Rozproszone zarządzanie blokadami

Projekt z przedmiotu RSO

Marek Majchrowski
M.Majchrowski@elka.pw.edu.pl

16 czerwca 2005

Spis treści

1 Rozproszone zarządzanie blokadami **2**

2 Podejścia stosowane w algorytmach wzajemnego wykluczania w systemach rozproszonych 2
2.1 Algorytmy scentralizowane 2
2.2 Algorytmy beztokenowe (ang. *non-token based*) 3
2.2.1 Algorytm Lamporta 3
2.3 Algorytmy tokenowe (ang. *token based*) 4
2.3.1 Algorytmy oparte na rozgłaszaniu 4
2.3.2 Algorytmy oparte na logicznej strukturze węzłów 5

3 Istniejące rozwiązania **5**
3.1 OpenDLM – Open Distributed Lock Manager 6
3.2 GULM – Grand Unified Lock Manager 6
3.3 SYNClib – Scalable Distributed Synchronization Library 6
3.4 JINI-DLM – JINI Distributed Lock Manager 6
3.5 IDML for OPS – Integrated Distributed Lock Manager for Oracle Parallel Server 6
3.6 Ensemble – The Ensemble Distributed Communication System . . . 7

4 Instalacja OpenDLM 0.9.3 **7**
4.1 Komponenty *OpenDLM* 8
4.2 Instalacja 8
4.2.1 Wstępne wymagania 8
4.2.2 Kompilacja 9
4.2.3 Instalacja 9
4.2.4 Konfiguracja 9
4.2.5 Uruchamianie 10
4.2.6 Zatrzymywanie 10
4.2.7 Programy użytkowe 11

5 Próba instalacji *GULM* **11**

6 Łaty **12**
6.1 patch-linux.2.6.11.12-exit_mm.diff 12
6.2 patch-opendlm.0.9.3-exports.diff 12

1 Rozproszone zarządzanie blokadami

W momencie gdy proces musi odczytać, bądź zmienić pewne zasoby współdzielone (ang. *shared resource*) (jak struktury danych, pliki, itd.), musi najpierw dostać się do sekcji krytycznej (ang. *critical section*) w celu zapewnienia wzajemnego wykluczenia (ang. *mutual exclusion*). Oznacza to, że żeden inny proces nie może mieć dostępu do tych zasobów w tym czasie. Jednak w wyniku rozprzestrzeniania się zasobów, braku globalnej informacji oraz opóźnień spowodowanych komunikacją, metody synchronizacji oraz algorytmy zapewniające wzajemne wykluczenia stosowane w systemach jedno-procesorowych nie mogą zostać zastosowane w systemach rozproszonych. W efekcie rozwinęły się prace nad algorytmami wzajemnego wykluczania w systemach rozproszonych.

Jak sie okazuje podłoże teoretyczne dla tych rozwiązań zostało szeroko omówione w literaturze [Sin93], [Din89], [Era95], [Joh94], [MBB92], [CR83].

W dalszej części sprawozdania przedstawię najważniejsze techniki oraz algorytmy wzajemnego wykluczania w systemach rozproszonych, jak również klasyfikację opartą na stosowanych podejściach.

2 Podejścia stosowane w algorytmach wzajemnego wykluczania w systemach rozproszonych

Poniższa sekcja podsumowuje niektóre klasyczne algorytmy rozproszonej synchronizacji. Większość algorytmów tu przedstawionych zakłada, że istnieje jeden globalny zasób, do którego dostęp musi zostać uszeregowany (ang. *serialized*).

2.1 Algorytmy scentralizowane

Najbardziej bezpośrednią drogą, aby zapewnić wzajemne wykluczanie w rozproszonym środowisku jest zasymulowanie takiej sytuacji w systemie jedno-procesorowym. Możemy to zrobić w następujący sposób:

- jeden wyróżniony proces zostaje wybrany jako koordynator (ang. *coordinator*) dostępu do sekcji krytycznej innych procesów,
- każdy proces, który chce się dostać do sekcji krytycznej, wysyła wiadomość żądania (ang. *REQUEST message*) do koordynatora,
- kiedy proces odbierze wiadomość odpowiedzi (ang. *REPLY message*) od koordynatora, może on kontynuować swoje wykonanie w sekcji krytycznej,
- po opuszczeniu sekcji krytycznej proces informuje o tym koordynatora wysyłając wiadomość o jej zwolnieniu (ang. *RELEASE message*).

Proces koordynatora możemy sobie wyobrazić jako nieskończoną pętlę obsługująca przychodzące żądania oraz kolejkującą odwołania do sekcji krytycznych procesów.

Jest oczywiste, że algorytm ten gwarantuje wzajemne wykluczanie oraz nie dopuszcza do zagłodzenia procesów (zakładając sprawiedliwą strategię szeregowania odwołań - np. FIFO). Jednakże takie scentralizowane podejście ma także mankamenty oraz wady:

- Po pierwsze taki algorytm nie jest zdecentralizowany, co czyni go mało skalowalnym.
- Po drugie nie jest on odpowiedni dla dużej liczby procesów. Koordynator jest jego słabym punktem (ang. *single point of failure*) - jeśli mu się coś stanie, cały system przestanie działać.

Algorytm ten wymaga 3 wiadomości na każde wejście do sekcji krytycznej [Gos91].

2.2 Algorytmy beztokenowe (ang. *non-token based*)

Kluczowym pomysłem zwiększenia słabej skalowalności powyższego podejścia jest rozdzielenia funkcjonalności koordynatora na cały system. Jeśli wszystkie węzły będą wykonywały ten sam kod w serwerach synchronizacyjnych, ten algorytm można nazwać symetrycznym. Poniższy algorytm jest nazywany beztokenowym (ang. *non-token based*), ponieważ próbuje on dostać pozwolenie albo ustalić, który węzeł będzie miał prawo dostępu do sekcji krytycznej jako następny [RS95].

2.2.1 Algorytm Lamporta

Pierwszym rozwiązaniem problemu rozproszonej synchronizacji jest klasyczny algorytm Lamporta [Lam78]. Zakłada on rozproszony system, w którym każdy proces utrzymuje swoje własne kolejki żądań, które są lokalne dla każdego procesu. Każdy wpis w takiej kolejce zawiera $(T_i, P_i, AKCJA)$, gdzie:

- T_i jest znacznikiem czasu wiadomości,
- P_i jest identyfikatorem procesu powiązanego z daną wiadomością,
- $AKCJA$ definiuje akcję na zasobie, zazwyczaj ŻĄDANIE (ang. *REQUEST*), albo ZWOLNIENIE (ang. *RELEASE*).

Początkowo zakłada się, że wszystkie kolejki żądań zawierają pojedynczą wiadomość $(T_0, P_0, REQUEST)$, gdzie:

- T_0 jest znacznikiem czasu, który początkowo ma wartość mniejszą, niż którykolwiek zegar,
- P_0 jest identyfikatorem procesu, który początkowo ma prawo korzystać z zasobu.

Algorytm Lamporta gwarantuje wzajemne wykluczanie w następujący sposób:

- Jeśli P_i żąda zasobu, wysyła wiadomość zawierającą $(T_m, P_i, REQUEST)$ do pozostałych procesów i umieszcza tą wiadomość w swojej kolejce żądań.
- Jeśli P_j odbierze wiadomość taką jak powyższa, umieszcza ją w swojej kolejce żądań a następnie wysyła potwierdzenie wraz ze stemplem czasowym do żądającego procesu.
- Jeśli poniższe warunki zostały spełnione przez P_i lokalnie, proces P_i jest uprawniony do korzystania z zasobu:
  - Pierwszą wiadomością w swojej kolejce żądań jest wiadomość $(T_m, P_i, REQUEST)$.

  - Proces P_i odebrał wiadomość od pozostałych procesów później niż T_m.

- Jeśli P_i zwalnia zasób, usuwa wszystkie wiadomości typu $(T_m, P_i, REQUEST)$ ze swojej kolejki żądań i wysyła wiadomość $(T_n, P_i, RELEASE)$ do pozostałych procesów.
- Jeśli P_j odbierze wiadomość taką jak powyższa, usuwa wszystkie wiadomości typu $(T_m, P_i, REQUEST)$ ze swojej kolejki żądań.

Algorytm ten wymaga $3(N-1)$ wiadomości na każde wejście do sekcji krytycznej, gdzie N jest liczbą procesów [Gos91]. Ulepszoną wersję algorytmu Lamporta, która wymaga jedynie $2(N-1)$ wiadomości na każde wejście do sekcji krytycznej, zaproponowali Ricart i Agrawala [RA81]. W 1985 roku jeszcze wydajniejszą wersję zaprezentowali Suzuki i Kasami [SK85]. Wymagała ona jedynie N wiadomości na każde wejście do sekcji krytycznej, przy założeniu, że numery sekwencyjne zawarte w nagłówkach wiadomości pozostaną nieograniczone. Maekawa [Mae85] zaprezentował algorytm ze złożonością $O(sqrt(N))$. Obecnie najwydajniejszym algorytmem beztokenowym (biorąc pod uwagę czas odpowiedzi) jest algorytm Ramachandrana i Singhala opisany w [RS95]. Jednakże nawet ten algorytm wymaga przesłania średnio $2(N-1)$ wiadomości na każde wejście do sekcji krytycznej.

2.3 Algorytmy tokenowe (ang. *token based*)

W algorytmach tokenowych unikalny token jest współdzielony przez procesy. Wzajemne wykluczanie jest zagwarantowane w sposób oczywisty, ponieważ proces może jedynie wejść do swojej sekcji krytycznej w przypadku posiadania tokena i przy założeniu, że w systemie istnieje tylko jeden token. Ta zasada może zostać zaimplementowana zarówno poprzez rozgłaszanie do innych procesów chęci posiadania tokena (można tak zrobić zarówno w statycznie jak i w dynamicznie wybranym zbiorze węzłów) jak i przez przekazywanie tokena w logicznej strukturze węzłów, która może być zarówno statyczna jak i dynamiczna.

2.3.1 Algorytmy oparte na rozgłaszaniu

Algorytmy tego typu nie narzucają struktury komunikacji na procesy i dlatego muszą wysyłać wiadomości żądania do innych miejsc równolegle. Jednakże algorytmy te mogą być statyczne bądź dynamiczne.

Statyczne algorytmy są bezstanowe – w sensie tego, że nie pamiętają historii dostępu do sekcji krytycznej i w związku z tym muszą wysyłać wiadomość żądania do pozostałych procesów. Przykłady takiego podejścia zostały zaprezentowane w [SK85], [CR83], [NLM89].

Z drugiej jednak strony dynamiczne algorytmy (jak [CSL91]) trzymają ślad ostatnich lokalizacji tokena i dlatego wiadomość żądania może być wysłana tylko to wybranej części miejsc, które są podejrzane o posiadanie tokena (tj. proces, który aktualnie posiada token oraz procesy, które najprawdopodobniej będą się ubiegały o jego posiadanie). Należy zauważyć, że możliwe jest także przekazywanie wiadomości żądań przez procesy, które nie posiadają tokena, ale mogą wiedzieć kto go posiada.

Algorytmy statyczne wymagają wysłania średnio N wiadomości na każde wejście do sekcji krytycznej. Natomiast algorytmy dynamiczne wymagają wysłania średnio od $\frac{N}{2}$ wiadomości na każde wejście do sekcji krytycznej w mało obciążonych systemach, do N wiadomości na każde wejście do sekcji krytycznej w wysoko obciążonych systemach [Sin93].

2.3.2 Algorytmy oparte na logicznej strukturze węzłów

W przeciwieństwie do poprzedniego podejścia, algorytmy oparte na logicznej strukturze węzłów narzucają pewną wirtualną topologiczną komunikację między procesami. Podejście to unika rozgłaszania przez co redukuje obciążenie sieci. Jednakże algorytmy te wykazują większe opóźnienia i czasy odpowiedzi wynikające z przekazywania wiadomości.

Algorytmy oparte na logicznej strukturze możemy podzielić na:

1. statyczne,
2. dynamiczne.

W pierwszym przypadku logiczna struktura węzłów utworzona przez algorytm pozostanie niezmienna w trakcie pracy. Możliwe topologiczne struktury to:

- pierścienie [Mar85].
  Token cyrkuluje w pierścieniu szeregowo od procesu do procesu. Wymaga to wysłania N wiadomości na każde wejście do sekcji krytycznej.
- drzewa skierowane [Ray89]:
  Wiadomość żądania jest przekazywana w górę drzewa, od procesu żądającego aż do korzenia, skąd następnie token jest wysyłany drogą powrotną poprzez zmianę skierowania krawędzi. Raymond [Ray89] wykazał, że liczba wiadomości wymienianych w systemach o małym obciążeniu jest rzędu $O(log(N))$, natomiast w systemach o wysokim obciążeniu wystarczą jedynie 4 wiadomości na każde wejście do sekcji krytycznej.
- grafy skierowane [NM90]:
  Grafy skierowane są uogólnioną wersją struktury drzewiastej, w której nadmiarowe krawędzie powodują, że struktura jest odporna na awarie (ang. *fault-tolerance*) kosztem odrobinę zwiększonego obciążenia łączy [CSL90a].

W drugim przypadku logiczna struktura narzucona przez algorytm może dynamicznie ulegać zmianie w trakcie pracy (nie tylko pod względem skierowania krawędzi, ale także pod względem topologicznego kształtu). Przykłady takich technik zostały zaprezentowane przez Naimia i Trehela [NT87] oraz przez Bernabeu'a i Ahamada [BA89]. Obecnie najwydajniejszym algorytmem sprawdzającym się w systemach o różnym obciążeniu jest podejście zaproponowane przez Changa, Singhala i Liu [CSL90b].

3 Istniejące rozwiązania

Jak sie okazuje, w rzeczywistości nie istnieje zbyt wiele implementacji rozproszonego zarządzania blokadami (ang. *distributed lock manager – DLM*). Znalezione przeze mnie implementacje:

1. OpenDLM
2. GULM
3. SYNClib
4. JINI-DLM

Oprócz powyższych projektów, znalazłem wiele projektów komercyjnych (i nie tylko), których DLMy są częścią:

1. IDML for OPS
2. CEnsamble

3.1 OpenDLM – Open Distributed Lock Manager

OpenDLM [Ope] jest w pełni rozproszonym zarządcą blokad. Wiedza o każdej blokadzie jest rozprzestrzeniana pośród "zainteresowanych" węzłów. Każdy węzeł w takiej sieci posiada i wykorzystuje pełną logikę w celu obsługi żądań oraz zezwalania na wejście do sekcji krytycznych.

OpenDLM nie opiera się na strukturze scentralizowanej, nie ma w niej żadnego centralnego serwera zawierającego informacje o wszystkich blokadach, przez co jest odporny na awarie poszczególnych węzłów.

Posiada interfejsy do użycia zarówno w przestrzenie użytkownika jak i jądra.

3.2 GULM – Grand Unified Lock Manager

GULM [GUL] jest opartym na serwerach zarządcą blokad dedykowanym projektom: *GFS* [GFS], *GNBD* oraz *CLVM*. Może zastąpić *DLM* w projekcie *GFS*. Pojedynczy serwer *GULM* może zostać uruchomiony w wolno stojącym trybie. Jest wtedy newralgicznym punktem całego systemu – decyduje o jego niezawodności. Natomiast trzy lub pięć serwerów uruchomionych równolegle jest odpornych na awarię części z nich. *GULM* składa sie z biblioteki oraz programów napisanych w przestrzeni użytkownika.

Podobnie jak *DLM*, *GULM* jest częścią projektu *Cluster Project* [Pro].

3.3 SYNClib – Scalable Distributed Synchronization Library

Głównym celem projektu *SYNClib* było dostarczenie wydajnej oraz skalowalnej usługi synchronizacyjnej dla systemów rozproszonych, wykorzystującej jakąś bibliotekę wymiany wiadomości (jak np. *MPI* [MPI95]).

Biblioteka ta, wykorzystuje pewną modyfikację podejścia zaproponowanego przez Changa, Singhala i Liu [CSL90b]. Jest to system w pełni rozproszony, odporny na błędy (poprzez wykorzystanie *MPI*). Udostępnia programiście API, jest zorientowana obiektowo.

3.4 JINI-DLM – JINI Distributed Lock Manager

JINI-DLM [DML] jest odpornym na awarie, rozproszonym zarządcą blokad napisanym w Javie dla projektu *Blitz JINI* [JIN] w technologii *JGroups*.

3.5 IDML for OPS – Integrated Distributed Lock Manager for Oracle Parallel Server

IDML for OPS jest projektem komercyjnym używanym w rozwiązaniach firmy *Oracle*.

3.6 Ensemble – The Ensemble Distributed Communication System

Ensemble [Ens] jest pakietem narzędziowym służącym do budowy niezawodnych aplikacji. Udostępnia on biblioteki protokołów, które mogą zostać użyte w celu szybkiego zbudowania w pełni rozproszonej aplikacji. Wspiera języki ML, C, C++, Java na wielu systemach operacyjnych: Linux, Solaris, NT.

CEnsemble jest pakietem *Ensemble* przepisanym w języku C (*Ensemble* jest napisany w ML). Wymaga on wsparcia *multicast*. API bibliotek *CEnsemble* jest nieudokumentowane, jednak jest bardzo podobne do tego znanego z pakietu *Ensemble*.

Wraz z *CEnsemble* jest dystrybuowany pełny rozproszony zarządca blokad (stworzony za pomocą tego pakietu narzędziowego). DLM może być użyty zarówno poprzez interfejs tekstowy (za pomocą potoków), albo może być skonsolidowany z aplikacją jako biblioteka języka C. Aby posługiwać się DLM nie jest wymagana znajomość pakietu narzędziowego *Ensemble*. DLM zapewnia: odporność na błędy, balansowanie obciążeniem oraz blokady odczytu/zapisu.

4 Instalacja OpenDLM 0.9.3

OpenDLM został stworzony w celu dostarczenia w pełni funkcjonalnego i odpornego na awarie rozproszonego systemu blokad. Dostarcza semantykę identyczną z tą, zdefiniowaną w dokumentacji *DEC VAX Cluster*. Z początku kod *OpenDLM* był częścią *HACMP – High Availability Cluster Multi-Processing* – komercyjnego rozwiązania firmy *IBM* dla zastosowań klastrowych. W chwili obecnej *OpenDLM* jest wspierany przez firmę *Oracle*, która wykorzystuje go w swoich serwerach *OPS – Oracle Parallel Server*. Jednakże, *OpenDLM* jest dobrym rozwiązaniem dla dowolnego rozproszonego systemu, wymagającego wiarygodnego systemu blokad.

Opis API dla wczesnej wersji *OpenDLM* dostępny jest na stronach *HACMP*: *http://www-1.ibm.com/servers/eserver/pseries/library/hacmp_docs.html*. Odpowiada on wersji *HACMP* $V4.4.1$. Podręcznik ten opisuje zagadnienie *HACMP* na systemach *AIX*, jednakże opisywana składnia oraz semantyka funkcji bibliotecznych jest taka sama jak w *OpenDLM* dla systemu *Linux*.

W oficjalnym pliku *README* z pakietu *OpenDLM* jest napisane, że na chwilę obecną odwoływanie się do *OpenDLM* jest możliwe jedynie z przestrzeni użytkownika poprzez biblioteki dzielone (*libdlm.so*), nie jest natomiast możliwe to z przestrzeni jądra.

Jednak w kodzie znalazłem moduł *libdlmk*, ktróry udostępnia API dla przestrzeni jądra prawie identyczne z tym dla przestrzeni użytkownika.

Działanie systemu *OpenDLM* przetestowałem za pomocą programów testowych dystrybuowanych wraz z całym pakietem. Znajdują się one w katalogach *opendlm/src/user/tests* oraz *opendlm/src/kernel/tests* odpowiednio dla przestrzeni użytkownika oraz jądra systemu operacyjnego.

Muszę jednak zwrócić uwagę, że nie obyło się to bez mojej ingerencji w kod *OpenDLM*. Testy sprawdzające działanie systemu z poziomu przestrzeni użytkownika działały od razu. Natomiast moduł testujący ten system z poziomu przestrzeni jądra nie ładował się, wypisując komunikat:

```
kclient2: Unknown symbol dlmlock
kclient2: Unknown symbol exit_mm
kclient2: Unknown symbol dlmunlock
```

W załączniku umieściłem łaty poprawiające powyższe błędy. Jak się okazało moduł udostępniający API dla przestrzeni jądra nie eksportował symboli *dlmlock* oraz *dlmunlock*. Drugi błąd został wyeliminowany także poprzez dopisanie do źródeł jądra *Linuksa* eksportowania odpowiedniego symbolu (żaden inny moduł nie używa tej funkcji).

Oto wynik działania *OpenDLM* w przestrzeni jądra na podstawie przykładowego modułu:

```
init_module Given options:
init_module    Mode[CR] Name[RES-A] Waittime[10].
kclient2_main Call dlmlock mode/name/astarg[RES-A/CR/0]!
[ast_func] status/astarg/id[dlm_errmsg not yet supported/0/1(1)]
kclient2_main Return DLM_NORMAL from dlmlock, lksb.status = 0!
kclient2_main Sleeping 10 secs:
kclient2_main AST block executed!
kclient2_main Call dlmunlock id/name/flag[1(1)/RES-A/808000]!
[ast_unlock_func] status/astarg/id[dlm_errmsg not yet supported/0/1(1)]
kclient2_main Return DLM_NORMAL from dlmunlock!
kclient2_main Sleeping 10 secs:
kclient2_main UNLOCK AST block executed!
kclient2_main TEST FINISHED SUCCESSFULLY
```

4.1 Komponenty *OpenDLM*

OpenDLM zawiera:

- bibliotekę, udostępniającą API przestrzeni użytkownika (*libdlm.so*),
- demona (*dlmdu*) odbierającego zdarzenia z klastra oraz ładującego moduły jądra,
- cztery moduły odpowiedzialne za:
  - dlmdk_base – udostępnia globalne struktury oraz dane,
  - cccp – odpowiedzialny na komunikację pomiędzy węzłami w klastrze,
  - dlmdk_core – rdzeń zarządcy blokad, kontroluje wszystkie żądania blokad,
  - libdlmk – udostępnia API przestrzeni jądra,

4.2 Instalacja

4.2.1 Wstępne wymagania

Ponieważ *OpenDLM* zapewnia jedynie własną wiarygodną komunikację między węzłami klastra (poprzez moduł *cccp*), nie implementuje natomiast własnej usługi *heartbeat*, wymaga on takiego zarządcy w obrębie danego klastra. W chwili obecnej wspiera następujące rozwiązania:

- heartbeat – http://www.linux-ha.org,
- ccm – http://www.linux-ha.org,
- iiquorumd – http://www.ca.com/opensource
- ci_linux – http://ci-linux.sf.net

4.2.2 Kompilacja

W celu kompilacji należy wykonać następujące polecenia:

```
$ ./bootstrap (w przypadku źródeł z CVSa)
$ ./configure --with-<cluster manager>
```

Nazwą zarządcy infrastruktury klastra (ang. *cluster manager*) jest jedna z tych, wymienionych w poprzednim podrozdziale.

4.2.3 Instalacja

Po udanej kompilacji jako superużytkownik należy wykonać polecenia:

```
$ make install
$ depmod -ae
```

4.2.4 Konfiguracja

Należy stworzyć plik konfiguracyjny (domyślnie */etc/haDLM.conf* chyba, że podano inaczej w czasie kompilacji).

W pliku konfiguracyjnym należy podać:

- liczbę węzłów w klastrze oraz ich numery *IP*,
- nazwę intefejsu infrastruktury klastra,
- (opcjonalnie) nazwę urządzenia w katalogu */dev* (domyślnie *haDLM*,
- (opcjonalnie) numer *major* urządzenia (domyśle 250),
- (opcjonalnie) numery *minor* dla wpisów *locks* oraz *admin* (domyślnie odpowiednio 1 oraz 0),

Oto przykładowy plik konfiguracyjny:

```
NODECOUNT 1
1 marek 10.1.100.130
DLMNAME haDLM
DLMMAJOR 250
DLMCMGR heartbeat
DLMADMIN admin 0
DLMLOCKS locks 1
```

Należy dodać do pliku (w zależności od wersji jądra) następującą linię:

- dla wersji $2.6.x$ – do pliku */etc/modprobe.d/aliases*:

  ```
  alias haDLM dlmdk_core
  ```

- dla wersji $2.4.x$ – do pliku */etc/modprobe.conf*:

  ```
  alias haDLM dlmdk.core
  ```

a następnie uruchomić ponownie:

```
$ depmod -ae
```

4.2.5 Uruchamianie

Na początku należy uruchomić zarządcę infrastruktury klastra. Następnie albo uruchamiamy system *OpenDLM* poprzez skrypt startowy:

```
$ /etc/init.d/haDLM start
$ modprobe libdlmk
```

albo odpalamy demona *dlmdu* ręcznie:

```
$ /path/to/dlmdu [-C FILE] [-d OPTIONS]
$ modprobe libdlmk
```

gdzie:

- *-C FILE* – określa ścieżkę do pliku konfiguracyjnego,
- *-d OPTIONS* – określa flagi dla odpluskwiania (ang. *debug flags*).

W przypadku poprawnego wykonania wszystkich dotychczasowych czynności powinniśmy zobaczyć komunikaty podobne do tych:

```
Starting haDLM DLMDU Major/Minor(0/4/0x4) Time(12:42:19) Date(Jun 16 2005)
Opening configuration file [/usr/local/openDLM/etc/haDLM.conf]
Opened configuration file.
Attempt to load DLM module, cmd [modprobe haDLM haDLM_major_number=250 haDLM_name=haDLM haDLM_admin_minor=0 haDLM_locks_
Error loading DLM module, rc [-1]
This is a known OS issue, re-verifying status..SUCCESS
Opened [/dev/haDLM/admin], filedes [0]
1st thread, pid [30553]
Wrote code/size[0/12], blocks so far [1]
Created thread, id [-1209812048]
2nd thread, pid/tid [30553/-1209812048]
Local node is [marek]
Node have [1] nodeq elements queued
Node have [2] nodeq elements queued
Node have [3] nodeq elements queued
compare [marek] and [marek]
found number/name[1/marek/10.1.100.130]
find_configured_node: Local is marek
main: sending message
Wrote code/size[2/12], blocks so far [2]
main: sent message
compare [marek] and [marek]
found number/name[1/marek/10.1.100.130]
find_configured_node: Local is marek
main: sending message
Wrote code/size[2/12], blocks so far [3]
main: sent message
main: sending message
Wrote code/size[2/12], blocks so far [4]
main: sent message
```

Oprócz tego w katalogu */proc* powinny pojawić się:

- plik *cccp*,
- katalog *haDLM*.

4.2.6 Zatrzymywanie

Aby zatrzymać system *OpenDLM* poprzez skrypt startowy należy wykonać:

```
$ modprobe -r libdlmk
$ /etc/init.d/haDLM stop
```

Aby wykonać to ręcznie należy uruchomić:

```
$ kill <dlmdu pid>
$ modprobe -r libdlmk
$ modprobe -r haDLM
```

4.2.7 Programy użytkowe

Programy wykorzystujące *OpenDLM* powinny włączać plik nagłówkowy *#include <opendlm/dlm.h>*, oraz powinny być konsolidowane z biblioteką *dlm* (*-ldlm*).

5 Próba instalacji *GULM*

Na początku chcę zaznaczyć, że nie udało mi się w pełni uruchomić systemu *GULM*.

Projekt *GULM* jest częścią projektu *Cluster Project*. Źródła projektu pobrałem z *CVSa*.

```
$ cvs -d :pserver:cvs@sources.redhat.com:/cvs/cluster login
$ cvs -d :pserver:cvs@sources.redhat.com:/cvs/cluster checkout cluster
```

Następnie skompilowałem cały projekt:

```
$ cd cluster
$ ./configure --kernel_src=/lib/modules/2.6.11.12/source
$ make
```

oraz go zainstalowałem:

```
$ make install
```

Według instrukcji do programu *GULM* wystarczy, aby załadować odpowiednie moduły (co też zrobiłem):

```
$ modprobe gfs
$ modprobe lock_gulm
```

a następnie odpalić programy:

```
$ ccsd
$ lock_gulmd --use_ccs
```

Po stworzeniu odpowiednich plików konfiguracyjnych (*/etc/cluster/cluster.conf*) oraz poprawnym odpaleniu programów próbowałem uruchomić przykładowe programy testowe dla serwera *GULM*.

Niestety w przypadku obu programów testowych ich uruchomienia zakończyło się niepowodzeniem:

```
$ ./basiclocktest
Starting TestBox For Lock
Failed to send login request to core. err -111
$ ./coretest
Starting TestBox For Core
Failed to send login request to core. err -111
```

Mój plik konfiguracyjny ma postać:

```
<?xml version="1.0"?>
<cluster name="alpha" config_version="1">
<gulm>
     <lockserver name="marek"/>
</gulm>
<clusternodes>
<clusternode name="marek">
     <fence>
          <method name="simple">
               <device name="apc1" port="3"/>
          </method>
     </fence>
</clusternode>
</clusternodes>

<fencedevices>
   <fencedevice name="apc1" agent="fence_apc" ipaddr="10.1.100.130" login="apc" passwd="apc"/>
</fencedevices>
</cluster>
```

6 Łaty

6.1 patch-linux.2.6.11.12-exit_mm.diff

```
--- linux-2.6.11.12-old/kernel/exit.c  2005-06-16 15:12:44.000000000 +0200
+++ linux-2.6.11.12-new/kernel/exit.c  2005-06-16 12:58:59.000000000 +0200
@@ -508,6 +508,8 @@
     mmput(mm);
 }

+EXPORT_SYMBOL(exit_mm);
+
 static inline void choose_new_parent(task_t *p, task_t *reaper, task_t *child_reaper)
 {
     /*
```

6.2 patch-opendlm.0.9.3-exports.diff

```
--- opendlm-0.9.3-old/src/api/api_vms.c 2005-04-22 21:17:03.000000000 +0200
+++ opendlm-0.9.3-new/src/api/api_vms.c 2005-06-16 11:38:16.000000000 +0200
@@ -706,3 +706,10 @@

     return ret_val;
 }
+
+#ifdef __KERNEL__
+#ifdef MODULE
+EXPORT_SYMBOL(dlmlock);
+EXPORT_SYMBOL(dlmunlock);
+#endif /* MODULE */
+#endif /* __KERNEL__ */
```

Literatura

[BA89] J.M. Bernabeu and M. Ahamad. Applying a path-compression technique to obtain an effective distributed mutual exclusion algorithm. Technical report, Proceedings of the 3rd International Workshop on Distributed Algorithms, Nice, France, September 1989.

[CR83] O.S.F. Carvalho and G. Roucairol. On mutual exclusion in computer networks. Technical report, Communications of the ACM, Luty 1983.

[CSL90a] Y.-I. Chang, M. Singhal, and M. Liu. A fault-tolerant algorithm for distributed mutual exclusion. Technical report, Proceedings of the 9th Symposium on Reliable Distributed Systems, Listopad 1990.

[CSL90b] Y.-I. Chang, M. Singhal, and M. Liu. An improved O(log n) mutual exclusion algorithm for distributed systems. Technical report, International Conference on Parallel Processing, pp. III295-302, 1990.

[CSL91] Y.-I. Chang, M. Singhal, and M. Liu. A dynamic token-based distributed mutual exclusion algorithm. Technical report, 10th Annual International Phoenix Conference on Computers and Communications, Marzec 1991.

[Din89] A. Dinning. A Survey of Synchronization Methods for Parallel Computers. Technical report, IEEE Computer, July 1989. strony 66-77.

[DML] JINI DML. JINI DLM – JINI Distributed Lock Manager. Project home page at freshmeat.net, http://osx.freshmeat.net/projects/jini-dlm/.

[Ens] Ensemble. Ensemble - The Ensemble Distributed Communication System. Project home page, http://dsl.cs.technion.ac.il/projects/Ensemble/.

[Era95] S. Eranian. Un service de synchronisation distribuee tolerant les pannes: implantation dans CHORUS. These de doctora, Universite Denis Diderot Paris VII, ftp://sweet-smoke.ufr-info-p7.ibp.fr/divers/eranian/chorus/final.ps.gz, Wrzesień 1995.

[GFS] GFS. GFS – Global File System. Project home page, http://sources.redhat.com/cluster/gfs/.

[Gos91] A. Goscinski. *Distributed Operating Systems - The logical Design, Chapter 6: Synchronization*. Addison Wesley Publishing Company, 1991.

[GUL] GULM. GULM – Grand Unified Lock Manager. Project home page, http://sources.redhat.com/cluster/gulm/.

[JIN] Blitz JINI. Project home page, http://www.dancres.org/blitz/.

[Joh94] T. Johnson. A Performance Comparison of Fast Distributed Synchronization Algorithms. Technical report, Univ. of Florida, Dept. of CIS, TR94-032, ftp://sweet-smoke.ufr-info-p7.ibp.fr/divers/eranian/chorus/final.ps.gz, 1994.

[Lam78] L. Lamport. Time, clocks and the ordering of events in a distributed system. Technical report, Communications of the ACM, Vol. 21, No. 7, pp. 558-565, July 1978.

[Mae85] M. Maekawa. A sqrt(n) Algorithm for Mutual Exclusion in Decentralized Systems. Technical report, ACM Transaction on Computer Systems, Vol. 3, No. 2, pp. 145-159, May 1985.

[Mar85] A.J. Martin. Distributed Mutual Exclusion on a Ring of Processors. Technical report, Scientific Computer Programming 5, 1985.

[MBB92] K. Makki, K. Been, and N. Banta. On algorithms for mutual exclusion in distributed systems. International Conference on Parallel Processing pp. II149-152, 1992.

[MPI95] MPI. The Message Passing Interface. Project home page, http://www.mcs.anl.gov/Projects/mpi/, 1995.

[NLM89] S. Nishio, K.F. Li, and E.G. Manning. A resisilient mutual exclusion algorithm for computer networks. Technical report, IEEE Transactions on Parallel Distributed Systems 7, pp. 61-77, Luty 1989.

[NM90] M.L. Neilsen and M. Mizuno. A DAG-based algorithm for distributed mutual exclusion. Technical report, Proceedings of the 11th International Conference on Distributed Systems, July 1990.

[NT87] M. Naimi and M. Trehel. An improvement of the log(n) distributed algorithm for mutual exclusion. Technical report, 7th International Conference on Distributed Computing, pp. 371-375, 1987.

[Ope] OpenDLM. OpenDLM – Open Distributed Lock Manager. Project home page, http://opendlm.sourceforge.net/, http://sources.redhat.com/cluster/dlm.

[Pro] Claster Project. Project home page, http://sources.redhat.com/cluster/.

[RA81] L. Ricart and K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. Technical report, Communications of the ACM, Vol. 24, No. 1, pp. 9-17, January 1981.

[Ray89] K. Raymond. A Tree-Based Algorithm for Distributed Mutual Exclusion. Technical report, ACM Transaction on Computer Systems, Vol. 7, No. 1, pp. 61-77, 1989.

[RS94] M. Ramachandran and M. Singhal. On the Synchronization Mechanisms in Distributed Shared Memory Systems. Technical Report OSU-CISRC-10/94-TR54, Jan 1994.

[RS95] M. Ramachandran and M. Singhal. A Consensus-Based approach to implementing semaphores in a Distributed Environment. Technical report, The Ohio State University, Department of Computer and Information Science, TR01-95, ftp://archive.cis.ohio-state.edu/pub/tech-report/1995/TR01.ps.gz, 1995.

[Sin93] M. Singhal. *A Taxonomy of Distributed Mutual Exclusion.* Journal of Parallel and Distributed Computing 18, strony 94-101, 1993.

[SK85] I. Suzuki and T. Kasami. A distributed mutal exclusion algorithm. Technical report, ACM Transaction on Computer Systems, Vol. 3, No. 4, pp. 344-349, May 1985.

[SYN] SYNClib. SYNClib - Scalable Distributed Synchronization Library. Project home page, http://www.lfbs.rwth-aachen.de/~karsten/projects/SYNClib/.