ROZPROSZONE SYSTEMY OPERACYJNE

Niezawodne usługi w rozwiązaniach SSI
dokumentacja projektu

Karol Ostrowski

16 czerwca 2005

Spis treści

1 Wstęp

Realizacja niezawodności świadczonych usług stała się jednym z podstawowych zagadnień w tematyce sieci komputerowych. Dotyczy to zwłaszcza takich obszarów jak bankowość, gdzie przerwanie świadczenia usługi może wiązać się z dużymi stratami nie tylko materialnymi (utrata zaufania klientów). Na pewno nie podnosi również prestiżu portalowi internetowemu przerwanie działania z „przyczyn technicznych".

1.1 Kryteria oceny

Podstawowym kryterium oceny niezawodności usługi jest to jak moment potencjalnej awarii odczuje klient połączony z usługą. Możliwe są następujące warianty:

- niewidoczna dla klienta naprawa awarii,
- przerwanie i ponowne wznowienia sesji,
- przerwanie i ponowne wznowienie sesji z utratą danych poprzedniej sesji,
- przerwanie sesji i zawieszenie działania usługi na czas naprawy awarii,

1.2 Rozwiązania

Najbardziej rozpowszechnionym modelem zapewniającym niezawodność świadczenia usług jest model „serwer macierzystym, serwer kopii" w którym dane, pliki konfiguracyjne z serwera podstawowego są powielane na inny (inne) komputery. W przypadku awarii serwera macierzystego następuje „przełączenie" na serwer kopii. Podstawową wadą takiego rozwiązania jest to że faktycznie rolę serwera spełnia w danym momencie tylko jeden komputer, natomiast pozostałe (kopie) do momentu awarii nie pełnią żadnej roli.

Problem niezawodności świadczonych usług można również rozwiązywać stosując moduł wspierający rozproszone przetwarzanie danych (klaster). Klaster to zbiór niezależnych komputerów (węzłów), połączonych siecią komunikacyjną, które z punktu widzenia użytkownika są postrzegane jako pojedynczy system (komputer) . Klaster oprócz zwiększenia niezawodności zapewnia również zwiększenie szybkości i dostępności usług, choć te zagadnienia nie będą rozwijane w dalszej części pracy.

2 Realizacja niezawodności oparta na systemach SSI

Wykorzystanie systemów rozproszonych (SSI) do realizacji niezawodności usług opiera się na odporności tych systemów na awarię pojedynczej maszyny (węzła). Jeżeli klaster zapewnia mechanizmy niewidocznej dla użytkownika obsługi awarii węzła, to zastosowanie rozproszonej architektury systemu operacyjnego korzystającej z wielu komputerów może być bardzo udanym sposobem uzyskania niezawodności systemu informatycznego.

W systemach rozproszonych (SSI) najczęściej jeden węzeł pełni rolę zarządcy (CLMS Master). Jest on uruchamiany jako pierwszy i do niego „przyłączają" się pozostałe węzły. Nadzoruje on pracę pozostałych węzłów (przykładowo może nakazać odłączenie węzła). Nie musi być on jednak przypisany do jednego konkretnego węzła – w przypadku awarii możliwe jest przypisanie jego roli innemu węzłowi (wybranemu np. algorytmem elekcji).

2.1 Zagadnienia związane z zapewnieniem niezawodności usług

2.1.1 Restart usługi na innym węźle (komputerze) w klastrze

W przypadku awarii jednego węzła usługa powinna zostać jak najszybciej wznowiona na innym węźle,

2.1.2 Replikacja danych procesu usługi pomiędzy węzłami

Zwiększa odporność klastra na awarie (aż do pełnej replikacji w której każdy węzeł replikuje swoje sesje do wszystkich pozostałych węzłów, przy takim rozwiązaniu wystarczy, że chociaż jeden z węzłów pozostanie sprawny, aby nie utracić informacji).

2.1.3 Możliwość rekonfiguracji usługi bez przerywania pracy

2.2 Wsparcie realizacji niezawodności usług w systemie OpenSSI

W systemie OpenSSI możemy określić pewne usługi które chcemy, aby były cały czas uruchomione w ramach klastra, ale tylko na jednym węźle jednocześnie (każda usługa tylko raz uruchomiona w ramach klastra). Nazywane są one kluczowymi usługami (CLMS Key Services). Podczas bootowania każdego węzeł zostaje określone, czy może być na nim wykonywana konkretna kluczowa usługa.

Każda kluczowa usługa określa sposób postępowania podczas awarii węzła wywołując funkcję clms_register_key_service. W momencie wystąpienia awarii wśród węzłów klastra zostaje wybrany kolejny węzeł mający przejąć usługę. Zostają wywołane funkcje określone przez argumenty clms_register_key_service (failover function, pull data function, failover data function, server ready function), oraz ostaje zarejestrowana nowa procedura awaryjna (funkcja clms_register_key_service) dla nowego węzła.

Można również określić procedurę odzyskiwania danych (pull_data routine). W momencie utraty węzła zostaje rozesłany do pozostałych węzłów sygnał o gromadzeniu danych kluczowej usługi która wykonywana była na danym utraconym węźle. Po zebraniu danych wyznaczana jest najbardziej aktualna wersja. W końcu kluczowa usługa jest zostaje uruchomiona na innym węźle.

W systemie OpenSSI istnieje usługa Kernel Data Replication Service, mająca za zadanie replikację danych nie jest ona jednakże wykorzystywana przez żaden podsystem.

W celu zapewnienia replikacji danych można stosować również specjalnie do tego stworzone systemy plików (np. DRBD).

W kwestii autorestartu usługi w systemie OpenSSI został zaimplementowany demon mające za zadanie kontrolę wykonywania procesu (usługi) i w przypadku przerwania (np. odłączeniu węzła na którym była uruchomiona usługa) uruchomienie jej na innym węźle:

- **keepalive** – demon odpowiedzialny za kontrolowanie (monitorowanie) i ewentualne restartowanie procesów i demonów zarejestrowanych przy użyciu *spawndeamon*. W momencie gdy praca danego procesu bądź demona zostanie przerwana *keepalive* zapisuje sytuację w logu i rozpoczyna wykonywanie skryptu restartującego proces/demon. *Keepalive* jest uruchamiany przez *init* na początku pracy systemu na jednym węźle. Można określić drugi węzeł na który „migruje” *keepalive* w przypadku awarii pierwszego (wywołanie *keepalive* –P

(pierwszy) –S (drugi), przy braku określenia tych opcji wybierany jest kolejny aktywny węzeł), oraz czas po którym *keepalive* kontroluje procesy (opcja –t).

Keepalive jest bardzo ważnym narzędziem w realizacji niezawodności gdyż zapewnia ponowne uruchomienie usługi na innym węźle w przypadku utraty węzła na którym pierwotnie była ona uruchomiona.

- **spawndeamon** – komenda będąca interfejsem demona *keepalive*. Posiada szereg opcji, wśród nich rejestrację nowego procesu/demona do puli procesów kontrolowanych przez *keepalive,* ustalenie na którym węźle w kolejności będzie uruchamiany proces/demon przy restarcie, wyłączenie demona *keepalive.* Dokładna specyfikacja: http://ci-linux.sourceforge.net/spawndaemon.shtml

3 Projekt wstępny implementacji

3.1 Analiza zadania

Mamy pewną ważną usługę (np. serwer HTTP) i chcemy aby była ona możliwie cały czas była dostępna. W przypadku awarii jednego z komputerów (zawieszenia usługi) chcemy możliwie „bezboleśnie" i szybko przeprowadzić ponowne uruchomienie usługi na innej maszynie.

Jednym ze sposobów jest użycie procesu kontrolującego wykonywanie usługi. Wybieramy jeden węzeł i uruchamiamy na nim procesu kontrolujący (PK). PK wybiera dowolny inny węzeł i uruchamia na nim usługę. W momencie gdy zostanie utracona komunikacja (1) z węzłem PK uruchamia usługę na innym węźle.

Problem pojawia się gdy węzeł na którym jest uruchomiany PK ulegnie awarii. Wtedy zatrzymanie usługi nie spowoduje żadnego efektu. Dlatego też trzeba poinformować pozostałe węzły (2) o utracie procesu kontrolnego. Wśród nich powinien zostać wybrany kolejny węzeł (3) pełniący rolę nowego PK. W tym momencie trzeba też sprawdzić czy usługa jest aktywna i poinformować PK na którym węźle, lub jeśli nie jest aktywna uruchomić na wybranym węźle.

Potencjalne rozwiązania zagadnień:

Ad.1) Proces kontrolny może wywołać proces (usługę) na innym węźle i czekać na sygnał zakończenia od niego (funkcje rforkm, waitforpid). Sygnał taki zostanie wysłany zarówno w przypadku „zabicia" procesu potomnego, jak i w przypadku utraty połączenia z węzłem na którym był wykonywany.
Inną metodą jest sprawdzanie co pewien czas czy proces jest uruchomiony w systemie.

Ad.2) Węzły muszą dowiedzieć się o zakończeniu działania procesu kontrolnego. Aby to zrealizować można uruchomić na każdym węźle proces czekający na zakończenie procesu „ojca".

Ad.3) Aby nastąpił wybór nowego PK musi być zapewniona komunikacja pomiędzy procesami uruchomionymi na wszystkich węzłach. Następnie procesy muszą ustalić na którym węźle będzie od nowa uruchomiony PK.

3.2 Dodatkowe funkcje dostępne w systemie OpenSSI

- int ***rexecv(****const char *file, char const *argv, clusternode_t node)* – uruchamia program na wybranym węźle,
- *int* ***rfork(***clusternode_t *node)* **–** tworzy proces potomny na wybranym węźle,

- *int* ***migrate(****clusternode_t node)* – przenosi wykonywany proces na inny węzeł (zwraca 0 jeśli proces został przeniesiony na inny węzeł)
- *int* ***cluster_maxnodes()*** – zwraca maksymalną ilość węzłów przewidzianą w klastrze
- ***cluster_events_register_signal*** – umożliwia obsługa sygnałów przychodzących do węzła (tj. node up, node down)
- ***cluster_getnodebyname*** – zwraca numer węzła na podstawie jego nazwy
- ***cluster_membership*** – zwraca listę aktualnie aktywnych węzłów (ze stanem ustawionym na CLUSTERNODE_UP)
- *char ** ***cluster_name()*** – zwraca nazwę węzła na którym uruchomiony jest proces
- int **cluster_ssiconfig()** – zwraca czy uruchomiony jest system OpenSSI
- int **clusternode_avail(**clusternode_t *nodenum***) –** zwraca czy dany węzeł klastra jest aktywny
- int **clusternode_info**(clusternode_t *nodenum*, int *sizeof (clusternode_info_t)***,** clusternode_info_t **nodeinfo***)* – umożliwia uzyskanie dokładnych informacji odnośnie węzła,
- clusternode_t **clusternode_num**(*void*) **–** zwraca węzeł na którym jest wykonywany proces,
- int **clusternode_setinfo(**clusternode_t *nodenum*, int *action***,** int *sizeof (clusternode_info_t)***,** clusternode_info_t **nodeinfo**) **–** umożliwia zmianę stanu węzła (wymagany jest odpowiedni poziom uprzywilejowania procesu, zmiana może dotyczyć m.in. odłączenia węzła).

4 Implementacja

4.1 Wstęp

Bardzo dużą zaletą systemu OpenSSI jest fakt, że procesy uruchamiane na innych węzłach dalej widziane są w wspólnej tabeli procesów i jest z nimi możliwa komunikacja taka sama jak z procesami uruchomionymi na tym samym (fizycznie) komputerze. Komunikacja po lokalnej sieci pomiędzy komputerami staje się niewidoczna (przeźroczysta) dla procesów. Bardzo ułatwia to pisanie programów w architekturze rozproszonej.

4.2 Zmiana założeń

W punkcie 3 (*„Projekt wstępny implementacji”*) przedstawiony został ogólny model zapewniający niezawodność usług przy wykorzystaniu algorytmu elekcji i koordynatora. Jednak system rozproszony OpenSSI zapewniając, o czym była mowa w punkcie 4.1, przezroczystość lokalizacji wykonywania procesu umożliwia znaczne uproszczenie implementacji. Można zmienić podejście do problemu zapewnienia niezawodności usług – nie musimy myśleć w kategorii wielu komputerów i chęci aby

usługa była na przynajmniej jednym z nich uruchomiona, można traktować go jako problem stałego uruchomienia procesu na jednej maszynie.

4.3 Komunikacja między procesami

Komunikację między procesami można zapewnić na parę sposobów. W implementacji rozważałem:

- kolejki komunikatów,
- pamięć dzieloną,
- łącza nienazwane,
- łącza nazwane,

Kolejki komunikatów – mechanizm IPC (Interprocess Communication) wprowadzony w systemie UNIX w wersji V. Umożliwia on stworzenie kolejki komunikatów które mogą być odbierane przez inne procesy. Pozwala również na filtrowanie odbieranych komunikatów (określenie typów odbieranych wiadomości – pole *msgtype* w funkcji *msgrcv*). Umożliwia to adresowanie wiadomości do konkretnego odbiorcy, czyli problem zapewnienia komunikacji między procesami zostaje rozwiązany. Niestety ta metoda nie umożliwia łatwego sprawdzenia czy proces, do którego wysyłamy, istnieje (wysyłamy tylko określony typ wiadomości). Kłopot może powstać zwłaszcza podczas algorytmu elekcji gdy, czekając aż wszystkie procesy „wypowiedzą" się, nie mamy możliwości sprawdzenia czy są one nadal aktywne.
(funkcje w Linuxie: msgget, msgsnd, msgrcv, msgctl).

Pamięć dzielona – podobnie jak *Kolejka komunikatów* mechanizm wprowadzony w UNIX System V, pozwalający na określenie pewnego obszaru pamięci dostępnego dla wszystkich procesów. Pierwszym kłopotem pojawiającym się natychmiast, jest konieczność zapewnienia synchronizacji zapisów i odczytów. Ten problem można rozwiązać stosując semafory i przydział konkretnej komórki pamięci dla każdego procesu. Dalej jednak pozostaje nierozwiązany problem sprawdzania czy wszystkie procesy są aktywne.
(funkcje w Linuxie: shmget, shmat, shmdt, shmctl).

Łącza nienazwane – umożliwiają jednokierunkową komunikację pomiędzy procesami (w ramach jednego programu). Jeden z procesów wysyła dane do łącza a inny odczytuje w takiej samej kolejności w jakiej zostały wysłane. Łącze posiada więc organizację kolejki FIFO (First In First Out).
Sama realizacje systemowa opiera się na tworzeniu tymczasowych obiektów w pamięci jądra i udostępnienie ich poprzez interfejs systemu plików. Przy tworzeniu nowego łącza system otwiera je od razu do czytania i pisania.
Ponieważ łącza nienazwane są jednokierunkowe (jeden proces zamyka odczyt, drugi zapis) aby zapewnić komunikację dwustronną trzeba utworzyć dwa kanały dla każdego procesu.
Łącza nienazwane spełniają wymaganie dotyczące komunikacji między procesami, a dodatkowo umożliwiają sprawdzenie czy proces do którego wysyłamy, czy odbieramy, wiadomość jest aktywny (w przypadku zakończenia procesu zamykane są jego łącza i próba odczytania z niego wiadomości np. funkcją read kończy się zwrotem

błędu).
Podsumowując łącza nienazwane zapewniają:

- komunikację pomiędzy procesami uruchomionymi na różnych węzłach,
- informację o zakończeniu usługi na którymś węźle

(funkcje w Linuxie: pipe, read, write, close).

Łącza nazwane – są pewnym rozszerzeniem *łączy nienazwanych* pozwalającym na komunikację między dowolnymi procesami (nie tylko w ramach jednego programu). System realizuje to poprzez tworzenie tymczasowych plików (o określonej nazwie) typu FIFO.
Niestety znowu pojawiają się problemy ze sprawdzeniem istnienia innego procesu w systemie.
(funkcje w Linuxie: mknod, mkfifo, open, unlink).

4.4 Opis działania programu

Na początku działania programu sprawdzane są parametry klastra tj. ilość węzłów, numery aktywnych węzłów. Dalej tworzone są, dla każdego węzła, dwa kanały komunikacyjne. Tablicę kanałów do wszystkich węzłów posiada każdy proces. Następnie funkcją rfork tworzone są procesy potomne na wszystkich węzłach. Każdy proces potomny ma określony priorytet. Najwyższy (0) ma proces uruchomiony na węźle określonym przy inicjalizacji jako docelowy dla usługi.

Proces o najwyższym priorytecie rozpoczyna świadczenie usługi. Wszystkie inne przechodzą w stan „uśpienia" oczekując na otrzymanie informacji (funkcja read) od procesu który w danej chwili jest odpowiedzialny za usługę. Jeśli węzeł ten zostanie uszkodzony wtedy przerwane zostanie również oczekiwanie na odebranie informacji. Za pomocą tego mechanizmu zostaje również przesłany sygnał zakończenia działania usługi (określony w programie jako TERMINATE).

Każdy proces który nie odpowiada w danej chwili za usługę czeka na odebranie informacji po kolei od wszystkich procesów o wyższym priorytecie. Czyli można by powiedzieć czeka na swoją kolejkę. Gdy zostaną zerwane wszystkie łącza komunikacyjne do procesów o wyższym priorytecie uznaje on że powinien rozpocząć świadczenie usługi.
Warto również wspomnieć o konieczności przechwytywania sygnału SIGHUP, który pojawia się przy zamknięciu terminalu przydzielonego do procesu.
Zaimplementowane funkcje:

- **int spawn_init (cluster_node usluga, int rosnaco, int pisz)** – funkcja inicjalizująca proces zapewniający niezawodność.
  Parametry wejściowe:
  usluga = węzeł na którym ma być uruchomiona usługa,
  rosnaco = wybór kolejnych węzłów na których będzie uruchomiona usługa (1 – rosnąco, 0 malejąco),
  pisz = 1 – wypisuj komunikaty, 0 – brak komunikatów
  Wyjście:
  Funkcja zwraca 0 jeśli inicjalizacja przebiegła pomyślnie, -1 w przeciwnym

przypadku.

- **int spawn_start ()** – funkcja uruchamia proces mający zapewnić niezawodność usługi.
  Parametry wejściowe:
  brak
  Wyjście:
  brak

- **int spawn_terminate ()** – funkcja kończy proces mający zapewnić niezawodność usługi.
  Parametry wejściowe:
  brak
  Wyjście:
  brak

5 Możliwe rozszerzenia

Znacznym zwiększeniem funkcjonalności programu zapewnienia niezawodności usług będzie implementacja dwóch dodatkowych mechanizmów:

- obsługi zakończenia działania procesu na innym węźle w przypadku gdy ten węzeł jest dalej aktywny w klastrze (komputer jest włączony i pracuje w klastrze). Celem niniejszego projektu było zapewnienie kontynuacji świadczenia usługi w przypadku poważnej awarii (wyłączenie komputera, utrata połączenia) w której nie ma możliwości dalszego świadczenia usługi na danym komputerze w klastrze. Jednak dodanie mechanizmu ponownego uruchamiania usługi na tym samym komputerze zapewniło by zabezpieczenie przed przypadkowym wyłączeniem usługi (np. przez użytkownika) czy chwilowym błędem systemu operacyjnego. Rozszerzenie to można wprowadzić do programu zaraz po stwierdzeniu utraty połączenia (za funkcją read) sprawdzając czy węzeł na którym była uruchomiona usługa jest aktywny (funkcja clusternode_avil).

- uruchamiania procesów kontrolujących działanie usługi na węzłach nowo pojawiających się w klastrze (nieaktywnych w momencie inicjalizacji programu). Chociaż przy pojawianiu się nowych klientów procesy kontrolujące wykonywanie usługi tworzone są na nowo na wszystkich węzłach, ale można sobie wyobrazić przypadek w którym przy jednej długiej sesji następuje wyłączenie prawie wszystkich węzłów w kalstrze (prócz jednego) a następnie bez przerywania sesji ponowne uruchomienie wszystkich. Tego momentu program już nie będzie w stanie obsłużyć i w momencie wyłączenia jednego (tego który był cały czas aktywny) nastąpi przerwanie świadczenia usługi. Rozwiązanie problemu ma dwa aspekty: pierwszy z niech to rozpoznanie pojawia się nowych węzłów – można to uzyskać sprawdzając co pewien czas stan klastra, albo przechwytując sygnał node_up (funkcja *cluster_events_register_signal*). Drugi problem to dodanie węzła do struktury działającego programu. Można to rozwiązać na dwa sposoby: albo dokonać ponownej inicjalizacji programu (ponowne odczytanie parametrów klastra i ponowne stworzenie procesów kontrolnych na wszystkich

węzłach), albo „doczepiając” nowy proces kontrolny z najniższym priorytetem (będzie czekał na przerwanie świadczenia usług na wszystkich innych węzłach – czyli inne procesy, wcześniej stworzone, nie będą miały potrzeby komunikacji z nim).

6 Zakończenie

Niniejsza praca i projekt były próba stworzenia programu umożliwiającego świadczenie niezawodnych usług w systemach rozproszonych.

W systemie operacyjnym w którym wykonywany był projekt (Debian z klastrem OpenSSI) nie została do tej pory zaimplementowana podobna usługa. Istnieje wprawdzie, o czym była mowa w punkcie 2.2, demon zapewniający ponowne uruchomianie usługi na wybranych węzłach, lecz z pewną istotną różnicą – przy utracie połączenia z klientem. Dlatego też cały projekt jest w pewien sposób pionierski i stanowi raczej próbę zaproponowania pewnego (działającego) rozwiązania.