmiany w stosunku do oryginalnej wersji).

- **Metadane:** Możliwe jest przypisanie dowolnych właściwości (również podlegających kontroli wersji) plikom lub katalogom.
- **Obsługa plików binarnych:** Subversion poprawnie obsługuje zarówno pliki binarne jak i tekstowe korzystając z efektywnego algorytmu binary-delta. Dodatkowo nie musi być w żaden specjalny sposób sygnalizowany fakt, iż plik dodawany do repozytorium jest plikiem binarnym.
- **API:** Subversion oferuje biblioteki pozwalające na korzystanie z jego możliwośći dla wielu języków, takich jak C, C++, Java czy Python.

2.2.2 Tworzenie repozytorium

Struktura typowego repozytorium svn przedstawia się następująco:

```
branches/
tags/
trunk/
```

- **trunk** - katalog w którym jest trzymana najnowsza wersja kodu źródłowego. Zwykle nad nią toczą się główne prace zespołu,
- **branches** - katalog z gałęziami kodu źródłowego,
- **tags** - katalog z tagowanymi wersjami kodu źródłowego.

Warto podkreślić, iż ten układ katalogów nie jest sztywny i można go zawsze dostosować do własnych potrzeb.

Wszystkie poniższe przykłady zakładają, iż pracujemy na lokalnym repozytorium, dostępnym poprzez system plików. W innym przypadku należy dostosować metodę dostępu do repozytorium i zamienić *file://* na np. *http://*.

Zakładając że utworzyliśmy w/w pliki w katalogu *myproject*, zakładamy repozytorium komendą

```
svnadmin create --fs-type fsfs /home/mskrzedz/repo
```

A następnie wgrywamy pliki w następujący sposób:

```
svn import -m "initial import" \
  myproject file:///home/mskrzedz/repo
```

2.2.3 Tworzenie kopii roboczej i praca z plikami

Aby utworzyć lokalną kopię roboczą, należy wydać polecenie

```
svn co file:///home/mskrzedz/repo/trunk/
```

W ten sposób na naszym komputerze znajdzie się katalog *trunk*, w którym możemy rozpocząć pracę. Komendę *checkout* wykonuje się zwykle raz - później do aktualizacji naszej kopii roboczej używamy komendy *update*. Utwórzmy teraz plik *main.c*. Aby dodać utworzony przez nas plik *main.c* do repozytorium, należy wydać komendę

```
svn add main.c
```

Należy tu zaznaczyć dwie sprawy. Po pierwsze, nie musieliśmy przy wykonaniu tej komendy podawać lokalizacji repozytorium. W każdym katalogu będącym pod kontrolą svn zostaje utworzony katalog *.svn* w którym system Subversion przechowuje swoje ustawienia. Po drugie, wykonanie komendy *add* nie spowoduje natychmiastowego dodania pliku main.c do repozytorium. Informacja o dodaniu pliku zostanie umieszczona w kolejce zadań do wykonania przy następnym poleceniu *commit*.

2.2.4 Wprowadzenie zmian w strukturze plików/katalogów

Operacje kasowania, zmiany nazwy lub tworzenia katalogu wykonuje się używając poleceń:

```
svn move / svn mv
svn delete /svn rm
svn mkdir
```

Nie powinno się do tego celu używać standardowych poleceń mv, rm i mkdir.

2.2.5 Aktualizacja kopii roboczej

W celu dokonania aktualizacji (*update*) kopii roboczej, należy wykonać komendę

```
svn update
```

Spowoduje to ściągnięcie zmian wprowadzonych do repozytorium od ostatniego wykonania komendy *update* lub *checkout*.

2.2.6 Zatwierdzenie zmian

Po dokonaniu zmian w projekcie, nalezy je zatwierdzić komendą

```
svn commit
```

Przed wykonaniem komendy *svn commit* można wykonać komendę *svn update* aby ściągnąć zmiany które nastąpiły w repozytorium w czasie, gdy pracowaliśmy nad kodem.

2.2.7 Tagowanie i tworzenie gałęzi

Tworzenie tagów i gałęzi w Subversion jest bardzo proste. Jest to efekt tego, iż system svn po prostu nie rozróżnia tych pojęć - obie operacje sprowadzają się do wykonania kopii. Operacja tagowania wygląda następująco:

```
svn copy -m "Tagging version 1_0 release" \
  file:///home/mskrzedz/rep/trunk \
  file:///home/mskrzedz/rep/tags/release_1_0
```

Aby stworzyć nową gałąź należy wydać komendę

```
svn copy -m "Created 6.0 branch" \
   file:///home/mskrzedz/rep/trunk \
   file:///home/mskrzedz/rep/branches/6_0
```

W tym momencie należałoby wykonać operację *checkout* aby stworzyć kopię roboczą nowej gałęzi. Można to jednak zrobić prościej, wydając komendę

```
svn switch file:///home/mskrzedz/rep/branches/6_0
```

Spowoduje to przełączenie bieżącej kopii roboczej na pracę w nowoutworzonej gałęzi.

2.2.8 Rozwiązywanie konfliktów

Subversion jest systemem, który dopuszcza pracę wielu osób na tym samym pliku. Nie ma mechanizmu blokowania (ang. *locking*) który dawał by na pewien czas możliwość zapisu do pliku tylko jednej osobie. System svn próbuje w miarę możliwości połączyć zmiany wprowadzone przez różnych użytkowników. Jeśli to niemożliwe, sygnalizuje konflikt. Przykładowo, nasza operacja *commit* może spowodować następującego komunikatu:

```
$ svn commit -m "fixed buffer overflow in main"
Sending     main.c
svn: Commit failed (details follow):
svn: Out of date: 'main.c' in transaction '2-1'
```

Oznacza to, że zawartość pliku zmieniła się od ostatniej aktualizacji. Należy teraz użyć polecenia *svn update*. W tym momencie możliwe są następujące scenariusze:

- Subversion połączył pliki. Jeśli w komunikacie z serwera, przy nazwie pliku widnieje "G":

  ```
  $ svn update
  G   main.c
  Updated to revision 5.
  ```

  Znaczy to, iż Subversion dokonał automatycznego połączenia plików. Po sprawdzeniu, czy wynikowy plik jest poprawny, można zatwierdzić go komendą *svn commit*.

- Subversion nie zdołał automatycznie połączyć plików. Objawia się to literką "C" przy nazwie pliku:

  ```
  $ svn update
  C   main.c
  Updated to revision 5.
  ```

W tym przypadku można albo cofnąć wprowadzone przez siebie zmiany komendą *svn revert main.c*, albo edytować plik i ręcznie połączyć zmiany wprowadzone przez siebie i przez innego programistę. W takim przypadku, w konfliktowym pliku svn zaznaczy nam miejsca, w których nasza wersja pliku różni się od tej zatwierdzonej przez drugą osobę. Po poprawieniu pliku należy zasygnalizować, iż konflikt został rozwiązany. W naszym przypadku należy wydać komendę *svn resolved main.c*. Potem pozostaje jedynie zatwierdzenie zmian komendą *commit*.

2.2.9 Inne przydatne komendy

```
svn status
```

Pokazuje stan plików w bieżącym katalogu (jakie pliki są pod kontrolą svn, które są w stanie konfliktu, etc.).

```
svn info katalog
```

Pokazuje informacje na temat lokalnego lub zdalnego katalogu.

```
svn ls katalog
```

Pokazuje zawartość zadanego katalogu w repozytorium.

```
svn log plik.c
```

Pokazuje historię zmian wprowadzonych do pliku plik.c.

```
svn revert
```

Cofa zmiany wprowadzone lokalnie do kopii roboczej.

```
svn help komenda
```

Pokazuje dostępną pomoc na temat danej komendy.

2.3 Repozytorium zespołu RSO5

2.3.1 Lokalizacja repozytorium

Zgodnie z wymaganiami projektu, na serwerze *openone* zostało założone repozytorium svn. Znajduje się ono w katalogu */home/rso/mskrzedz/rso_rep*. Tylko osoby z grupy rso5 posiadają możliwość zapisu do repozytorium. Struktura repozytorium wygląda następująco

```
branches
\---mysql_official
docs
tags
trunk
\---mysql_rso
\---proxy_daemon
```

- branches/mysql_official: Oficjalna wersja MySQL, bez modyfikacji
- docs: Katalog z dokumentacją
- tags: Tu będą umieszczane otagowane wersje, np przygotowywane do demonstracji podczas milestone'a
- trunk/mysql_rso: Wersja MySQL zmodyfikowana przez zespół rso5
- trunk/proxy_daemon: Daemon zarządzający przekazywaniem zapytań do serwerów MySQL

Kopię roboczą można pobrać komendą

```
svn checkout file:///home/rso/mskrzedz/rso_rep/...
```

Gdzie ... oznacza katalog do pobrania, np:

```
svn checkout \
  file:///home/rso/mskrzedz/rso_rep/trunk/mysql_rso
```

2.3.2 Backup repozytorium

Backup repozytorium realizowany jest poprzez skrypt backup_repo.sh:

```
REPO_PATH=/home/rso/mskrzedz/rso_rep
REMOTE_PATH=docent@rasta.pl:rso_backup/
DUMPFILE=dump_`date +%F_%H-%M`

svnadmin dump $REPO_PATH > $DUMPFILE
gzip $DUMPFILE
scp $DUMPFILE.gz $REMOTE_PATH
rm $DUMPFILE.gz
```

Repozytorium jest wpierw zapisywane poleceniem *svnadmin dump* do postaci w rodzaju dump_2006-03-31_03-00, pakowane a następnie wysyłane na zewnętrzny serwer poleceniem *scp*. Skrypt ten jest wywoływany poprzez cron.

2.3.3 Dostęp do repozytorium z zewnątrz

Serwer *openone* nie jest dostępny spoza serwerów *galera, mion, csd*. Powoduje to problem w uzyskaniu dostępu do repozytorium z zewnątrz. Dopisanie nowego kodu wymaga każdorazowego zalogowania się na serwer pośredniczący, a następnie na openone. Dodatkowo praca z kodem źródłowym jest możliwa tylko lokalnie co przy opóźnieniu klienta ssh może być sporym utrudnieniem.

Można temu zaradzić w prosty sposób, tworząc przezroczysty tunel za pomocą popularnego programu *xinetd*. Oto plik konfiguracyjny umożliwiający połączenie z repozytorium na serwerze *openone*.

```
defaults
{
  only_from       = 127.0.0.1
  instances       = 3
}

service svnserve
{
  type = UNLISTED
  user = creep
  flags = IPv4
  socket_type = stream
  protocol = tcp
  wait = no
  server = /usr/bin/ssh
  server_args = mkoziej@galera.ii.pw.edu.pl \
         ssh mkoziej@openone.ia.pw.edu.pl \
         svnserve -t -r /home/rso/mskrzedz/rso_rep
  port = 6666
}
```

Plik ten należy dostosować do własnych potrzeb. W szczególności w polu *user* należy wpisać nazwę użytkownika komputera z którego będziemy łączyć się do repozytorium, *mkoziej@galera.ii.pw.edu.pl* oraz *mkoziej@ia.pw.edu.pl* zamienić na odpowiedni login na serwerach galera i openone. Dodatkowo należy utworzyć za pomocą polecenia *ssh-keygen*, a następnie umieścić w odpowiednim miejscu klucze, umożliwiające bezhasłowe połączenie między naszym komputerem i serwerem *galera* oraz między serwerem galera i openone. Parametr *port* pozwoli na wybór lokalnego portu na którym uruchomione ma być połączenie z svn. Jeśli wszystkie te warunki zostaną spełnione należy uruchomić xinetd komendą

```
xinetd -f svn.conf
```

Gdzie svn.conf jest nazwą stworzonego przez nas pliku konfiguracyjnego. Jeśli wszystko przebiegło pomyślnie, można już korzystać z repozytorium, np:

```
svn --username mkoziej ls svn://localhost:6666
```

Jako parametr po *–username* należy podać nazwę użytkownika. Hasło, o które poprosi svn, zostanie zapamiętane i następnym razem nie jest już wymagane jego podanie. Użycie *–username* gwarantuje, że zmiany wprowadzone do plików będą skojarzone z użytkownikiem, który je wprowadził.

Rozdział 3

Co powinno się wiedzieć o rozwiązaniu OpenSSI w kontekście uruchamiania serwera MySQL w środowisku klastrowym.

Marcin Koziej

3.1 Czym jest OpenSSI?

OpenSSI (Open Single System Image) jest systemem klastrowym tworzącym środowisko zbliżone dla uruchamianych aplikacji do pojedynczego systemu. Jednocześnie, ponieważ jest to system rozproszony, cechuje go skalowalność wykorzystania zasobów, wysoka dostępność, odporność na uszkodzenie fragmentu systemu.

Aby uzyskać takie własności, funkcjonalność systemu rozproszonego wbudowana została w różne płaszczyzny systemu operacyjnego Linux.

1. Jedna, obejmująca cały klaster, przestrzeń procesów.
2. Jeden, dostępny dla wszystkich procesów klastra, rozproszony system plików.
3. Mechanizm komunikacji między węzłami - na poziomie jądra oraz procesów użytkowych.
4. Mechanizmy komunikacji międzyprocesowej POSIX (POSIX IPC) pomiędzy wszystkimi procesami klastra.
5. Imitowanie pojedynczego interfejsu sieciowego przez cały klaster (pojedynczy adres IP).

3.2 Architektura OpenSSI

OpenSSI ma architekturę *Master-Slave*, gdzie istnieje jeden węzeł nadrzędny (z ewentualnymi węzłami zapasowymi) oraz wiele równoważnych węzłów podrzędnych. Serwery Master w większości podsystemów OpenSSI odgrywają bardzo istotną rolę, udostępniając poszczególnym węzłom połączenia nawiązane z klastrem, korzeń systemu plików, zarządcę blokad, itp. Aby ten węzeł nie stał się pojedynczym punktem awarii, wykorzystuje się serwery zapasowe, które przejmują rolę master'a w przypadku jego awarii.

Każdy z węzłów klastra ma jeden lub więcej adresów, które służą do komunikacji wewnątrz systemu. Powinny to być nierutowalne adresy lokalne. Lista węzłów jest zapisana w pliku */etc/clustertab*, w którym znajdują się: numer węzła, adres ip, adres ethernetowy (MAC karty sieciowej), flaga węzła startowego (*initnode*), oraz opcjonalnie partycja z głównym system plików (*root filesystem*) dla węzła startowego.

3.2.1 ICS

Na potrzeby klastra stworzono podsystem komunikacji międzywęzłowej *ICS - Internode Communication Subsection*. Jego cechy to:

- Możliwość komunikacji jądro–jądro, także przed inicjalizacją stosu TCP/IP.
- Zapewnienie komunikacji między wszystkimi węzłami klastra, współpraca z podsystemem Cluster Membership (CLMS) kontrolującym jakie węzły należą do klastra.
- Obsługa różnych rozmiarów przesyłanych wiadomości (min. 64K), kolejkowanie, priorytetowanie (także w celu uniknięcia zakleszczeń),
- Możliwość przekazywania wiadomości lub zdalne synchroniczne/asynchroniczne wywołanie procedur. Interfejs RPC definiuje się w pliku svc, *icsgen* (w katalogu *cluster/util* w źródłach jądra) w celu wygenerowania 'zaślepek' (stubs).

3.2.2 CLMS

Podsystem *CLMS (Cluster Membership)* wykorzystuje ICS do koordynacji połączeń między węzłami klastra (może więc nawiązać połączenia z innymi węzłami bardzo wcześnie podczas uruchomienia – przed inicjalizacją stosu TCP/IP) oraz utrzymywaniem spójnego stanu przynależności do klastra, w przypadkach dołączania się lub awarii węzłów. CLMS zarządza zmianami poziomów uruchomienia (runlevels) węzłów, reagowaniem na zdarzenia dołączenia lub odłączenia węzła (udostępniając API służące do monitorowania tych zdarzeń).

3.2.3 Polecenia

- */bin/cluster* - wypisuje informacje o węzłach klastra, domyślnie aktywne węzły.
- */bin/clusternode_num* - podaje numer węzła na którym jest proces.
- */bin/clusternode_avail,/sbin/clusternode_getstate* - sprawdza stan danego węzła.

3.2.4 Pliki

- */proc/clms_masterlist* - lista węzłów Master

Interfejsem programisty do funkcjonalności CLMS (między innymi) jest wywołanie systemowe *ssisys*, którego plik nagłówkowy to */usr/include/linux/ssisys.h*.

3.3 Vproc

3.3.1 Charakterystyka Vproc

Vproc jest podsystemem zarządzającym procesami w klastrze, pozwalając uzyskać jedną wspólną przestrzeń procesów w obrębie całego klastra. Procesy mogą być dowolnie rozproszone w klastrze, niezależnie od położenia procesów z którymi są w relacji dziecko-rodzić, jednej grupie procesów lub grupie sesji. Funkcjonalność obejmuje:

- Wspólny system /proc, widoczność każdego procesu w każdym węźle.
- Możliwość uruchomienia procesu na innym węźle.
- Przeźroczystą migrację procesów (przenoszenie procesu z węzła na węzeł).
- Przeźroczysta, rozproszona obsługa ptrace() (używana do śledzenia i przez debugger'y).
- Poprawną obsługę odłączenia/przyłączenia do klastra węzłów.

3.3.2 Polecenia

- */bin/where_pid* – zwraca numer węzła na którym jest uruchomiony proces o zadanym numerze.
- */bin/onnode*,*/bin/onall* - uruchom dane polecenie na węźle numer/na wszystkich węzłach.
- */bin/bash-ll* - wersja programu bash, która rozkłada obciążenie uruchamianych procesów na cały klaster.
- */usr/bin/migrate* - migruje proces na węzeł numery obu podaje się jako argument. Równoważne wpisaniu numeru węzła do pliku */proc/#/goto*, gdzie # jest numerem pid procesu.

3.4 System plików

System plików w OpenSSI jest wspólny dla całego klastra i umieszczony w węźle inicjującym, zgodnie z wpisem w pliku */etc/clustertab*. Systemy plików kolejno dołączanych węzłów są montowane w */cluster/node#*, gdzie "#" oznacza numer węzła. Urządzenia dyskowe są montowane w każdy z węzłów zgodnie z wpisami w pliku */etc/fstab*, z tą różnicą, że plik ten zawiera wpisy `node` w polu opcji, które określają którego węzła dane urządzenie dotyczy.

OpenSSI obsługuje kontekstowo zależne dowiązania symboliczne (*CDSL - context dependent symlinks*), które pozwalają na stworzenie dowiązania do pliku zależnego od węzła, na którym znajduje się proces otwierający dane dowiązanie. W nazwie pliku na który wskazuje dowiązanie można wstawić ciągi {*hostname*},{*nodenum*}, {*mach*}, {*os*}, które są odpowiednio zamienane na nazwę hosta, numer węzła, architekturę węzła, kod systemu operacyjnego.

Funkcjonalność CDSL jest wykorzystana np. w dowiązaniu */cluster/node*{*nodenum*}, które zawsze wskazują na system plików bieżącego węzła, lub /tmp będące wskazaniem na katalog plików tymczasowych na dyskach lokalnych każdego z węzłów.

3.5 HA-LVS

3.5.1 Charakterystyka HA-LVS

OpenSSI implementuje technologię HA LVS – High Availability Linux Virtual Server. Klaster dla zewnętrznego klienta realizuje semantykę pojedynczego serwera – ma przypisany jeden (lub więcej) adres IP (CVIP - Cluster Virtual IP), do którego wysłane zapytania są przyjmowane i rozpraszane wewnątrz klastra na różne maszyny przeźroczyście dla klienta, zgodnie z pewnym algorytmem rozkładania obciążenia. Obsługę CVIP konfiguruje się za pomocą pliku */etc/cvip.conf*. W nim ustala się wirtualne adresy ip, węzły pełniące funkcję rozdzielania zapytań (bramy,gateway'e), węzły będące serwerami odpowiadającymi na zapytania oraz metodę przekazywania połączeń. Dla openone.ia.pw.edu.pl gateway'em jest węzeł 1, a metodą NAT (Network Adress Translation), inną metodą jest DR (Direct Routing), gdzie pakiety są rutowane do węzłów bez zmiany adresów nagłówka IP.

Aby usługa była przez klaster OpenSSI udostępniona w ten sposób, należy skonfigurować przedział portów, które mają być przez nią obsługiwane za pomocą narzędzia */usr/sbin/setport_weight*, następne, kiedy aplikacja wykona funkcje systemu `listen()` dla danego portu, serwer na danym węźle zostanie dodany do puli serwerów.

3.5.2 Pliki

- Skrypty konfigurujące LVS znajdują się w katalogu */cluster/etc/ha-lvs*.
- Usługi uruchamiane standardowo w konfiguracji HA-LVS są startowane za pomocą skryptów */cluster/etc/ha-lvs.d/**.
- */etc/lvs.VIP.active* – Zawiera adres CVIP oraz numer węzła, który kontroluje dostęp do klastra przez ten adres (master director). Ta sama informacja znajduje się w */proc/cluster/lvs*.
- */proc/cluster/lvs_routing* – Wskazuje, jaka metoda rozpraszania połączeń jest aktywna.
- */proc/cluster/ip_vs_portweight* – zawiera informacje dot. przedziałów portów, które będą automatycznie (przy wykryciu wywołania listen()) obsługiwane przez system rozkładający obciążenie.
- */bin/clustername* - wypisuje nazwę klastra do jakiej powinni odwoływać się zewnętrzni klienci (domenę wskazującą na CVIP).

3.6 OpenSSI implementuje obiekty IPC specyfikacji POSIX.

- sygnały
- potoki
- kolejki fifo
- sygnały
- kolejki wiadomości
- pamięć dzieloną
- gniazda

Obiekty są tworzone w węźle, w którym działa tworzący je obiekt, są jednak dostępne w całym klastrze. Wspólną przestrzeń nazw obiektów IPC obsługuje serwer nazw IPC Namesever, automatycznie reaktywowany w przypadku awarii węzła, na którym pracował. Obiekty IPC nie mogą migrować pomiędzy węzłami.

Obiekty IPC mogą być tworzone jako lokalne, do których można odwoływać się jedynie z danego węzła.

Polecenie *ipcs* wyświetla aktualnie istniejące w systemie obiekty.

W OpenSSI można wykorzystać protokół MPI (Message Passing Interface, implementacja LAM/MPI `http://www.lam-mpi.org/`). MPI jest protokołem bardzo powszechnym i dobrze opisanym (np. na stronie projektu `http://www.lam-mpi.org/using/docs/`). Pliki nagłówkowe znajdują się w */usr/include/lam*.

Rozdział 4

Wprowadzenie do zagadnienia realizacji niezawodności i wydajności poprzez replikację zasobów. Wstęp do algorytmów realizacji wysokiej dostępności i niezawodności rozproszonych zasobów.

Krzysztof Jamróz

4.1 Wstęp

Jest wiele przyczyn stosowania systemów rozproszonych i klastrowych. Mają one niezaprzeczalne zalety. Mogą mieć większą wydajność przez zrównoleglenie przetwarzania i rozdzielanie zadań między węzły. Posiadają cechę tolerowania uszkodzeń – jeśli jeden z węzłów przestanie działać, to inne mogą dalej dostarczać usług, być może tylko z mniejszą wydajnością.

Oprócz wymienionych zalet są systemy rozproszone mają również wady. Są przede wszystkim znacznie bardziej złożone i skomplikowane niż systemy scentralizowane, wymagają stosowania bardziej wyrafinowanych algorytmów.

Jednym z podstawowych zagadnień przy projektowaniu systemu rozproszonego, którego obiekty posiadają jakiś stan wewnętrzny i są współdzielone między różnymi węzłami systemu jest zapewnienie tego, żeby modyfikacje dokonywane na jednym z węzłów zostały rozpropagowane do pozostałych. Ma to dwa cele: zachowanie spójności stanu obiektu w systemie oraz zapewnienie zapewnienie, że w przypadku awarii zmiany nie zostaną utracone. Wymagania te są realizowane przez algorytmy replikacji.

4.2 Typy algorytmów replikacji

Można wyróżnić kilka podziałów algorytmów replikacji skupiając się na różnych ich aspektach. Dalej poruszone zostały kwestie szczególnie istotne z punktu widzenia rozproszonych baz danych.

4.2.1 Moment replikacji

Ze względu na moment replikacji można wyróżnić replikację:

- *Synchroniczna* (*eager*) – replikacja następuje w ramach transakcji przed ostatecznym jej zatwierdzeniem (commit), a gdy się nie powiedzie, np. z powodu konfliktu, transakcja jest wycofywana. Algorytmy tego typu umożliwiają łatwe utrzymanie spójności danych bez szczególnych dodatkowych zabiegów. Wadami mogą być słaba skalowalność, mała wydajność, dłuższy czas odpowiedzi (gdyż replikacja do wszystkich węzłów musi nastąpić przed udzieleniem odpowiedzi klientowi).
- *Asynchroniczna* (*lazy*) – zdalne kopie są aktualizowane dopiero po zatwierdzeniu transakcji. Ogranicza to tolerowanie uszkodzeń, może powodować nieaktualność lub niespójność danych w replikach. Algorytmy tego typu mają większą wydajność niż algorytmy replikacji synchronicznej. Ten typ replikacji jest często stosowany w systemach rozległych geograficznie i tam, gdzie niewielka nieaktualność danych nie jest poważnym problemem.
- Mieszane.

4.2.2 Architektura

Można wyróżnić następujące rodzaje architektury systemu:

- Z węzłami nadrzędnymi i podrzędnymi (*primary* i *secondary*) – zmiany dokonywane są w jednej, wybranej kopii (primary), a następnie są propagowane do pozostałych (secondary), które mają dane tylko do odczytu. Uaktualnienie może nastąpić na żądanie (pull) węzłów secondary lub po zmianie danych (push). Ten typ replikacji może powodować powstanie *single point of failure* oraz ogranicza wydajność transakcji modyfikujących dane, gdyż wszystkie są wykonywane w tym samym węźle. Ten typ replikacji może być implementowany z użyciem zmaterializowanych widoków lub triggerów.
- Z węzłami równorzędnymi (*update everywhere*, *multimaster*) – wiele kopii może być modyfikowanych jednocześnie. Wymaga to częstej synchronizacji między replikami, łączenia zmian wykonanych w różnych kopiach i rozwiązywania konfliktów, kiedy kilka węzłów na raz modyfikuje ten sam zasób (np. wiersz w tabeli bazy danych). Zaletą tego podejścia jest to, że modyfikacje niezależnych obiektów (np. różnych wierszy w tabeli) mogą odbywać się równolegle.
- Mieszana – między częścią węzłów następuje replikacja multimaster, pozostałe natomiast zawierają migawkowe kopie danych.

4.2.3 Równoważność węzłów

Występuje także podział rozwiązań ze względu na równoważność węzłów:

- *Pełna replikacja* – każdy węzeł posiada całą bazę danych. W takim przypadku wszystkie modyfikacje muszą dotrzeć do każdego z węzłów. Ogranicza to skalowalność i zwiększa koszty komunikacji przy dużej ilości węzłów i częstych modyfikacjach. Zaletą jest duża wydajność przy transakcjach odczytu.

- *Częściowa replikacja* – niektóre dane nie są replikowane w ogóle lub są replikowane tylko na niektórych węzłach. Umożliwia to równoważenie obciążenia, zmniejszenie ilości przesyłanych komunikatów, gdyż nie wszystkie węzły muszą otrzymywać informacje o modyfikacjach, oraz lepsze skalowanie. Komplikuje to jednak system. Nie każdy węzeł może wykonać samodzielnie każde zapytanie, co może wymagać wprowadzenia rozproszonych transakcji obejmujących wiele węzłów, odpowiedniego podziału danych między węzły pod kątem przewidywanych zapytań albo implementacji migracji danych między węzłami. Możliwe jest zarówno rozdzielanie całych obiektów (np. tabel) jak i ich fragmentów (np. niektórych kolumn lub wierszy tabeli).

4.3 Sposoby zwiększania wydajności

Wydajność zapewniana przez algorytm jest bardzo istotnym czynnikiem, gdyż jednym z głównych motywów stosowania systemów rozproszonych, a szczególnie klastrowych jest chęć zwiększenia wydajności.

Nawet w systemach klastrowych, których węzły połączone są szybką siecią lokalną, należy brać pod uwagę ilość komunikacji, która w istotny sposób rzutuje na wydajność. Im więcej komunikatów i punktów synchronizacji tym mniej wydajny system. Ważnym czynnikiem jest też to w jaki sposób ilość przesyłanych komunikatów rośnie wraz ze wzrostem liczby węzłów w systemie, co ma bezpośrednie przełożenie na skalowalność rozwiązania.

Można wymienić kilka możliwych sposobów zwiększania wydajności:

- Stosowanie efektywnych algorytmów, które przesyłają mało komunikatów i nie wymagają wysokiego poziomu ich uporządkowania.

- Unikanie stosowania rozproszonych blokad, które wpływają negatywnie na wydajność i mogą być przyczyną zakleszczeń.

- Grupowanie zapisów w paczki w przesyłanie ich razem dopiero na końcu transakcji. Dzięki temu uzyskuje się znaczne zmniejszenie ilości przesyłanych komunikatów.

- Replikacja może zwiększać prawdopodobieństwo konfliktów i zakleszczeń. Wynika to z potencjalnie większej ilości transakcji wykonujących się równolegle oraz dłuższego czasu trwania transakcji (z powodu komunikacji). Do rozwiązywania konfliktów może być przydatne zastosowanie grupowej komunikacji zapewniającej całkowite uporządkowanie komunikatów do rozsyłania informacji o zapisach pochodzących z jednej transakcji. Można także nadawać transakcjom znaczniki czasowe i dawać wyższy priorytet zapisom tej o wcześniejszym znaczniku.

Możliwe są także rozwiązania, które na podstawie wiedzy o bieżącym stanie przetwarzania i napływających transakcjach będą opóźniały na pewien czas transakcje, które mogłyby wywołać konflikt z już wykonującymi się. Można na przykład wykrywać transakcje modyfikujące te same wiersze w tabeli.

- Stosowanie częściowej replikacji w sytuacji, gdy modyfikacje danych są częste. Podział danych między węzły może być modyfikowany na podstawie danych o obciążeniu.
- Równoważenie obciążenia węzłów.

4.4 Tolerowanie uszkodzeń

Częstym wymaganiem stawianym systemom rozproszonym, a także jedną z przyczyn ich tworzenia jest tolerowanie uszkodzeń – na przykład wypadnięcia węzła lub awarii łącza komunikacyjnego – w sposób przezroczysty dla klienta oraz zapewniający ciągłość działania systemu.

W związku z tym należy rozwiązać następujące problemy:

- Wykrywanie uszkodzonych węzłów na podstawie braku odpowiedzi w określonym czasie (timeout).
- W przypadku podziału sieci na fragmenty pracę kontynuuje ta część, która zawiera więcej niż połowę węzłów.
- Wiarygodne dostarczanie komunikatów do wszystkich działających węzłów przy występowaniu awarii – np. protokoły *view change*
- Ponowne dołączanie węzła, który uległ awarii. Najpierw musi zostać odtworzona spójność bazy danych węzła na podstawie jego własnych logów. Następnie muszą zostać uwzględnione wszystkie zmiany, które zaszły w systemie w trakcie, gdy dany węzeł był wyłączony. Te transfery muszą odbywać się w trakcie normalnego działania systemu jako całości i być tak zorganizowane, aby nie utracić żadnej modyfikacji. Dopiero, gdy to się zakończy, węzeł może rozpocząć normalne przetwarzanie transakcji.

Oprócz rozwiązania wymienionych problemów dotyczących oprogramowania należy zapewnić również odpowiedni sprzęt, który również będzie pozwalał na tolerowanie uszkodzeń, jednak to zagadnienie wykracza to poza ramy niniejszego opracowania.

Protokoły tolerowania uszkodzeń wprowadzają dodatkowy narzut jeśli chodzi o komunikację. Dlatego należy znaleźć kompromis między wymaganym tolerowaniem uszkodzeń a wydajnością.

4.4.1 Protokoły view change

Każdy węzeł przechowuje bieżący widok węzłów w systemie V_i. Każdy wysyłany komunikat jest oznaczany numerem bieżącego widoku. Gdy grupa węzłów wykryje zmianę ilości węzłów inicjuje procedurę zmiany widoku, która gwarantuje, że wszystkie węzły, które nie uległy awarii otrzymają dokładnie te same komunikaty. Może to wymagać ponownego przesłania komunikatów nadanych przez węzeł, który uległ

awarii, przez jeden z węzłów, który je odebrał. Potem następuje przełączenie na nowy widok V_{i+1}.

Protokoły te pozwalają zrealizować atomowy broadcast oraz umożliwiają wykrywanie, czy węzeł znajduje się w dominującej części systemu po podziale w wyniku awarii. Tylko węzły z dominującej części mogą kontynuować działanie, pozostałe zachowują się tak, jakby uległy awarii.

4.5 Implementacje komercyjne

W systemach komercyjnych zazwyczaj stosowana jest replikacja asynchroniczna ze względu na łatwość implementacji i wydajność. Istotnym czynnikiem może być tu fakt, że repliki często znajdują się w odległych miejscach, kiedy komunikacja wprowadza znacznej opóźnienia. W wielu przypadkach używany jest model z węzłami nadrzędnymi i podrzędnymi. Replikacja jest realizowana przez utworzenie migawki, która jest uaktualniana na żądanie lub za pomocą specjalnych triggerów. Niespójność danych w systemach multimaster jest rozwiązywana automatycznie (np. na podstawie znaczników czasowych) lub radzenie sobie z nią jest przerzucane na aplikację.

Ciekawym rozwiązaniem jest Oracle Real Application Clusters. W tym systemie jest jedna wspólna macierz dyskowa, która zawiera bazę danych i do której podłączone są węzły klastra. Węzły są również połączone siecią między sobą i komunikują się aby synchronizować dostęp do plików danych. Dzięki temu rozwiązaniu unika się kosztownej replikacji danych między serwerami.

Rozdział 5

Technologie klastrowe w MySQL 5.1

Michał Podsiadłowski

MySQL w wersji 5.1 posiada 2 technologie, które umożliwiają tworzenie rozproszonych baz danych. Pierwszą z nich jest replikacja, drugą – MySQL Cluster.

5.1 Replikacja

MySQL udostępnia jednostronną asynchroniczną replikację na podstawie zindeksowanego binarnego logu, do którego serwer master zapisuje wszystkie zmiany w bazie danych. Serwery slave podłączają się do serwera master podając pozycję, na której skończyły czytać log, otrzymując w zamian listę wszystkich zmian w bazie danych. Kiedy synchronizacja się zakończy, slave blokuje się w oczekiwaniu na komunikat o kolejnych zmianach. Replikację można skonfigurować w trybie łańcuchowym jak również multiple-master. Jednakże w trybie multiple-master konieczne jest ustawienie zmiennych auto_increment_increment i auto_increment_offset w celu uniknięcia dublowania się wartości inkrementowanych przez bazę danych, które w większości przypadków są używane w celu uzyskania unikalności wartości. Replikacja może przebiegać na podstawie wierszy jak również kwerend. Jeśli prowadzimy zmianę wartości wyłącznie na serwerze master nie robi to większej różnicy, którą wybierzemy. Natomiast kiedy zmiany prowadzone są również na serwerach slave wybór odpowiedniego typu jest bardzo istotny ze względu na charakter tych typów. Przykładem niech będzie tu prosta sekwencja kwerend.

```
slave: INSERT INTO tbl VALUES (1);
master: DELETE FROM tbl;
```

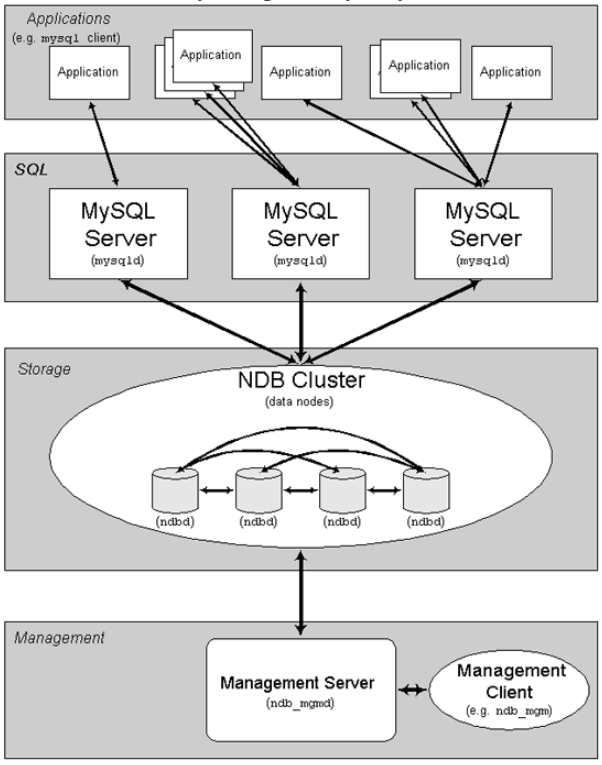
Używając replikacji na podstawie kwerend usuniemy wszystkie wiersze, natomiast replikacja na podstawie wierszy na serwerze slave usunie tylko te wartości, które były również na serwerze master. W rezultacie tabele na serwerze slave mogą mieć inną zawartość niż na serwerze master, co może powodować problemy z dalszą replikacją.

Replikacja może zapewnić nam:

- większą odporność na awarię — w razie awarii serwera master serwer slave przejmuje jego rolę.
- lepszy czas dostępu — w środowiskach gdzie dominują kwerendy select (czyli w większości przypadków) serwer master może zostać odciążony przez kierowanie przez load-balancer zapytań do serwerów slave. W celu zachowania spójności bazy zapytania modyfikujące muszą być kierowane do serwera master.
- ułatwienie administracji — kopie bazy można wykonywać na serwerach slave nie obciążając w ten sposób serwera master.

5.2 MySQL Cluster

MySQL Cluster jest to technologia oparta o architekturę *share-nothing* (systemy operacyjne na węzłach są zupełnie rozłączne i posiadają osobną pamięć i dyski). Rozwiązanie to opiera sie o silnik bazodanowy NDB (Network Database). W skład klastra MySQL Cluster wchodzą węzły danych z uruchomionym demonem ndbd, węzły zarządzające z demonem ndb_mgmd, węzły klienckie z mysqld oraz dodatkowo specjalistyczne programy klienckie napisane z użyciem NDB API. Relację tych węzłów można zobaczyć na poniższym rysunku:



Z poziomu klientów MySQL silnik NDB zachowuje się przezroczyście. Nie jest zauważalna żadna różnica pomiędzy składowaniem za pomocą niego a składowaniem danych w bazach danych typu INNODB czy MyISAM.

5.2.1 Silnik NDB

NDB jest to klastrowy silnik składujący dane. Co jest dla niego charakterystyczne przechowuje on dane w pamięci. Jak wyżej zostało to wymienione w skład tego silnika wchodzą: demon ndbd - przechowujący dane i demon ndb_mgmd trzymający konfigurację. W klastrze podczas uruchamiania musi być obecny węzeł zarządzający. Na podstawie informacji z tego węzła pozostałe węzły są uruchamiane. Podczas pracy nie jest on konieczny do poprawnego funkcjonowania systemu. Aczkolwiek w czasie pracy ndb_mgmd tworzy dziennik klastra, który jest głównym źródłem wiadomości na temat stanu klastra, zaistniałych problemów itp.

Komunikacja

Komunikacja odbywa się za pomocą modułów zwanych Transporters. NDB obsługuje w tym momencie Transportery:

- "Shared memory" dla komunikacji przy użyciu pamięci dzielonej. Wykorzystywane jest to kiedy demony są uruchomione na jednej fizycznej maszynie. Oczywiście ma to sens jeśli jest to maszyna wieloprocesorowa.
- TCP
- OSE Link Layer dla OSE Delta Systems
- SCI dla sieci "Scalable Coherent Interface"

Transporter TCP jest wolniejszy niż OSE, natomiast OSE jest wolniejszy niż SCI i "Shared memory".

Rozpraszanie i replikacja danych

Klaster NDB automatycznie rozprzestrzenia i replikuje dane przezroczyście dla aplikacji użytkownika. Wiersze należące do jednej tabeli (logicznej) są rozproszone horyzontalnie po węzłach na podstawie wartości funkcji mieszającej (przeważnie z głównego klucza). Standardowo NDB tworzy repliki wszystkich danych aby podnieść niezawodność. Ilość replik jest konfigurowalna lecz ograniczona do 4. Grupa węzłów jest to zbiór węzłów replikujący te same fragmenty danych.Ilość węzłów w grupie jest równa ilości replik. Standardowa konfiguracja z 4 węzłami danych zawiera 2 repliki. Repliki są uaktualniane synchronicznie, oznacza to, że każda modyfikująca transakcja jest potwierdzana dopiero po modyfikacji wszystkich węzłów.

Współbieżność i spójność

Klaster NDB używa pesymistycznej kontroli współbieżności na bazie blokad. Jeśli niemożliwe jest założenie blokady w podanym czasie to jest generowany błąd przedawnienia. Deadlocki rozwiązywane są przez przedawnianie i kolejki blokad. Spójność jest gwarantowana przez *shadow copies* w pamięci. Gdy wykonywany jest rollback dane są modyfikowane spowrotem do wartości z tej kopii.

Część II

Faza II - prototyp

Rozdział 6

Koncepcja rozwiązania

Krzysztof Jamróz

6.1 Założenia projektowe

Podstawowe założenia projektowymi przyjęte przez nas są następujące:

- Dane muszą być spójne, niedopuszczalne jest powstanie nieścisłości, które musiałyby być rozwiązywane przez użytkownika.
- Dane muszą być bezpieczne, to znaczy w przypadku awarii dowolnego węzła nie mogą zostać utracone żadne wyniki zatwierdzonych transakcji.
- Dostosowanie aplikacji do korzystania z rozwiązania klastrowego powinno wymagać co najwyżej drobnych zmian w kodzie tej aplikacji.
- Użycie InnoDB jako silnika przechowującego dane.

Cechy aplikacji, dla której zaproponowane rozwiązanie powinno się dobrze sprawdzać:

- Znaczna przewaga transakcji odczytu nad transakcjami zapisu.
- Modyfikacje dotyczące naraz niewielkiej ilości wierszy.
- Wymaganie wysokiej dostępności.

6.2 Ustalenia dotyczące architektury

Ta sekcja przedstawia ustalenia dotyczące architektury naszego rozwiązania. Opisane są podjęte przez nasz zespół decyzje łącznie z uzasadnieniem oraz dyskusją innych możliwości. Decyzje zostały podjęte na spotkaniach zespołu w wyniku dyskusji nad wadami i zaletami różnych rozwiązań.

6.2.1 Replikacja

Kluczową decyzją było ustalenie sposobu replikacji danych w systemie.

Synchroniczność

Zostanie użyta replikacja synchroniczna gdyż jest w niej łatwiej zachować spójność danych (przy asynchronicznej jest to bardzo trudne). Przed odesłaniem odpowiedzi do klienta o zatwierdzeniu transakcji, wszystkie modyfikacje muszą zostać rozesłane do wszystkich węzłów, które posiadają repliki zmodyfikowanych tabel oraz węzeł mastera musi otrzymać potwierdzenia, że dane zostały na nich uaktualnione do odpowiedniego stanu.

Replikacja asynchroniczna oferuje większą wydajność, jednak za cenę niespójności w danych. Jest ona szczególnie dobra w systemach rozległych geograficznie, w których są znaczne opóźnienia transmisji. W środowisku klastrowym czynnik ten nie będzie zbyt istotny, gdyż węzły są połączone szybką siecią lokalną.

Równoważność węzłów

Użyjemy pełnej replikacji, w której wszystkie węzły posiadają kopie wszystkich danych.

Rozwiązanie to jest proste i bardzo wydajne w transakcjach odczytujących dane. Jego wadą jest mała wydajność przy zapisach i gorsza skalowalność w porównaniu do replikacji częściowej (w której węzły posiadają tylko część tabel), gdyż informacje o modyfikacjach muszą być rozesłane do wszystkich węzłów. W efekcie pełna, synchroniczna replikacja może być wolniejsza przy zapisach od zwykłego pojedynczego serwera.

Replikacja cześciowa wymaga podjęcia decyzji o sposobie podziału tabel. Aby uniknąć stosowania transakcji obejmujących wiele węzłów musi on zostać tak zrealizowany, aby zbiór tabel potrzebnych do wykonania każdej transakcji był w całości na jakimś węźle. Podział może być ustalany przez użytkownika lub wykonywany automatycznie przez analizę przychodzących zapytań. Rozwiązanie to jednak powoduje spore komplikacje architektury systemu.

6.2.2 Rozsyłane dane

Są dwa sposoby realizacji replikacji:

- *Row-based* – przesyłane są zmiany dokonane na poszczególnych wierszach. Inne węzły nie muszą powtarzać obliczeń lub np. zagnieżdżonych zapytań SELECT, dzięki czemu są mniej obciążone. Jednak modyfikacjach wielu wierszy trzeba przesłać znaczną ilość danych.
- *Statement-based* – przesłane są zapytania. Ta replikacja może źle działać jeśli wykorzystuje się funkcje niedeterministyczne (MySQL w szczególny sposób obsługuje funkcje RAND i pobierające bieżący czas). Wymaga przesłania mniejszej niż replikacja *row-based* ilości danych jeśli zapytanie modyfikuje na wiele wierszy.

Wybraliśmy replikację *row-based*. W MySQL'u jest ona zaimplementowana z użyciem logu binarnego przechowującego nowe wartości kolumn wiersza.

6.2.3 Zapisy

Rozważaliśmy kilka sposobów implementacji zapisów do bazy danych. Zostały one omówione w kolejnych punktach.

W pełni rozproszone

Każdy węzeł może zapisywać do dowolnej tabeli, której replikę posiada. Zmiany są rozsyłane do wszystkich pozostałych węzłów, które też mają repliki danej tabeli, z użyciem totalnie uporządkowanej komunikacji. Następnie używany jest protokół 2PC do sprawdzenia, czy na żadnym węźle nie występuje konflikt z inną transakcją (być może połączonego z blokowaniem odpowiednich wierszy). Jeśli nie ma konfliktów – transakcja jest zatwierdzana, jeśli są – transakcja musi zostać wycofana.

Aby zmnniejszyć ilość konfliktów możnaby użyć blokowania wierszy, ale to wymagałoby dodatkowej komunikacji z grupą węzłów replikujących daną tebelę i mogłoby być zupełnie nieopłacalne.

Rozwiązanie to jest najbardziej skomplikowane, wymaga znacznej komunikacji między węzłami (protokół 2PC). Mogą się pojawić konflikty wynikające ze współbieżnych transakcji zapisu, co można nieco złagodzić ponawiając transakcję, która nie powiodła się z powodu konfliktu. Jest również konieczność scalania modyfikacji własnych i wykonanych przez inne węzły.

Podejście to oferuje potencjalnie najwyższą wydajność, jeśli współbieżne zapisy rzadko dotyczą tych samych wierszy. Wymaga jednak obsługi 2PC przez engine (InnoDB ją posiada).

Częściowo scentralizowane

Każda tabela posiada wyznaczony węzeł (mastera), który ma prawo zapisywać do niej. Pozostałe węzły mogą tylko czytać daną tabelę. Master rozsyła modyfikacje do pozostałych węzłów.

Podstawową wadą tego rozwiązania jest to, że trudno w nim zrealizować transakcje, które modyfikują wiele tabel naraz, gdyż wymaga to albo użycia rozproszonych transakcji albo tego, żeby masterem kilku tabel był jeden węzeł. Zmiana mastera jest dość długotrwała i skomplikowana (konieczność dokończenia lub przerwania wszystkich transakcji na aktualnym masterze, w między czasie trzeba gdzieś buforować nowe żądania), a mogłaby być w niektórych zastosowaniach konieczna często (np. gdyby były transakcje modyfikujące tabele A i B, i inne modyfikujące A i C) lub doszłoby do sytuacji, że masterem większości tabel byłby ten sam węzeł (sytuacja podobna do rozwiązania scentralizowanego).

Zaletą tego rozwiązania jest to, że jest stosunkowo proste, nie wymaga 2PC jeśli nie stosuje się kluczy obcych, nie ma konfliktów między zapisami. Może ono działać przy replikacji częściowej. Jednak ograniczenie transakcji do modyfikacji tylko jednej tabeli mogłoby być zbyt restrykcyjne.

Scentralizowane

Jest jeden wybrany (np. algorytmem elekcji) węzeł, który obsługuje wszystkie zapisy do wszystkich tabel. W gruncie rzeczy jest to nieco zmodyfikowana zwykła replikacja z jednym węzłem master, który może być zmieniany dynamicznie po awarii. Logi zmian muszą być wysyłane atomowym rozgłaszaniem, aby w przypadku awarii mastera wszystkie węzły otrzymały ten sam ciąg logów.

Rozwiązanie to jest najprostsze koncepcyjnie i implementacyjnie. Nie ma konieczności rozwiązywania konfliktów między współbieżnymi transakcjami, gdyż są one obsługiwane wewnętrznie przez samą bazę danych.

Wybór

Zdecydowaliśmy się na rozwiązanie scentralizowane ze względu na jego prostotę oraz przyjęte założenie, że zapisów nie będzie zbyt wiele.

6.2.4 Dynamiczne zmiany struktury systemu

Aby zapewnić możliwość dołączania nowych węzłów oraz odporność na uszkodzenia trzeba zaimplementować dynamiczną zmianę struktury systemu - dodawanie i usuwanie węzłów oraz wybór węzła mastera.

Awarie węzłów

Węzły co jakiś czas będą się pingować. Jeśli węzeł ulegnie awarii, to musi zostać usunięty przez mastera z listy węzłów slave.

Jeśli któryś z węzłów wykryje awarię mastera, to rozpocznie proces elekcji nowego mastera. Kryterium może być na przykład PID procesu lub numer węzła w klastrze (jeśli na każdym działa co najwyżej jeden proces).

Dołączanie nowego węzła

Dołączanie nowego węzła powinno odbywać się online, tzn. bez przerywania normalnej pracy systemu.

Jeśli założymy, że każda tabela ma co najmniej 3 repliki, to można zastosować następujący scenariusz bez tworzenia *single point of failure* (replika każdej tabeli jest tworzona osobno):

1. Wybierz jeden z węzłów przechowujących tabelę (nie będący masterem) i oznacz go jako źródło replikacji tej tabeli. Powoduje do zablokowanie tabeli na tym węźle w bieżącym stanie, żadne modyfikacje nie są wprowadzane. Jednak logi zmian są przez niego zbierane.

2. Utwórz migawkę tej tabeli i prześlij ją do nowego węzła. Po utworzeniu migawki i wprowadzeniu zbuforowanych logów binarnych węzeł źródłowy może wrócić do normalnego przetwarzania.

3. Utwórz tabelę na podstawie otrzymanej migawki na nowym węźle.

4. Pobierz zmiany, które zaszły od czasu utworzenia migawki i wprowadź je do tabeli. Można je wziąć od węzła źródłowego lub w sposób taki, jak przy dołączaniu po awarii, lub też nowy węzeł sam może zarejestrować się, że chce odbierać logi zmian tej tabeli (na początku replikacji).

5. Poinformuj innych o tym, że replikacja się zakończyła i zreplikowana tabela na tym węźle działa normalnie.

Węzeł zaczyna normalne działanie (wykonywanie zapytań) po zreplikowaniu całej bazy danych.

Dołączanie węzła, który uległ awarii

Gdy jakiś węzeł ulegnie awarii, a potem, po jakimś czasie, zostanie znów uruchomiony, musi pobrać modyfikacje, które zaszły w tabelach w czasie gdy był wyłączony. Dlatego węzeł musi przechowywać w sposób odporny na awarię bieżący stan replikacji – które elementy logu binarnego zostały już przez niego wprowadzone.

Binrane logi nie mogą być przechowywane w nieskończoność ze względu na oszczędność miejsca na dysku. Logi będzie przechowywał tylko węzeł master.

Kolejno są synchronizowane wszystkie tabele. Jeżeli w systemie są jeszcze logi zmian dla tabeli, to należy je pobrać. (trzeba będzie je zablokować, aby nie zostały skasowane przez normalną procedurę wyrzucania starych logów.) Następnie należy wprowadzić te zmiany.

Należy także uwzględnić bieżące zmiany, które zaszły w trakcie przetwarzania starych logów.

6.2.5 Partycjonowanie tabel

Rozważaliśmy użycie partycjonowania tabel – podziału wierszy tabeli między węzły (podobnie jak ma to miejsce w NDB), ale byłoby to nieprzydatne w naszym rozwiązaniu, a poza tym wprowadzałoby dodatkowe komplikacje i opóźnienia (konieczność komunikacji w celu wykonania zapytania). Pociągałoby to też za sobą konieczność bardzo znacznych modyfikacji kodu InnoDB.

Z powyższych powodów zrezygnowaliśmy z partycjonowania.

6.2.6 Inne cechy

Brak ograniczeń MySQL Cluster:

- Nie będzie ograniczeń co do stosowania więzów integralności, w szczególności będzie można używać kluczy obcych. W MySQL Cluster klucze obce nie są zaimplementowane.
- Brak konieczności przechowywania całej bazy danych w pamięci RAM dzięki użyciu InnoDB.

6.3 Implementacja

Ta sekcja przedstawia nasze plany dotyczące implementacji.

6.3.1 Serwer proxy

Nie planujemy utworzenia specjalnego serwera proxy.

Każdy z serwerów MySQL będzie w pewnym sensie takim serwerem, będzie się sam komunikował z innymi węzłami. Dzięki temu unikniemy wprowadzania dodatkowych opóźnień transmisji przez dodatkowy punkt (proxy), przez który musiałyby przechodzić wszystkie zapytania i odpowiedzi. W naszym rozwiązaniu klient zewnętrzny będzie się łączył dzięki mechanizmowi Virtual IP bezpośrednio z jednym z serwerów, który będzie wykonywał transakcje niemodyfikujące danych i odsyłał odpowiedź bez żadnych pośredników. Transkacje modyfikujące dane będą przekazywane do węzła mastera, a następnie odpowiedź od niego zostanie przesłana do klienta.

Zaletą serwera proxy jest możliwość elastycznego równoważenia obciążeń oraz (przy częściowej replikacji) kierowanie zapytań do odpowiednich węzłów na podstawie tabel użytych w zapytaniu.

6.3.2 Log binarny

Węzeł master będzie rozsyłał do pozostałych węzłów swój log binarny. Od węzłów slave będzie otrzymywał informacje o bieżącej pozycji logu, która została już przetworzona. Na podstawie tej informacji master będzie mógł stwierdzić, czy można już wysłać potwierdzenie zatwierdzenia transakcji oczekujących na replikację.

Do rozsyłania logu binarnego planujemy użyć standardowej procedury replikacji udostępnianej przez MySQL, odpowiednio dostosowanej do naszych potrzeb (odsyłanie bieżącej pozycji logu, zmiana węzła master po awarii, zmiana węzła slave w master).

6.3.3 Uruchamianie węzłów

Pierwszy uruchomiony węzeł będzie węzłem master. Kolejne uruchamiane procesy będą poszukiwać mastera - sprawdzą, czy jest utworzony pewien obiekt w systemie plików. Po dołączeniu się będą pobierać listę wszystkich węzłów.

6.3.4 Komunikacja

Do komunikacji między węzłami użyjemy POSIX'owych kolejek wiadomości. Skorzystamy z tego, że w OpenSSI wszystkie obiekty IPC są dostępne w całym klastrze. Dzięki temu komunikacja wygląda tak, jakby klaster był pojedynczą maszyną.

Każdy węzeł będzie miał swoją kolejkę (lub kilka kolejek dla różnych celów).

6.3.5 Bezpieczeństwo

Aby uprościć implementację założyliśmy, że jedynym użytkownikiem w bazie danych będzie *root*, który będzie miał puste hasło. Nie ma jednak przeszkód, aby system działał także z innymi kontami użytkowników.

Rozdział 7

Tworzenie paczki MySQL z rozszerzeniem NDB Cluster

Łukasz Andrzej Bartnik

7.1 Informacje wstępne

Konfiguracja i tworzenie paczki z rozszerzeniem NDB Cluster bazuje na procedurze opracowanej na potrzeby fazy pierwszej - tworzenia zwykłej paczki MySQL. Pominięte zostały opcje związane z poprzednią fazą, a dokument przedstawia przede wszystkim zagadnienia dotyczące fazy drugiej.

7.2 Przygotowanie źródeł

Do budowy paczki nr 2 użyć można źródeł przygotowanych dla pierwszej paczki. W tym celu należy wydać polecenie *make distclean*. Można również pobrać oryginalne źródła MySQL z repozytorium SubVersion.

7.3 Konfiguracja

Tak jak poprzednio, należy wydać polecenie *configure*. Do opcji opisanych dla paczki nr należy dodać *–with-ndbcluster*. Opcja ta włącza kompilację rozszerzenia dla fazy drugiej.

7.4 Kompilacja i instalacja

Tak jak przy przygotowywaniu paczki nr 1, jeśli program *configure* zakończy się bez błędów, należy wykonać polecenia:
make, które skompiluje wymagane pliki i utworzy pliki wykonywalne oraz
make install, które zainstaluje MySQL w katalogu podanym w opcji *–prefix* dla configure.

Paczkę binarną tworzy się z zawartości docelowego katalogu instalacyjnego.

7.5 Automatyzacja

W repozytorium znajduje się zmodyfikowana wersja skryptu magetgz.sh, która automatyzuje wykonanie powyższych poleceń, włączając jednocześnie kompilację NDB Cluster. Jako wynik pracy w katalogu z instalacją binarną powstaje plik mysql.tgz. Wszelkie informacje konfiguracyjne pojawiające się na ekranie podczas pracy skryptu maketgz.sh są zapisywane do odpowiednich plików dziennika o nazwach zaczynających się na maketgz.

7.6 Uruchomienie klastra

W repozytorium znajduje się zestaw skryptów, które automatyzują przygotowanie środowiska uruchomieniowego dla MySQL oraz umożliwiają włączanie i wyłączanie wszystkich węzłów jako jednej usługi.
W pliku *config.sh* znajdują się przypisania wartości do zmiennych, z których skrypt *init.sh* generuje pliki konfiguracyjne. Na podstawie wartości tych zmiennych przygotowywane są także katalogi z bazą danych dla każdego węzła klastra.
Wynikiem pracy skryptu *init.sh* jest zestaw plików konfiguracyjnych, których używają instancje MySQL wywoływane przez skrypt *service* oraz katalogi rso5 tworzone w /cluster/nodeXX/tmp, w których utworzone są kopie domyślnej bazy danych MySQL. Skrypt *service* przyjmuje jako parametr jedną z wartości: start, stop, ndbmgm lub mysql. Ich znaczenie jest następujące:

- *start* - uruchamia węzeł ndbmgmd, czyli proces zarządzający klastrem, następnie pewną, określoną przez pliki konfiguracyjne, ilość węzłów ndbd (węzły danych), a na koniec proces mysqld (węzeł SQL, który pośredniczy w komunikacji z klientem MySQL)

- *stop* - wysyła do procesu zarządzającego polecenie zatrzymania klastra; wywołanie tego polecenia kończy pracę wszystkich procesów klastra

- *ndbmgm* - uruchamia program ndbmgm i przekazuje mu konsolę; program ten pozwala na konfigurację klastra w trakcie jego pracy oraz przeglądanie podstawowych danych o przebiegu jego pracy

- *mysql* - uruchamia proces klienta MySQL, który łączy się do węzła SQL (proces mysqld); pozwala na korzystanie z klastra od strony klienta bazy danych

Dokładny opis znaczenia zmiennych znajduje się w pliku *config.sh*. Przed uruchomieniem skryptu *init.sh* należy sprawdzić wartości przypisane odpowiednim zmiennych. Najważniejsze jest przypisanie odpowiedniej wartości do zmiennych *HOMEDIR* oraz *BASEDIR*. Druga ze zmiennych wskazuje na katalog, w którym znajdują się pliki konfiguracyjne dla *init.sh* oraz *service*.

Część III

Faza III - dokumentacja końcowa

Rozdział 8

Dokumentacja systemu

Marcin Koziej
Michał Skrzędziejewski
Łukasz Andrzej Bartnik
Krzysztof Jamróz
Michał Podsiadłowski

8.1 Wstęp

W ramach projektu z przedmiotu Rozproszone Systemy Operacyjne nasz zespół podjął próbę modyfikacji serwera bazodanowego MySQL w taki sposób, aby działał on w środowisku klastra OpenSSI.

MySQL dostarcza rozwiązanie MySQL Cluster, które nie korzysta z żadnych specyficznych cech OpenSSI. Zaimplementowany w nim silnik NDB jest dość ograniczony, nie obsługuje między innymi kluczy obcych, a dane muszą mieścić się w pamięci operacyjnej (to ostatnie zmieniło się w najnowszej wersji).

Nasze rozwiązanie może współpracować z dowolnym silnikiem przechowującym dane, na przykład z bardzo dobrym, sprawdzonym, posiadającym wiele funkcji silnikiem InnoDB.

8.2 Architektura systemu

W tej sekcji opisujemy architekturę naszego systemu bez wgłębiania się w szczegóły implementacji.

8.2.1 Założenia projektowe

Podstawowe założenia projektowe przyjęte przez nas były następujące:

- Dane muszą być spójne, niedopuszczalne jest powstanie nieścisłości, które musiałyby być rozwiązywane przez użytkownika.
- Dane muszą być trwałe, to znaczy w przypadku awarii dowolnego węzła nie mogą zostać utracone żadne wyniki zatwierdzonych transakcji.

- Dostosowanie aplikacji do korzystania z rozwiązania klastrowego powinno wymagać co najwyżej drobnych zmian w kodzie tej aplikacji.
- Użycie InnoDB jako podstawowego silnika przechowującego dane.

Cechy aplikacji, dla której zaproponowane rozwiązanie powinno się dobrze sprawdzać:

- Znaczna przewaga transakcji odczytu nad transakcjami zapisu.
- Modyfikacje dotyczące naraz niewielkiej ilości wierszy.
- Wymaganie wysokiej dostępności.

8.2.2 Ogólne omówienie

Nasz system działa w środowisku klastrowym, które jest środowiskiem rozproszonym, mimo pewnych przezroczystości oferowanych przez OpenSSI. Dlatego musi on uwzględniać fakt, że system składa się z wielu maszyn, które komunikują się przez sieć i mogą ulegać awariom. Rzutuje to architekturę rozwiązania, która musi być inna niż rozwiązania scentralizowanego działającego na pojedynczym węźle. Na przykład awaria węzła nie może powodować awarii całego systemu, nowe węzły mogą być dołączane w trakcie działania.

Z powyższych przyczyn musieliśmy rozważyć dodatkowe kwestie, między innymi związane z replikacją i modyfikacjami danych oraz ich spójnością i awariami.

8.2.3 Replikacja

Kluczową decyzją było ustalenie sposobu replikacji danych w systemie.

Replikacja w naszym rozwiązaniu jest:

- synchroniczna
- pełna
- oparta o zapytania (*statement-based*)

Synchroniczność

Została użyta replikacja synchroniczna gdyż jest w niej łatwiej, niż przy asynchronicznej, zachować spójność danych.

Przed odesłaniem odpowiedzi do klienta o zatwierdzeniu transakcji, wszystkie modyfikacje muszą zostać rozesłane do wszystkich działających węzłów, które posiadają repliki zmodyfikowanych tabel oraz węzeł mastera musi otrzymać potwierdzenia od węzłów podrzędnych, że dane zostały na nich uaktualnione do odpowiedniego stanu.

Replikacja asynchroniczna oferuje większą wydajność, jednak za cenę niespójności w danych. Jest ona szczególnie dobra w systemach rozległych geograficznie, w których są znaczne opóźnienia transmisji lub gdy część węzłów posiada repliki tylko do odczytu. W środowisku klastrowym czynnik opóźnień transmisji jest niezbyt istotny, gdyż węzły są połączone szybką siecią lokalną.

Równoważność węzłów

Użyliśmy pełnej replikacji, w której wszystkie węzły posiadają kopie wszystkich danych.

Rozwiązanie to jest proste i bardzo wydajne w transakcjach odczytujących dane. Jego wadą jest mała wydajność przy zapisach i gorsza skalowalność w porównaniu do replikacji częściowej (w której węzły posiadają tylko część tabel), gdyż informacje o modyfikacjach muszą być rozesłane do wszystkich węzłów. W efekcie pełna, synchroniczna replikacja może być wolniejsza przy zapisach od zwykłego pojedynczego serwera.

Replikacja cześciowa wymaga podjęcia decyzji o sposobie podziału tabel. Aby uniknąć stosowania transakcji obejmujących wiele węzłów musi on zostać tak zrealizowany, aby zbiór tabel potrzebnych do wykonania każdej transakcji był w całości na jakimś węźle. Podział może być ustalany przez użytkownika lub wykonywany automatycznie przez analizę przychodzących zapytań. Rozwiązanie to jednak powoduje spore komplikacje architektury systemu.

Rozsyłane dane

Są dwa sposoby realizacji replikacji w bazie danych:

- *Statement-based* – przesłane są zapytania. Ta replikacja może źle działać jeśli wykorzystuje się funkcje niedeterministyczne (MySQL w szczególny sposób obsługuje funkcje RAND i pobierające bieżący czas). Wymaga przesłania mniejszej niż replikacja *row-based* ilości danych jeśli zapytanie modyfikuje na wiele wierszy.
- *Row-based* – przesyłane są zmiany dokonane na poszczególnych wierszach. Inne węzły nie muszą powtarzać obliczeń lub np. zagnieżdżonych zapytań SELECT, dzięki czemu są mniej obciążone. Jednak przy modyfikacjach wielu wierszy trzeba przesłać znaczną ilość danych.

Początkowo planowaliśmy użyć replikacji *row-based*, jednak ostatecznie wybraliśmy replikację *statement-based*. Wynikało to z pewnych względów implementacyjnych.

8.2.4 Zapisy

Rozważaliśmy kilka sposobów implementacji zapisów do bazy danych. Zostały one omówione w kolejnych punktach.

W pełni rozproszone

Każdy węzeł może zapisywać do dowolnej tabeli, której replikę posiada. Zmiany są rozsyłane do wszystkich pozostałych węzłów, które też mają repliki danej tabeli, z użyciem totalnie uporządkowanej komunikacji. Następnie używany jest protokół 2PC do sprawdzenia, czy na żadnym węźle nie występuje konflikt z inną transakcją (być może połączonego z blokowaniem odpowiednich wierszy). Jeśli nie ma konfliktów – transakcja jest zatwierdzana, jeśli są – transakcja musi zostać wycofana.

Aby zmnniejszyć ilość konfliktów możnaby użyć blokowania wierszy, ale to wymagałoby dodatkowej komunikacji z grupą węzłów replikujących daną tebelę i mogłoby być zupełnie nieopłacalne.

Rozwiązanie to jest najbardziej skomplikowane, wymaga znacznej komunikacji między węzłami (protokół 2PC). Mogą się pojawić konflikty wynikające ze współbieżnych transakcji zapisu, co można nieco złagodzić ponawiając transakcję, która nie powiodła się z powodu konfliktu. Jest również konieczność scalania modyfikacji własnych i wykonanych przez inne węzły.

Podejście to oferuje potencjalnie najwyższą wydajność, jeśli współbieżne zapisy rzadko dotyczą tych samych wierszy. Wymaga jednak obsługi 2PC przez engine (InnoDB ją posiada).

Częściowo scentralizowane

Każda tabela posiada wyznaczony węzeł (mastera), który ma prawo zapisywać do niej. Pozostałe węzły mogą tylko czytać daną tabelę. Master rozsyła modyfikacje do pozostałych węzłów.

Podstawową wadą tego rozwiązania jest to, że trudno w nim zrealizować transakcje, które modyfikują wiele tabel naraz, gdyż wymaga to albo użycia rozproszonych transakcji albo tego, żeby masterem kilku tabel był jeden węzeł. Zmiana mastera jest dość długotrwała i skomplikowana (konieczność dokończenia lub przerwania wszystkich transakcji na aktualnym masterze, w między czasie trzeba gdzieś buforować nowe żądania), a mogłaby być w niektórych zastosowaniach konieczna często (np. gdyby były transakcje modyfikujące tabele A i B, i inne modyfikujące A i C) lub doszłoby do sytuacji, że masterem większości tabel byłby ten sam węzeł (sytuacja podobna do rozwiązania scentralizowanego).

Zaletą tego rozwiązania jest to, że jest stosunkowo proste, nie wymaga 2PC jeśli nie stosuje się kluczy obcych, nie ma konfliktów między zapisami. Może ono działać przy replikacji częściowej. Jednak ograniczenie transakcji do modyfikacji tylko jednej tabeli mogłoby być zbyt restrykcyjne.

Scentralizowane

Jest jeden wybrany (np. algorytmem elekcji) węzeł, który obsługuje wszystkie zapisy do wszystkich tabel. W gruncie rzeczy jest to nieco zmodyfikowana zwykła replikacja z jednym węzłem master, który może być zmieniany dynamicznie po awarii. Logi zmian muszą być wysyłane atomowym rozgłaszaniem, aby w przypadku awarii mastera wszystkie węzły otrzymały ten sam ciąg logów.

Rozwiązanie to jest najprostsze koncepcyjnie i implementacyjnie. Nie ma konieczności rozwiązywania konfliktów między wpółbieżnymi transakcjami, gdyż są one obsługiwane wewnętrznie przez samą bazę danych.

Wybór

Zdecydowaliśmy się na rozwiązanie scentralizowane ze względu na jego prostotę oraz przyjęte założenie, że zapisów nie będzie zbyt wiele.

Zamiast atomowego rozgłaszania użyliśmy nieco zmodyfikowanej replikacji dostępnej w MySQL.

8.2.5 Dynamiczne zmiany struktury systemu

Aby zapewnić możliwość dołączania nowych węzłów oraz odporność na uszkodzenia została zaimplementowana dynamiczna zmiany struktury systemu - dodawanie i usuwanie węzłów oraz wybór węzła mastera (to ostatnie nie zostało zaimplementowane).

Awarie węzłów

Węzły co jakiś czas powinny się pingować. Jeśli węzeł ulegnie awarii lub zostanie wyłączony, to musi zostać usunięty przez mastera z listy węzłów slave.

Jeśli któryś z węzłów wykryje awarię mastera, to rozpocznie proces elekcji nowego mastera. Kryterium może być na przykład PID procesu lub numer węzła w klastrze (jeśli na każdym działa co najwyżej jeden proces) albo stan zaawansowania replikacji (który węzeł ma najbardziej aktualne dane). W trakcie tego procesu wszystkie transakcje modyfikujące dane powinny zostać wstrzymane.

Dołączanie nowego węzła

Dołączanie nowego węzła powinno odbywać się online, tzn. bez przerywania normalnej pracy systemu.

Jeśli założymy, że każda tabela ma co najmniej 3 repliki (przy replikacji częściowej), lub gdy są co najmniej 3 węzły (przy replikacji pełnej) to można zastosować następujący scenariusz bez tworzenia *single point of failure* (replika każdej tabeli jest tworzona osobno):

1. Wybierz jeden z węzłów przechowujących tabelę (nie będący masterem) i oznacz go jako źródło replikacji tej tabeli. Powoduje do zablokowanie tabeli na tym węźle w bieżącym stanie, żadne modyfikacje nie są wprowadzane. Jednak logi zmian są przez niego zbierane.

2. Utwórz migawkę tej tabeli i prześlij ją do nowego węzła. Po utworzeniu migawki i wprowadzeniu zbuforowanych logów binarnych węzeł źródłowy może wrócić do normalnego przetwarzania.

3. Utwórz tabelę na podstawie otrzymanej migawki na nowym węźle.

4. Pobierz zmiany, które zaszły od czasu utworzenia migawki i wprowadź je do tabeli. Można je wziąć od węzła źródłowego lub w sposób taki, jak przy dołączaniu po awarii, lub też nowy węzeł sam może zarejestrować się, że chce odbierać logi zmian tej tabeli (na początku replikacji).

5. Poinformuj innych o tym, że replikacja się zakończyła i zreplikowana tabela na tym węźle działa normalnie.

Węzeł zaczyna normalne działanie (wykonywanie zapytań) po zreplikowaniu całej bazy danych.

Dołączanie węzła, który uległ awarii

Gdy jakiś węzeł ulegnie awarii, a potem, po jakimś czasie, zostanie znów uruchomiony, musi pobrać modyfikacje, które zaszły w tabelach w czasie gdy był wyłączony. Dlatego każdy węzeł przechowuje w sposób odporny na awarię bieżący stan replikacji – które elementy logu binarnego zostały już przez niego wprowadzone.

Binrane logi nie mogą być przechowywane w nieskończoność ze względu na oszczędność miejsca na dysku. Logi przechowuje tylko węzeł master.

8.2.6 Partycjonowanie tabel

Rozważaliśmy użycie partycjonowania tabel – podziału wierszy tabeli między węzły (podobnie jak ma to miejsce w NDB), ale byłoby to nieprzydatne w naszym rozwiązaniu, a poza tym wprowadzałoby dodatkowe komplikacje i opóźnienia (konieczność komunikacji w celu wykonania zapytania). Pociągałoby to też za sobą konieczność bardzo znacznych modyfikacji kodu InnoDB.

Z powyższych powodów zrezygnowaliśmy z partycjonowania.

8.2.7 Inne cechy

W naszym rozwiązaniu nie ma ograniczeń, które występują w MySQL Cluster:

- Można stosować wszystkie dostępne rodzaje więzów integralności, w szczególności można używać kluczy obcych. W MySQL Cluster klucze obce nie są zaimplementowane.
- Brak konieczności przechowywania całej bazy danych w pamięci RAM dzięki użyciu InnoDB.

8.3 Implementacja systemu

Implementację systemu poprzedziło gruntowne rozpoznanie źródeł MySQL. Zidentyfikowaliśmy funkcje i metody oraz struktury danych odpowiedzialne za wykonanie zapytania, replikację, przesyłanie logu binarnego. Rozszerzenie MySQL miało dwojaką postać: dodaliśmy kilka nowych klas (z główną klasą *RSO_Neigh*) oraz wprowadziliśmy zmiany w kilku metodach z implementacją algorytmów replikacji i wykonania zapytania SQL. Przygotowana klasa udostępniała interfejs do wykonania konkretnych zadań w ramach klastra:

- uruchomienie wątku obsługującego kolejkę wiadomości (IPC)
- zgłoszenia węzła podrzędnego (*slave*) do węzła głównego (*master*)
- uruchomienie wątku obsługującego zapytania (wykonującego zapytania) pochodzące od węzłów podrzędnych - zapytania zapisu
- wewnętrzna synchronizacja wątku kolejki wiadomości IPC i wątku wykonującego zapytania na strukturze danych przechowującej zapytania zapisu oczekujące na wykonanie
- synchronizacja pomiędzy węzłem głównym i węzłami podrzędnymi podczas replikacji (semafory IPC)

8.3.1 Komunikacja pomiędzy węzłami

W naszym rozwiązaniu komunikacja pomiędzy poszczególnymi węzłami służy dwóm celom: przesłaniu wymaganych danych (zgłoszenia o nowym węźle podrzędnym do węzła koordynatora) oraz synchronizacji wykonywanych operacji (potwierdzeniu replikacji danych na danym węźle podrzędnym). W pierwszym przypadku dla każdego węzła tworzona jest kolejka wiadomości - standardowy mechanizm IPC zgodny z POSIX. Do synchronizacji wątków z różnych procesów używamy semaforów - także

zgodych z POSIX. Wybraliśmy te rozwiązania z uwagi na ich przeźroczystą implementację w OpenSSI - poszczególne węzły, znajdujące się na różnych maszynach nie implementują komunikacji sieciowej i posiadają ten sam obraz utworzonych kolejek i semaforów. Dostarczenie wiadomości i opearcje na semaforze gwarantuje warstwa komunikacji z OpenSSI.

Kolejki komunikatów

Dla każdego węzła tworzona jest kolejka komunikatów. Aby zapewnić dostępność kluczy/identyfikatorów kolejek w całym systemie, korzystamy ze standardowej funkcji *ftok*, dla której ustalamy pewien plik jako *pathname*, zaś jako wartość *proj_id* podajemy numer węzła. Oznacza to, że na każdej maszynie można uruchomić tylko jeden proces MySQL, przy czym uruchamianie większej ich ilości wydaje się być pozbawione celu.

Zmodyfikowana replikacja

Do synchronizacji zawartości baz danych wykorzystaliśmy dostępny w MySQL mechanizm replikacji poprzez rozsyłanie logu binarnego. Wymagał on jednak modyfikacji, aby zapewnić synchroniczność replikacji. W tym celu dla każdego wątku zlecającego wykonanie zapytania tworzymy semafor, generując klucz na podstawie ustalonego pliku oraz numeru wątku, które wydają się być unikalne w ramach całego klastra. Bezpośrednio po przesłaniu zapytania do logu binarnego, a przed odesłaniem klientowi komunikatu o wykonaniu zapytania, wątek zawiesza się w oczekiwaniu na obniżenie semafora o wartość równą ilości zarejestrowanych węzłów podrzędnych. Każdy węzeł podrzędny otrzymuje numer wątku jako job_id (dzięki tej metodzie podczas replikacji możliwe jest tworzenie unikalnych tabel tymczasowych) i na jego podstawie generuje numer semafora. Po udanej replikacji zwiększa wartość o 1. Gdy wszystkie wątki podrzędne zreplikują zapytanie, węzeł główny opuści semafor, zakończy oczekiwanie i prześle do klienta komunikat o zakończeniu obsługi zapytania.

8.3.2 Istotne pliki źródłowe MySQL

Modyfikowane przez nas pliki znajdowały się w katalogu *sql/* w źródłach MySQL.

sql_class.h i sql_class.cc

Zawierają definicję struktury THD, która jest tworzona dla każdego wątku. Przechowuje ona bardzo różne dane, jest to swojego rodzaju „worek” na globalne zmienne wątku. Obiekt ten jest przekazywany do większości funkcji.

My dodaliśmy:

- *ulong THD::rso_job_id*: Oznacza numer (thread_id) wątku serwera, który zapisał zdarzenie do logu binarnego. Zmienna ta jest ustawiana w konstruktorze klasy THD na 0, natomiast przy odczycie z logu binarnego przez slave'a, ustawiana jest w jego strukturze THD na numer wątku po stronie serwera, który zapisał odczytywane przez slave'a zdarzenie.

  Zmienna ta jest wykorzystywana jako argument metod *RSO_Neigh::wait()* i *RSO_Neigh::post()*, przede wszystkim aby zidentyfikowac semafor, do którego odnosi się wywołanie metody. Wartość zerowa oznacza, iż w bieżącym wątku

nie jest uruchamiane zapytanie pochodzące od master'a (Zapytanie od podłączonego użytkownika, nie z wątku obsługującego log binarny).

- *RSO_THD_data *rso_ptr* – jest to obiekt klasy stworzonej przez nas, który przechowuje istotne dla naszej implementacji informacje. Zdecydowaliśmy się na utworzenie osobnej zmiennej, gdyż od struktury *THD* zależą praktycznie wszystkie pliki i jakakolwiek jej modyfikacja wiązała się z koniecznością rekompilacji całego serwera.
- Funkcja *RSO_runQuery* implementuje wątek mastera, którego zadaniem jest wykonywanie zapytań modyfikujących dane przekazanych przez węzły slave. Jest to zmodyfikowany zwykły wątek obsługujący połączenie z klientem. Usunęliśmy komunikację sieciową, która nie jest w nim potrzebna, a zamiast tego czeka on na wewnętrznej kolejce. Pominięcie standardowego *handshake'u* z protokołu wymagało ręcznego ustalenia uprawnień tego wątku. Działa on jako użytkownik root (w bazie danych).

sql_parse.cc

Zawiera kod funkcji przetwarzających zapytanie po otrzymaniu go od klienta, z których najistotniejsza jest *mysql_execute_command*. Parsuje ona zapytanie i na podstawie jego typu przekazuje je do wykonania w odpowiedniej funkcji.

Nasza modyfikacja polegała na sprawdzeniu czy zapytanie może zmodyfikować dane (czyli czy jest to INSERT, UPDATE albo DELETE). Jeżeli tak jest, a zapytanie wykonywane jest na węźle slave w zwykłym wątku obsługującym klienta oraz serwer nie jest w trybie inicjalizacji (*bootstrap* – używany w trakcie inicjalizacji bazy danych przez skrypt *mysql_install_db*) to jest ono przekazywane do węzła master, a wątek następnie oczekuje na odpowiedź od niego. Powinna ona zostać przekazana do klienta, ale nie zdążyliśmy tego zaimplementować, więc zawsze jest przekazywana informacja o powodzeniu.

mysqld.cc

Zawiera implementację metody *main* serwera, zawiera kod przetwarzający parametry wywołania, inicjujący różne struktury wewnętrzne, tworzący wątki komunikacyjne i inne.

Do tego pliku dodaliśmy obsługę naszego parametru: *--rso-master* oraz wywołania inicjujące nasze rozszerzenia, między innymi tworzące obiekt *RSO_Neigh* i wątek obsługujący zapytania przekazane przez węzły slave.

slave.cc

Plik ten zawiera implementacje wątków węzła slave przy replikacji (odbierającego log binarny i wykonującego zapisane w nim zdarzenia).

Dodaliśmy ustawienie specjalnego oznaczenia wątku wykonującego zapytania SQL przekazane w logu binarnym od mastera ($thd->rso\_ptr->isSlaveSQL$), gdyż te zapytania muszą być wykonane na węźle, a nie przekazane do mastera.

Wątek slave'a informuje również mastera o tym, które zdarzenia z logu już wykonał. Używa w tym celu naszej metody *RSO_Neigh::post*.

log.cc

log_event.cc

Zawiera implementacje klas różnych zdarzeń, które są zapisywane do logu binarnego.

Nasza modyfikacja umożliwia węzłowi slave zidentyfikowanie tego, od kogo dane zdarzenie pochodzi i później odpowiednie powiadomienie.

protocol.h i protocol.cc

Zawierają implementacje funkcji i klas obsługujących komunikację serwera z klientem, zwracanie wyników i informacji o sukcesie operacji lub błędzie.

Ich modyfikacja byłaby potrzebna do implementacji zwracania wyniku zapytania do węzła slave.

8.3.3 Dyskusja nad innymi możliwymi rozwiązaniami

Podczas pracy nad naszym rozwiązaniem, rozważane były różne metody implementacji poszczególnych elementów systemu.

Należało rozważyć, w jaki sposób prowadzona będzie komunikacja między węzłami. Chodzi tu zarówno o przesyłanie komunikatów sterujących (np. informacja o podłączeniu nowego węzła typu slave) jak i zapytań SQL (sytuacja ta występuje, gdy następuje zapis do węzła typu slave i musi on przesłać treść tego zapytania do węzła typu master).

Rozważaliśmy następujące metody komunikacji

- MPI
- ICS
- Mechanizmy IPC OpenSSI

Standard MPI (Message Passing Interface) okazał się nieadekwatny dla naszego rozwiązania. Jest on zbyt skomplikowany dla naszych zastosowań, nadaje się głównie do przeprowadzania rozproszonych obliczeń. Ponadto MPI nie jest zaprojektowane do przetrwania awarii nawet jednego węzła.

Rozważaliśmy również użycie ICS (Internode Communication Subsystem). Okazało się jednak, że jest to system komunikacji kernel-to-kernel, oferujący komunikację nie na tym poziomie, którego potrzebowaliśmy.

Ostatecznie do komunikacji wybrane zostały mechanizmy IPC OpenSSI. Są one przede wszystkim przezroczyste i łatwe w użyciu. Użycie semafora czy kolejki wiadomości wygląda dokładnie tak samo jak w każdym systemie zgodnym ze standardem POSIX, nie trzeba w żaden specjalny sposób określać, iż obiekt IPC jest obiektem rozproszonym. Ma to tę dodatkową zaletę, iż można testować system na jednym komputerze.

8.4 Opis uruchamiania i konfiguracji systemu

W skład 3 paczki wchodzi zestaw skryptów, które służą do przygotowania środowiska i uruchomienia programu. Przygotowanie środowiska polega na usunięciu ew. pozostałości po poprzednich uruchomieniach, które nie powinny być widoczne w klastrze

oraz zainicjowaniu katalagów przechowujących bazy danych dla poszczególnych węzłów. Wykorzystywana jest tutaj własność OpenSSI, dzięki której każdy proces przy jednakowym obrazie systemu plików dostęp do niektórych folderów realizuje lokalnie (na tym samym węźle, na którym pracuje). Wszystkie procesy MySQL utrzymują swoją bazę danych w katalogu */tmp/rso5/var*, i właśnie te katalogi (na wszystkich dostępnych węzłach) inicjalizuje skrypt *init.sh*.

Skrypt *service* umożliwia uruchomienie procesów klastra, zamknięcie ich oraz uruchomienie programu klienta. Po zainstalowaniu MySQL w katalogu z binariami (w katalogu głównym, zawierającym min. katalog bin, var) należy umiścić katalog scripts, a w nim zawartość katalogu rso-scripts ze źródeł MySQL. Następnie w katalogu scripts należy uruchomić *init.sh*, który przygotuje wszystkie wymagane katalogi. Po tym można uruchamiać usługę za pomocą *service start*.

8.5 Wyniki testów

Podstawą naszych testów była specjalnie napisana aplikacja w c++ oparta na bibliotece mysql++. Skrypt sql-bench nie mogł zostać użyty ze względu na stopień ukończenia naszego rozwiązania - brak proxy które rozdzielałoby zapytania do róznych węzłów. Co za tym idzie aplikacja testująca musiała sama łaczyć się z poszególnymi węzłami i odptywać je równomiernie. Mysql NDB było testowane również tym programem jedynie z tą różnicą, że łaczenie się z bazą danych było wielowątkowe ale na adres głównego węzła. Aplikacja testująca zliczała czas od zadania zapytania do otrzymania odpowiedzi dla każdego wątku z osobna. Plik zawierający kwerendy był dzielony pomiędzy wszytkie wątki równomiernie. Do testów został użyty zestaw kwerend z testu sql-bench test-ATIS. Ilość wątków była dobrana do ilości węzłów bazy danych - 1, 2, 3 i 4 wątki na serwer.

Wyniki testów

RSO Threads	Queries	Maxtime	MinTime	AvgTime	TotalThd	Q/s/thd
3	29800	71,988	65,855	67,923	203,77	146,243
6	29800	74,494	60,298	65,161	390,97	76,228
7	29800	81,989	50,637	66,806	467,648	63,723
9	29800	70,604	36,4748	58,259	524,335	56,864
12	29800	71,813	45,959	63,360	760,331	39,261
NDB Threads	Queries	Maxtime	MinTime	AvgTime	TotalThd	Q/s/thd
3	29800	92,753	92,059	92,398	277,195	107,505
6	29800	92,735	88,857	91,216	547,296	54,449
9	29800	89,878	79,593	81,775	735,978	40,947
12	29800	93,503	76,687	86,265	1035,19	28,788

Jak widać w tabelce nasze rozwiązanie przy przetwarzaniu zapytań typu select jest szybsze niz NDB. Wiąże się to również ze specyfiką testów gdzie wiele zapytań przetwarzało w zagnieżdżonych zapytaniach całe tabele. NDB ze wzgledu na wertykalne rozproszenie tabel musi przesylać dane przez sieć co przy zapytaniach, które wymagają przesłania całych rzedów jest bardzo czasochłonne. Nasza baza danych

nie wprowadza tego opóźnienia ze względu na pełna replikację. Przy zwiekszaniu obciążenia (zwiekszanie ilości wątków) w NDB i RSO nie widać spadku wydajności. Zapytania przetwarzane są mniej wiecej z taka samą szybkościa co widać po średnim czasie wątku. W celu pełnego wykorzystania możliwości bazy ważne jest równomierne obciążenie węzłów widać to przy teście gdzie na bazie RSO zostało uruchomionych 7 wątków. Jeden z serwerów obsługiwał wtedy 3 - a nie 2 watki. Najdłuższy wątek wykonywał się o 30 sekund dłuzej niz najkrótszy. Zatem proxy które rozdziela zapytania powinno dostawać od serwerów stan ich obciążenia by móc efektywnie wykorzystać zasoby.

8.6 Podsumowanie

8.6.1 Otwarte kwestie

W naszym rozwiązaniu nie udało się obsłużyć odłączania węzła głównego. Należałoby zaimplementować jeden z algorytmów elekcji, z ew. warunkami na liczność działającej grupy. W takim przypadku pierwszym krokiem byłoby ustalenie węzła z najnowszą wersją danych i ustalenie go, jak węzła *master*. Być może konieczna byłaby dalsza ingerencja w obsługę logu binarnego tak, aby replikowane zapytania były zapamiętywane w oddzielnym buforze. Po zmianie węzła *master* bufor ten służyłby do replikacji stanu bazy na pozostałe węzły ze starszymi danymi.

Nasz system korzysta z własności silnika InnoDB, ale nie pozwala na przezroczystą obsługę transakcji. Wiąże się to z przesyłaniem zapytań zapisujących do węzła głównego i obsługą zapytań odczytu lokalnie. Taki mechanizm odciąża węzeł główny przy odczytach i pozwala na pełną replikację przez niego sterowaną, ale jednocześnie ukrywa stan transakcji przed innymi transakcjami. Rozwiązaniem mogłoby być sprawdzanie rodzaju zapytań wykonywanych w transakcji: jeśli wystąpiłyby zapytania zapisu, całość wykonywana byłaby wykonywana na węźle głównym. Można też wybrać drogę dalszej modyfikacji *mysqld* i replikowane zmiany udostępniać danej transakcji. Głównym problemem pozostaje tutaj możliwość wycofania danej transakcji z wielu węzłów, jeśli każde zapytanie jest od razu replikowane, lub sprawdzenie wykonalności danego zapytania na wszystkich węzłach, jeśli transakcja wykonywana jest lokalnie.

Aby zwiększyć elastyczność rozwiązań, należałoby przygotować jednolitą warstwę komunikacyjną opartą o kolejki wiadomości IPC, gdzie punktami końcowymi byłyby wątki (z możliwością adresowania w ramach całego klastra).

8.6.2 Stan rozpoznania MySQL

Wykonane rozpoznanie źródeł MySQL pokazuje, że jakkolwiek możliwa jest modyfikacja programy w taki sposób, by zbudować w niego mechanizmy replikacji i synchronizacji z wieloma węzłami, to struktura programu nie wspiera takich modyfikacji, a przyjęte rozwiązania czasem skutecznie ją utrudniają. Omówione poprzednio modyfikacje w funkcjach wykonujących zapytania i rozsyłających log binarny nie powodują zmiany ścieżki wykonania, ale dodają do niej pewne elementy. Bardziej zaawansowane mechanizmy, które można zaimplementować w klastrze z całą pewnością wymagałyby nie tylko dodania nowych metod i struktur danych, ale także zmiany algorytmów obsługi transakcji, buforowania itd. W przypadku transakcji należałoby najprawdopodobniej zejść na poziom silnika InnoDB.

Każda zmiana tego typu pociąga jednak za sobą konieczność sprawdzenia tych elementów MySQL, które zależą od elementów modyfikowanych. Brak dobrej dokumentacji źródeł MySQL powoduje, że niektóre zmiany destabilizują pracę systemu. Dodatkowym problemem są niektóre rozwiązania, na które zdecydowali się twórcy MySQL, a które nie współgrają z koncepcją rozproszonej bazy danych (np. obsługa transakcji). Końcowym efektem jest duża ilość pracy poświęcana na analizę kodu i usuwanie nieumyślnie wprowadzonych usterek.

Przykładem jest próba uruchomienia przetwarzania zapytania sql poprzez wywołanie funkcji, które to wiąże się z koniecznością wcześniejszego zainicjalizowania pól struktury THD związanymi z prawami dostępu. Problemem jest to, iż pola te ustawiane są podczas nawiązywania połączenia między klientem i serwerem, a więc należało emulować to zachowanie.

Biorąc to wszystko pod uwagę, kontynuowanie prac w kierunku przebudowy MySQL może się okazać nieefektywne, a lepszym podejściem być może byłoby dodanie warstwy koordynującej pracę węzłów nad warstwą MySQL; taka warstwa dokonywałaby wstępnej analizy zapytań i sterowała przepływem danych pomiędzy węzłami.

8.6.3 Prace wykonane w ramach projektu

Etap pierwszy

- 18-19.03 – Praca ze środowiskiem OpenSSI
- 26.03 – Praca ze środowiskiem OpenSSI
- 31.03 – Spotkanie zespołu: przedstawienie algorytmów replikacji, dyskusje dotyczące architektury, sposobu zatwierdzania transakcji, czy należy używać węzła–proxy, wstępne ustalenia dotyczące architektury, sformułowanie kwestii do rozpatrzenia.
- 1-4.04 – Prace nad przygotowaniem pierwszego etapu.
- 4.04 – Spotkanie zespołu: ustalenie pytań, które chcieliśmy zadać prowadzącemu, dyskusja dotycząca implementacji kluczy obcych i zapytań przy częściowej replikacji, rozważenie potencjalnych zastosowań systemu.

Etap drugi

- 9.04 – Spotkanie zespołu. Praca z OpenSSI i MySQL
- 25.04 – Spotkanie zespołu. Praca z OpenSSI i MySQL
- 28.04 – Spotkanie zespołu: dalsze ustalenia dotyczące architektury – węzły równorzędne, replikacja częściowa, synchroniczna, row-based, brak kluczy obcych, partycjonowania tabel i zapytań wykonywanych na więcej niż jednym węźle; użycie dodatkowego daemona, który przyjmowałby zapytania od klientów i rozdzielał do odpowiednich węzłów; dyskusja dotycząca sposobu dołączania nowego węzła i takiego, który uległ awarii.
- 13.05 – Spotkanie zespołu: ustalenie zakresu funkcjonalności do zrealizowania w prototypie, modyfikacja ustaleń dotyczących architektury: pełna replikacja z zapisem do jednego węzła, który ma być wybierany algorytem elekcji, rezygnacja z funkcji związanych z bezpieczeństwem, gdyż to komplikowałoby implementację (użycie tylko użytkownika *root* bazy danych)

- 13-15.05 – kikugodzina praca nad kodem mysql w lab 9.

Etap trzeci

- 23.05 - praca nad kodem mysql w lab 9.
- 2-4.06 - spotkania całodniowe. Uruchomienie synchronicznej replikacji.
- 8.06 - praca nad kodem mysql w lab 9.
- 9.06 - praca nad kodem mysql w lab 9.
- 9-12.06 - spotkania całodniowe. Kończenie implementacji przekierowywania zapytań.

8.6.4 Uczestnictwo w realizowanym projekcie

- Marcin Koziej – Koordynator
  - Opracowanie wstępne dotyczące środowiska OpenSSI
  - Uczestnictwo w spotkaniach
  - Praca nad koncepcją rozwiązania
  - Rozpoznawanie funkcjonowania MySQL
- Łukasz Bartnik
  - Przygotowanie demonstracji projektu
  - Przygotowanie paczek i ich dokumentacji
  - Kodowanie
  - Debugging
  - Opracowanie koncepcji rozwiązania
- Krzysztof Jamróz
  - Opracowanie wstępne dotyczące algorytmów realizacji wysokiej dostępności i niezawodności rozproszonych zasobów
  - Zbieranie i przygotowanie dokumentacji
  - Opracowanie koncepcji rozwiązania
  - Kodowanie
  - Debugging
- Michał Skrzędziejewski
  - Administracja repozytorium oraz tworzenie kopii bezpieczeństwa, opracowanie wstępne dotyczące systemu kontroli wersji *Subversion*
  - Rozpracowywanie kodu MySQL
  - Opracowanie koncepcji rozwiązania
  - Kodowanie
  - Debugging

- Michał Podsiadłowski
  - Opracowanie wstępne dotyczące istniejących rozwiązań w ramach MySQL umożliwiających realizację klastrowości, zwiększenia niezawodności i wydajności serwera bazy danych
  - Administracja systemu zarządzania treścią opartego o wiki *MoinMoin*
  - Opracowanie koncepcji rozwiązania
  - Przygotowanie i przeprowadzenie testów

8.7 Dodatkowe uwagi

8.7.1 Analiza kodu MySQL

Podczas wielu godzin spędzonych na analizie kodu MySQL, pomocne okazały się następujące narzędzia:

- gdb: Użycie debuggera pozwoliło na lepsze zaznajomienie się ze sposobem działania MySQL, w szczególności znalezienie odpowiednich funkcji odpowiedzialnych za przetwarzanie zapytania, replikację, itp.

  Podczas debugowania MySQL z poziomu gdb należy pamiętać o uruchomieniu procesu *mysqld* z opcją *--gdb*. Powoduje to między innymi zainstalowanie handlera SIGINT, co pozwala zatrzymywanie serwera MySQL z poziomu gdb poprzez naciśnięcie klawiszy CTRL-C, dodanie np. breakpointów i wznowienie jego pracy.

  Niestety na chwilę obecną praca w gdb wiąże się z ryzykiem zawieszenia klastra.

- Framework DBUG: Jest to zestaw makr języka C, jak sama nazwa wskazuje ułatwiający debugowanie programu. W katalogu źródłowym MySQL znajduje się w podkatalogu *dbug*. Framework udostępnia między innymi następujące makra
  - *DBUG_ENTER(fname)*: Makro używane przy wejściu do funkcji o nazwie fname.
  - *DBUG_RETURN(value)*: Makro zastępujące wyrażenie *return value* z języka c.
  - *DBUG_VOID_RETURN*: Zastępuje wyrażenie return (void) lub po prostu return z języka c.

  Użycie tych makr jest konieczne dla poprawnego utworzenia wewnętrznego stosu wywołań funkcji przez moduł czasu wykonania DBUG'a.

  Dodatkowo, wszystkie wywołania *printf* / *fprintf* dla celów diagnostycznych można zastąpić poprzez wywołanie makra *DBUG_INFO*. Makro to powoduje wypisanie na ekran komunikatu diagnostycznego z pewnej klasy, jednak tylko wtedy gdy DBUG jest uaktywniony.

  Warto zauważyć, iż podejście to ułatwia wyłączenie diagnostyki w wersji wydaniowej oprogramowania. Użycie wyżej wymienionych makr wymaga przekazania parametru *--debug* podczas wywoływania polecenia *configure*, tworzącego

plik *Makefile* używany przy kompilacji MySQL. Brak tej opcji powoduje że nie są one uwzględniane przy kompilacji, co eliminuje niepotrzebny w wersji wydaniowej narzut.

Podczas analizy kodu MySQL najczęściej korzystaliśmy z możliwości DBUG poprzez wywołanie *mysqld* z opcją *-#d:t*. Powoduje to wyświetlenie na ekran stosu wywołań i powrotów z funkcji oraz przydatnych komunikatów diagnostycznych. Oto fragment tego co wypisuje na ekran MySQL z włączoną diagnostyką:

```
| | <my_malloc
| | >simple_open_n_lock_tables
| | | >open_tables
| | | | >init_alloc_root
| | | | | enter: root: 0xbfede780
| | | | | >my_malloc
| | | | | | my: size: 8040 my_flags: 0
| | | | | | exit: ptr: 0x94a31a8
| | | | | <my_malloc
```

Instrukcję obsługi DBUG można znaleźć pod tym adresem:

```
http://www.ba-stuttgart.de/~boehm/
    lib.C/man/dbug.html
```

- cscope: Jest to program ułatwiający przeglądanie kodu źródłowego programów napisanych w c/c++. Wydawany jest na licencji open-source. Program działa w trybie interaktywnym pod konsolą i umożliwia między innymi znalezienie miejsca zdefiniowania danej zmiennej, plików w których użyto poszukiwanej funkcji. Narzędzie tego typu bardzo pomaga przy pracy z oprogramowaniem posiadającym bardzo wiele plików źródłowych, jak MySQL. Na stronie *http://cscope.sourceforge.net/* można pobrać aplikację jak i zapoznać się z jej działaniem na przykładzie kodu źródłowgo jądra linuksa.

- doxygen: Służy głównie do generowania dokumentacji z kodu źródłowego, podobnie jak w javadoc. Nawet jeśli kod źródłowy nie jest udokumentowany, przepuszczenie go przez doxygen wygeneruje zestaw plików HTML który znacząco ułatwia wyszukiwanie w kodzie źródłowym potrzebnych informacji. W szczególności pozwala na szybkie obejrzenie / skok do pól i metod danej klasy.

8.7.2 GDB

Zaobserwowaliśmy zawieszanie klastra w trakcie używania gdb. Wymaga to jeszcze przetestowania, ale wydaje się nam, że zachodzi to, gdy debugowany serwer mysqld jest zamykany w sposób normalny. Komunikaty od gdb świadczą o tym, że jakieś wątki lub procesy niespodziewanie lub za wcześnie zniknęły. Ostateczny efekt jest taki, że nie działają systemowe funkcje operujące na procesach, w szczególności nie jest możliwe uruchomienie żadnego nowego procesu. Jedynym rozwiązaniem jest restart klastra.

8.7.3 Kompilacja

Zastosowane w MySQL rozwiązania implementacyjne wymagają ustawienia pewnych przełączników w kompilatorze (g++, np. *-fno-implicit-templates*). Powoduje to problemy z użyciem wzorców z STL. Rozwiązaliśmy je tworząc własne reguły kompilacji dla pliku rso_neigh.cc, które znajdują się w pliku Makefile.rso. Plik ten jest dołączany w odpowiednim miejscu w głównych pliku Makefile.