Komunikacja międzywęzłowa w rozwiązaniach klastrowych

Projekt z przedmiotu RSO

Michał Lech
M.Lech@elka.pw.edu.pl

16 czerwca 2005

Spis treści

1 Wstęp.

Model sieciowy klastra OpenSSI składa się z dwóch części. Po pierwsze, każdy węzeł ma jeden lub kilka adresów, które są widoczne tylko lokalnie. Jeden adres używany jest w komunikacji typu kernel-kernel, w celu zapewnienia abstrakcji pojedynczego bytu (*SSI*). Ten adres przechowywany jest w */etc/clustertab*. Adres wewnątrzklastrowy może być także używany przez MPI oraz inne aplikacje komunikujące się na poziomie węzłów (ang. *cross-node communicationr*).

Po drugie, aby klaster mógł wyglądać z zewnątrz jak pojedynczy byt, istnieje adres CVIP *Cluster Virtual IP*. Adres ten widoczny jest w sieci zewnętrznej i jest aliasem zewnętrznej karty sieciowej.

Skupie się teraz na opisie pierwszej części modelu.

W celu zapewnienia mechanizmu komunikacji międzywęzłowej twórcy klastra OpenSSI stworzyli *Internode Communication Subsystem (ICS)*.

2 Własności *Internode Communication Subsection (ICS)*.

- zaprojektowany do komunikacji kernel-kernel;
- zapewnia start połączeń przed/po inicjalizacji stosu TCP/IP;
- może być używany w *luźno powiązanych* (ang. *loosely coupeld*) środowiskach klastrowych;
- współpracuje z systemem *Claster Membership (CLMS)*, w celu zapewnienia środowiska *mocno powiązanego* (ang. *tightly coupled*), w którym wszystkie węzły zgadzają się na istnienie listy członków klastra i zapewniona jest komunikacja między wszystkimi węzłami;
- pomiędzy każdym węzłem istnieje zbiór kanałów komunikacyjnych, a sterowanie przepływem odbywa się na poziomie tych kanałów;
- wspiera zróżnicowany rozmiar wiadomości (wiadomości o rozmiarze co najmniej 64K);
- kolejkowanie wiadomości wychodzących;
- priorytety wiadomości w celu uniknięcia zastojów (ang. *deadlock*);
- kolejkowanie wiadomości przychodzących;
- wsparcie dla bezpiecznego odłączania węzła z klastra;
- posiada zależne i niezależne sieciowo elementy;
- wspiera trzy paradygmaty komunikacji:

  wiadomości w jednym kierunku,

  tradycyjne *RPC*, gdzie klient musi synchronicznie czekać na odpowiedź,

  żądanie/odpowiedź albo asynchroniczne *RPC*, w którym odbiorca sam określa moment oczekiwania na odpowiedź;
- bardzo prosty język generacji *ICSgen*;
- współpracuje z *XDR/RPCgen*;
- zapewnia dostarczanie sygnałów od węzła klienta do węzła realizującego usługę, w celu wykonania przerwania lub kontroli wykonania zadania.

3 Wiadomości i odpowiedzi w *ICS*.

Wiadomości i odpowiedzi składają się z:

- bufora *in-line* o stałej długości (*ICS_MAX_INLINE_DATA_SIZE*);
- od zera do siedmiu buforów *out-of-line (OOL)* o nieograniczonej długości.

Ze względu na właściwości protokołu *TCP*, dane *OOL doczepiane są* na końcu bufora *in-line*.

W obecnym kontekście przez argumenty wejściowe rozumie się dane w wiadomościach przesyłanych do serwera (zarówno dane *in-line* jak i *out-of-line*). Argumenty wejściowe dotyczą zarówno wiadomości jak i *RPC*. Przez argumenty wyjściowe rozumie się natomiast dane zawarte w odpowiedzi od serwera (zarówno dane *in-line* jak i *out-of-line*). Argumenty wyjściowe dotyczą tylko *RPC*.

Aby kod kliencki mógł wysyłać wiadomości (i czekać na odpowiedzi) oraz, aby kod serwera mógł obsłużyć wiadomości od klienta, stosowane są uchwyty. Uchwyty przechowują:

- stan transakcji,
- bufor *in-line* oraz informacje o używanych buforach *out-of-line*.

Wiele elementów systemu *ICS* korzysta z funkcji typu *callback*. Są to funkcje wywoływane przez niskopoziomowy *ICS* w momencie wystąpienia określonych zdarzeń. Każdy *callback* ma argument określany przez *callarg*. Argument tego typu może być używany tylko przez funkcje *callback*.

4 Generowanie kodu *ICS* przy użyciu narzędzia *icsgen*.

Na najwyższym poziomie, interfejs między systemem *ICS* i innymi komponentami *SSI* stanowi zbiór serwisów, z których każdy zapewnia dobrze zdefiniowaną funkcjonalność. Owe serwisy definiowane są za pomocą specjalnego języka definicyjnego.

Definicje serwisów znajdują się w plikach źródłowych przetwarzanych przez narzędzie nazywane *icsgen*. *Icsgen* na podstawie plików źródłowych generuje kolejne pliki źródłowe zawierające:

- pliki nagłówkowe z deklaracjami funkcji, które mają zostać utworzone - prototypy te zapewniają kompatybilny interfejs do *ICS*,
- pliki nagłówkowe zawierające makro interfejs nazywany *ICS-supported services*,
- funkcje mapujące operacje pomiędzy warstwami *ICS* i pomiędzy węzłami,
- tablice do wewnętrznego użytku *ICS*.

5 Wysoko-poziomowy *ICS*.

Wysoko-poziomowy *ICS* często jest określany po prostu mianem interfejsu *ICS*. Interfejs ten jest używany do komunikacji pomiędzy węzłami. Można w nim wyróżnić następujące kategorie funkcji:

- funkcje ogólnego przeznaczenia inicjalizujące *ICS*,
- funkcje klienckie używane przez byty *stubs* generowane przez *icsgen*,
- kod kontrolny serwerów, kontrolujący tworzenie i niszczenie uchwytów i demonów,
- funkcje serwera używane przez byty *stubs* generowane przez *icsgen*.

5.1 Serwisy i kanały komunikacyjne.

Komponenty *SSI* do komunikacji używają serwisów. Serwisy klastrowe wykorzystują kanały komunikacyjne *ICS* jako warstwę transportową. Możliwe jest zgrupowanie kilku serwisów jako podserwisów współdzielących ten sam kanał *ICS*. Wiadomości i odpowiedzi są przesyłane za pomocą kanałów komunikacyjnych. Kanał *ICS* jest w rzeczywistości połączeniem TCP/IP, nawiązywanym pomiędzy węzłami klastra do komunikacji wewnątrz-klastrowej. Istnieje *ICS_NUM_CHANNELS (ics_num_channels)* oddzielnych kanałów. Są one identyfikowane przez swój numer. Wiadomości i *RPC* przesyłane przez serwisy są mapowane na odpowiednie kanały komunikacyjne.

Od następnego akapitu rozpocznie się opis mechanizmu dławienia (ang. *throttling*), który nie zostałjeszcze zaimplementowany.

Serwisy są połączone w grupy na danym kanale, ponieważ istnieje możliwość sterowania przepływem na kanałach. *Zdławienie* kanału polega na całkowitym zatrzymaniu komunikacji przez niego przechodzącej. Mechanizm ten jest wykorzystywany w przypadku, gdy klientowi lub serwerowi zaczyna brakować zasobów/pamięci.

Dławienie kanałów komunikacyjnych może prowadzić do zastojów (ang. *deadlock*). Częsta komunikacja międzywęzłowa sprzyja występowaniu zastojów. Na przykład: węzeł A wysyła wiadomość *RPC* do węzła B. W odpowiedzi usługa serwerowa na węźle B wysyła wiadomość *RPC* do węzła A. W tym momencie mogło się zdarzyć, że kanał *ICS* po stronie węzła A został *zdławiony*, więc odpowiedź nie mogła dotrzeć. W niekorzystnych okolicznościach może się okazać, iż spowoduje to niemożliwość zwolnienia pewnych zasobów na węźle A.

Aby temu zapobiegać posłużono się mechanizmem priorytetów. Gdy pojawia się niebezpieczeństwo zaistnienia takiej sytuacji system *ICS* automatycznie podnosi priorytet odpowiedzi.

Część kanałów posiada priorytety i kanały te nie mogą być *dławione*. Żadne serwisy nie są mapowane na te kanały. Aby wiadomość lub *RPC* mogło skorzystać z jednego z tych kanałów, musi zostać użyta funkcja *ics_setpriority()*, podnosząca priorytet aktywnego wątku.

Element *ics_prio*, znajdujący się w strukturze danych *task_struct*, zawiera wartość priorytetu, która określa, jakiego kanału wolno użyć. Wartość 0 oznacza, że serwis użyje przypisanego mu kanału. Każda wartość powyżej 0 spowoduje, że wiadomość zostanie przesłana jednym z czterech kanałów priorytetowych.

W celu przesyłania odpowiedzi *RPC* zostały zdefiniowane dwa kanały *ICS*. Kanał *ics_reply_chan* jest używany do przesyłania zwykłych wiadomości *RPC*, natomiast

kanał *ics_reply_prio_chan* jest używany do przesyłania priorytetowych wiadomości *RPC*. Żaden z tych kanałów, podobnie jak w przypadku priorytetowych kanałów *ICS*, nie może zostać *zdławiony*.

W klastrze może być obecnych maksymalnie *ICS_MAX_CHANNELS* kanałów *ICS*. Wartość tą można modyfikować w pliku nagłówkowym *ics.h*. Istnieje 8 różnych typów kanałów *ICS*. Jak już wspomniano są cztery kanały priorytetowe, dwa kanały niskopoziomowe związane z przesyłaniem odpowiedzi *RPC*. Dodatkowo istnieją jeszcze dwa kanały zdefiniowane dla klastrowych serwisów. Jeden z nich jest używany m.in. przez system *CLMS* i *API* klastrowe. Drugi z nich jest tylko na użytek *CLMS*.

5.2 Kod kontrolny po stronie serwera.

Kod kontrolny serwera jest jedną z bardziej znaczących części systemu *ICS*. Zapewnia on kontrolę nad serwerowymi demonami i serwerowymi uchwytami.

W przypadku demonów, kod kontrolny stara się dostosować ilość istniejących demonów wyrażoną przez *icsdeamon_avail_lwm* do wymaganej liczby *icsdeamon_avail_hwm* demonów w stanie dostępności (nie przetwarzających w danym momencie ani wiadomości ani *RPC*). Jeśli liczba demonów jest mniejsza niż *icsdeamon_avail_hwm*, tworzone są dodatkowe demony. Jeśli natomiast jest odwrotnie, tzn. liczba demonów jest większa od *icsdeamon_avail_hwm*, demony nadmiarowe są niszczone. Demony są używane zarówno do obsługi normalnych jak i priorytetowych wiadomości *RPC*. Istnieją również specjalne demony dla każdego priorytetu większego niż jeden. W momencie, gdy przychodzi wiadomość lub *RPC* o wysokim priorytecie, a nie ma dostępnych zwykłych demonów, jej obsługą zajmie się demon specjalny przeznaczony dla odpowiedniego priorytetu.

W przypadku serwerowych uchwytów, kod kontrolny stara się dostosować ilość istniejących uchwytów wyrażoną przez *lwm_svc_handles* do wymaganej liczby *hwm_svchandles* uchwytów znajdujących się w stanie *icssvr_handle_recv()*. Jeśli ilość uchwytów w takim stanie będzie mniejsza od *hwm_svc_handles* wtedy nastąpi utworzenie dodatkowych uchwytów. Jeśli wystąpi sytuacja odwrotna, tzn. takich uchwytów będzie więcej niż *hwm_svc_handles* wtedy nastąpi zniszczenie nadmiarowych uchwytów.

5.3 Typowe scenariusze działania po stronie klienta.

Przedstawione przykłady są charakterystyczne dla kodu generowanego przez *icsgen*. Wiadomości do serwera nie mogą być wysyłane z kontekstu obsługi przerwania.

Pierwszy przykład dotyczy sytuacji, w której wątek klienta wykonuje *RPC*.

- Uzyskanie uchwytu klienta, przy użyciu *icscli_handle_get()*.
- Przygotowanie argumentów wejściowych, przy użyciu rodziny funkcji *icscli_encode_ * ()*.
- Przygotowanie informacji potrzebnych dla argumentów wyjściowych *OOL*, przy użyciu rodziny funkcji *icscli_encoderesp_ool_ * ()*.
- Wysłanie wiadomości do serwera, przy użyciu *icscli_send()*. Funkcja *isccli_wait_callback* jest typowo używana jako operacja typu *callack*.
- Oczekiwanie nie odpowiedź, typowo, przy użyciu *icscli_wait()*.

- Odczytanie argumentów wyjściowych z odpowiedzi, przy użyciu rodziny funkcji *icscli_decode_ * ()*.
- Zwolnienie uchwytu klienta, przy użyciu *icscli_handle_release()*.

Drugi przykład dotyczy sytuacji, w której wątek klienta wysyła wiadomość do serwera.

- Uzyskanie uchwytu klienta, przy użyciu *icscli_handle_get()*.
- Przygotowanie argumentów wejściowych, przy użyciu rodziny funkcji *icscli_encode_ * ()*.
- Wysłanie wiadomości do serwera, przy użyciu *icscli_send()*. Funkcja *isccli_wait_callback* jest typowo wybierana jako operacja typu *callack*.

Trzeci przykład dotyczy sytuacji, w której wątek klienta wykonuje *broadcast RPC*. Polega to na wykonaniu *RPC* do określonych węzłów.

- Uzyskanie uchwytów klientów, przy użyciu *icscli_handle_get()* (po jednym dla każdego węzła docelowego).
- Przygotowanie argumentów wejściowych, przy użyciu rodziny funkcji *icscli_encode_ * ()* (kodowanie argumentów musi się odbywać dla każdego uchwytu).
- Przygotowanie informacji potrzebnych dla argumentów wyjściowych *OOL*, przy użyciu rodziny funkcji *icscli_encoderesp_ool_ * ()* (ponownie dla każdeg o uchwytu).
- Wysłanie wiadomości do serwerów, przy użyciu *icscli_send()* wywoływanej dla każdego uchwytu. Funkcja *isccli_wait_callback* jest typowo wybierana jako operacja typu *callack*.
- Wykonanie lokalnych operacji.
- Oczekiwanie na odpowiedzi od wszystkich serwerów, przy użyciu *icscli_wait()* dla każdego uchwytu.
- Odczytanie argumentów wyjściowych z odpowiedzi, przy użyciu rodziny funkcji *icscli_decode_ * ()* (dekodowanie argumentów musi się odbywać dla każdego uchwytu).
- Zwolnienie uchwytów klientów, przy użyciu *icscli_handle_release()*.

5.4 Typowe scenariusze działania po stronie serwera.

Kod kontrolny serwera jest odpowiedzialny za uzyskanie właściwej liczby serwerowych uchwytów, przy pomocy funkcji *icssvr_handle_get()*. Następnie na rzecz każdego z tych uchwytów wywoływana jest funkcja *icssrv_recv()*. Jeśli serwer będzie obsługiwany z kontekstu wątku, to kod kontrolny serwera jest odpowiedzialny za stworzenie odpowiedniej ilości demonów, aby każda otrzymana wiadomość mogła zostać obsłużona w odpowiedni sposób.

Pierwszy przykład dotyczy obsługi przez serwer i*RPC*. W przykładzie zakłada się, iż demon został obudzony za pomocą odpowiedniej funkcji *callback* i to on zajmuje się obsługą *RPC*.

- Odczytanie argumentów z wiadomości, przy użyciu rodziny funkcji *icssrv_decode_*()*.
- Wywołanie funkcji *icssrv_decode_done()* w celu potwierdzenia zakończenia dekodowania.
- Wykonanie odpowiednich zadań po stronie serwera.
- Przygotowanie odpowiedzi, przy użyciu rodziny funkcji *icssrv_encode_ * ()*.
- Wysyłanie odpowiedzi, za pomocą funkcji *icssrv_reply()* oraz wywołanie *icssrv_recv()*.
- Uśpienie demona w oczekiwaniu na kolejną wiadomość.

Drugi przykład dotyczy obsługi przez serwer *IPC* (np. wiadomości od klienta). Założenia jak w poprzednim przykładzie.

- Odczytanie argumentów z wiadomości, przy użyciu rodziny funkcji *icssrv_decode_*()*.
- Wywołanie funkcji *icssrv_decode_done()* w celu potwierdzenia zakończenia dekodowania.
- Natychmiastowa odpowiedź do klienta, przy użyciu funkcji *icssrv_reply()* oraz wywołanie *icssrv_recv()*.
- Wykonanie odpowiednich zadań po stronie serwera.
- Uśpienie demona w oczekiwaniu na kolejną wiadomość.

Trzeci przykład także dotyczy obsługi *IPC* przez serwer, ale działa on w kontekście procedury obsługi przerwania.

- Odczytanie argumentów z wiadomości, przy użyciu tych funkcji z rodziny *icssrv_decode_*()*, których można bezpiecznie użyć w kontekście procedury obsługi przerwania.
- Wywołanie funkcji *icssrv_decode_done()* w celu potwierdzenia zakończenia dekodowania.
- Natychmiastowa odpowiedź do klienta, przy użyciu funkcji *icssrv_reply()* oraz wywołanie *icssrv_recv()*.
- Wykonanie odpowiednich zadań po stronie serwera, ale takich które mogą być wykonywane w procedurze obsługi przerwania.
- Powrót z obsługi przerwania.

6 Nisko-poziomowy *ICS*.

Nisko-poziomowa część systemu *ICS* jest zależna od warstwy transportowej, tzn. kod implementujący tą część systemu będzie się różnił w zależności od użytego protokołu transportowego.

Nisko-poziomowe funkcje *ICS* mogą być używane tylko przez wysoko-poziomową część systemu.

Komunikacja między wyższym i niższym poziomem korzysta z uchwytów. Używa się dwóch typów uchwytów:

- uchwyty klientów (typ *cli_handle_t*),
- uchwyty serwerów (typ *svr_handle_t*).

Uchwyty zostały podzielone na dwie sekcje: część wysoko-poziomową i nisko-poziomową. Nisko-poziomowa część uchwytów klientów jest zadeklarowana jako *ch_llhandle* typu *cli_llhandle_t* w strukturze *cli_handle_t*. Nisko-poziomowa część uchwytów serwerów jest zadeklarowana jako *sh_llhandle* typu *svr_llhandle_t* w strukturze *svr_handle_t*. Część nisko-poziomowa uchwytów jest całkowicie niewidoczna przez wysoko-poziomowy *ICS*. Alokowana jest w *icscli_handle_get()* / *icssrv_handle_get()*. Niektóre elementy części wysoko-poziomowej wykorzystywane są do komunikacji z poziomem niskim (*icscli_llsend()*, *icscli_sendup_reply*, *icssrv_sendup_msg()*, *icssrv_llreply()*). Pozostałe elementy wysoko-poziomowe są niewidoczne dla części nisko-poziomowej.

Nisko-poziomowa część *ICS* może w dowolny sposób korzystać z części uchwytów nisko-poziomowych.

6.1 Serwisy i nisko-poziomowe kanały komunikacyjne.

Interfejsy pomiędzy wysoko-poziomowym i nisko-poziomowym *ICS* określają, których kanałów ma użyć nisko-poziomowy ICS do przesyłania wiadomości i odpowiedzi. Jak wspomniano wcześniej istnieje *ICS_NUM_CHANNELS* (*ics_num_channels*) oddzielnych kanałów używanych przez węzły do komunikacji. Identyfikowane są one za pomocą numerów: od 0 do *ICS_NUM_CHANNELS* - 1 (*ics_num_channels* - 1). Z punktu widzenia wysoko-poziomowego *ICS*, te kanały mają różne przeznaczenie, chociaż nisko-poziomowy *ICS* traktuje je identycznie.

Ogólnie rzecz biorąc, *ICS* nie wysyła wiadomości/odpowiedzi do innego węzła aż do chwili użycia przez węzeł *ics_nodeup()* (i co za tym idzie *ics_llnodeup()*). Dzięki temu nisko-poziomowy *ICS* może użyć *ics_llnodeup()* w celu przeprowadzenia ustawień (*setup*) dla wszystkich kanałów. W przypadku jednak kanałów priorytetowych i priorytetowego kanału dla odpowiedzi, komunikacja może mieć miejsce po użyciu *ics_seticsinfo()* (i co za tym idzie *ics_llseticsinfo()*) (w celu *bootowania*). Nisko-poziomowy *ICS* musi jeszcze zezwolić na komunikację przez te kanały zgodnie z wcześniej podanymi warunkami.

Kanały komunikacyjne mogą być *dławione* (*throttled*). Wysoko-poziomowy *ICS* decyduje, kiedy należy *zdławić* komunikacje oraz kiedy ją wznowić. Nisko-poziomowy *ICS* zapewnia taką możliwość sterowania przepływem (tzn. wysoko-poziomowy *ICS* jest odpowiedzialny za implementację polityki *dławienia*, natomiast nisko-poziomowy *ICS* jest odpowiedzialny za implementację samego mechanizmu sterowania przepływem).

Decyzja odnośnie ograniczania komunikacji jest podejmowana na podstawie informacji o poziomie wykorzystania zasobów przez węzeł serwera. Kiedy następuję *zdławienie* kanału, serwer przestaje przyjmować wiadomości/*RPC*. Dławienie jest przeprowadzane z wykorzystaniem funkcji: *icscli_llwaitfornothrottle()*, *icscli_wouldthrottle*, *icssrv_find_recv_handle()*, *icssrv_llhandles_present()*.

Nisko-poziomowy *ICS* może mieć dodatkową politykę dotyczącą *dławienia* kanałów komunikacyjnych, ale nie może ona powodować zastojów (*deadlock*).

Dodatkowo mogą się zdarzyć rozbieżności czasowe dotyczące chwili rozpoczęcia *dławienia* kanału przez serwer i chwili zauważenia tego stanu przez klienta. Taka sytuacja jest akceptowalna o ile żadne wiadomości/*RPC* nie zostaną utracone.

Nisko-poziomowy *ICS* jest całkowicie nieświadomy istnienia wysoko-poziomowych serwisów i priorytetów. Zajmuje się on jedynie komunikacją.

6.2 Typowe scenariusze działania po stronie klienta.

Zostanie teraz przedstawiony typowy sposób współdziałania pomiędzy wysoko-poziomowym i nisko-poziomowym *ICS*. Przedstawiony przykład dotyczy każdego z wcześniej omówionych przykładów dla wysoko-poziomowego *ICS* po stronie klienta.

- Po użyciu funkcji *icscli_handle_get()*, w celu inicjalizacji uchwytów, używana jest funkcja *icscli_llhandle_init()*.
- Każde wywołanie funkcji z rodziny *icscli_encode_ * ()* w celu zakodowania argumentów wejściowych, spowoduje wywołanie odpowiadającej funkcji *icscli_llencode_ * ()*.
- Gdy klient woła funkcje *icscli_send()*, pociąga to za sobą wywołanie funkcji *icscli_llsend()*. W razie potrzeby nisko-poziomowy *ICS* korzysta z funkcji *icscli_sendup_reply()*, która jest odpowiedzialna za wykonanie funkcji *callback* dla *icscli_send()*. Jeśli to jest RPC, nisko-poziomowy *ICS* wywoła *icscli_find_transid_handle()*, aby ustalić uchwyt, którego dotyczy odpowiedź.
- Jeśli pierwotną akcją było *RPC*, to wtedy należy zdekodować argumenty. Każde użycie funkcji z rodziny *icscli_decode_ * ()*, spowoduje użycie odpowiadającej funkcji *icscli_lldecode_ * ()*.
- Kiedy następuje zwolnienie uchwytu klienta za pomocą *icscli_handle_release()*, wywołana zostanie funkcja *icscli_llhandle_deinit()*.

6.3 Typowe scenariusze działania po stronie serwera.

Zostanie teraz przedstawiony typowy sposób współdziałania pomiędzy wysoko-poziomowym i nisko-poziomowym *ICS*. Przedstawiony przykład dotyczy każdego z wcześniej omówionych przykładów dla wysoko-poziomowego *ICS* po stronie klienta.

- Funkcja alokująca uchwyty serwerowe *icssrv_handle_get()* wywołuje *icssrv_llhandle_init()*.

- Wysoko-poziomowy *ICS* woła funkcje *icssrv_llhandle_init_for_recv()* (typowo z *icssrv_recv()*). Jeśli *icssrv_llhandle_init()* zwróci *NULL*, to *icssrv_llhandle_init()* będzie wywołana ponownie, z kontekstu wątku, gdzie nie może zwrócić *NULL*.

- Kolejny krok różni się w zależności od tego, czy nisko-poziomowy *ICS* potrzebuję uchwytu, aby odebrać wiadomość, czy nie. Jeśli potrzebuje uchwytu, to, aby go otrzymać, wywołuje *icssrv_handle_recv()*. W przeciwnym razie, wywołanie *icssrv_handle_recv()* następuje dopiero, gdy serwer otrzyma wiadomość. W obu wiec przypadkach, każdej odebranej wiadomości będzie odpowiadał jeden uchwyt (poza przypadkiem, gdy *icssrv_handle_recv()* zwróci NULL).

- Nisko-poziomowy *ICS* używa *icssrv_sendup_msg()*, gdy przybywa wiadomość i argumenty wejściowe są gotowe do zdekodowania.

- Każde użycie funkcji z rodziny *icssrv_decode_ * ()* spowoduje użycie odpowiadającej funkcji *icssrv_lldecode_ * ()*.

- Funkcja *icssrv_lldecode_done()* jest typowo wołana z funkcji *icssrv_decode_done()*.

- Jeśli pierwotną akcją było *RPC*, to wtedy są wyjściowe argumenty do zakodowania. Każde użycie funkcji z rodziny *icssrv_encode_ * ()* spowoduje użycie odpowiadającej jej funkcji *icssrv_encode_ * ()*.

- Funkcja *icssrv_llreply()* będzie wywołana z *icssrv_reply()*. Kiedy nisko-poziomowy *ICS* nie potrzebuje już uchwytu, wywoła *icssrv_sendup_replydone()*. Funkcja ta jest odpowiedzialna za zapewnienie, iż zostanie wywołany *callback* dla *icssrv_reply()*.

- Jeżeli wysoko-poziomowy *ICS* zdecyduje, iż zajęty uchwyt będzie ponownie wykorzystany, schemat zostanie powtórzony od punktu drugiego. W przeciwnym wypadku nastąpi zwolnienie uchwytu, przy pomocy funkcji *icssrv_handle_release()*, która wywoła *icssrv_llhandle_deinit()*.

7 *ICS* w statystyce

W celu zaprezentowania działania systemu *ICS*, napisałem skrypt, który wydobywa statystykę, dotyczącą komunikacji międzywęzłowej (plik *ics_stat*. Można go uruchomić z trzema opcjami:

- s - Wyświatla statystykę związaną z komunikacją po stonie serwera. Są to ilość uchwytów serwerowych stworzonch i obecnych w danej chwili oraz ilość wiadomości i *RPC* otrzymanych przez serwery.

- k - Wyświatla statystykę związaną z komunikacją po stonie klienta. Są to ilość uchwytów klienta obecnych w danej chwili oraz ilość wiadomości i *RPC* otrzymanych przez klientów.

- g - Wyświatla globalną statystykę dotyczącą komunikacji. Są to nazwa kanału *ICS* (numer kanału), nazwa serwisu na danym kanale i nazwa operacji związana z danym serwisem. Drugie wystąpienie tej samej operacji wraz z

wartością liczbową pokazuje ile ta operacja po stronie klienta wysłała *RPC*. Trzecie wystąpienie tej operacji także z wartością liczbową pokazuje ile ta operacja po stronie serwera odebrała *RPC*.

Literatura

[com] *http://ci-linux.sourceforge.net/components.shtml*.

[enh] *http://ci-linux.sourceforge.net/enhancing.shtml*.

[ics] *http://ci-linux.sourceforge.net/ics.shtml*.

[int] *http://openssi.org/cgi-bin/view?page=docs2/1.2/Introduction-to-SSI*.