STUDIA INFORMATICA 2003
Volume 24 Number 4 (56)

Paweł L. KACZMAREK, Henryk KRAWCZYK
Politechnika Gdańska, Wydział Elektroniki, Telekomunikacji i Informatyki

STRATEGIE OBSŁUGI WYJĄTKÓW W APLIKACJACH ROZPROSZONYCH

Streszczenie. Rozpatrzono wykorzystanie mechanizmu obsługi wyjątków w systemach rozproszonych. Zaprezentowano różne strategie obsługi wyjątków dla różnych modeli przetwarzania i odpowiadających im środowisk programistycznych. Przyjęto nową koncepcję zdalnego odbiorcy wyjątków oraz zaprezentowano jego implementację przy wykorzystaniu biblioteki MPI oraz RMI.

Słowa kluczowe: aplikacje rozproszone, obsługa wyjątków, platformy programowania, zdalne wyjątki

EXCEPTION HANDLING STRATEGIES IN DISTRIBUTED APPLICATIONS

Summary. Utilisation of exception handling mechanisms in distributed applications is considered. Different exception handling strategies are presented for various suitable programming platforms. The new concept of remote exception handling is proposed and its implementations in MPI and RMI libraries are given.

Keywords: distributed applications, exception handling, programming platforms, remote exceptions

1. Koncepcja obsługi wyjątków

Mechanizm obsługi wyjątków umożliwia zwiększenie niezawodności programu oraz jakości kodu źródłowego przez rozdzielenie wykonania zwykłego i wyjątkowego. W programie zostają zdefiniowane *regiony chronione*. Jeżeli w czasie wykonywania kodu należącego do regionu chronionego wystąpi błąd, to zgłaszany jest wyjątek i sterowanie

zostaje przekazane do *funkcji obsługi* [3]. Funkcja obsługi usiłuje naprawić niepoprawną sytuację lub zgłosić nieprawidłowość i zakończyć działanie funkcji.

Obsługa wyjątków została rozpowszechniona wraz z obiektowymi językami programowania i zyskała uznanie wśród programistów i projektantów. Generalnie wyróżnia się trzy sposoby przekazania sterowania po zakończeniu funkcji obsługi:

- terminacja - sterowanie jest przekazywane na koniec bloku chronionego, część kodu za miejscem wystąpienia wyjątku nie zostaje wykonana,
- wznowienie - sterowanie powraca do instrukcji za tą, która spowodowała wystąpienie wyjątku,
- powtórzenie - sterowanie powraca na początek bloku chronionego i cała sekwencja jest powtarzana.

Obecnie największą popularność zyskał sposób obsługi zgodnie z modelem terminacji. W ten sposób działają popularne języki programowania jak C++ i Java. W dalszej części pracy przedstawiono koncepcje obsługi wyjątków ze szczególnym uwzględnieniem systemów rozproszonych. W następnym rozdziale przeanalizowano technologię RPC, MPI oraz RMI pod kątem obsługi błędów. W rozdziałach 3 oraz 4 przedstawiono uogólniony model obsługi wraz z projektem wykorzystania w MPI oraz RMI. Rozdział 5 zawiera opis koncepcji specjalizowanego procesu (zdalnego odbiorcy) obsługującego zdalnie zgłaszane wyjątki.

2. Obsługa wyjątków w systemach rozproszonych

W rozdziale przedstawiono możliwości obsługi wyjątków w wybranych platformach programowania rozproszonego. Główną uwagę zwrócono na porównanie mechanizmów, które są udostępniane programiście. W tabeli 1 na końcu rozdziału zebrano najważniejsze cechy i funkcjonalności, które są dostępne w różnych systemach programowania.

2.1. Błędy w wywołaniu zdalnych procedur

RPC (Remote Procedure Call) umożliwia wywołanie przez klienta procedury na zdalnym serwerze. Jeżeli wystąpi błąd w wołaniu RPC, funkcja zwraca kod błędu oraz wypełniana jest struktura rpc_err [2]. Duży zbiór funkcji został zdefiniowany do wyświetlania błędów, komunikatów o błędach itp. w kodzie klienta i serwera.

Możliwości obsługi błędów w RPC zilustrujemy na bazie modelu klient-serwer oraz błędu konwersji danych. Serwer wysyła daną typu „int", podczas gdy klient oczekuje danej typu „double". Rozpatrzmy następujący kod programu:

```
ogólne:
  double d;

serwer:
   ...
   (void)svc_sendreply(t, (xdrproc_t)xdr_int /*xdr_double*/,
          (char *)&d);

klient:
   ...
   if ( clnt_call(c, 1, (const xdrproc_t)xdr_void, 0,
         (const xdrproc_t)xdr_double, (char *)&d, t) != 0 )
   rpc_error(c, "Błąd podczas odpowiedzi");
   ...
```

Błąd występuje w związku z wywołaniem funkcji clnt_call, więc jest faktycznie zgłaszany lokalnie. Serwer nie posiada możliwości samodzielnego powiadomienia klienta o swoim stanie ani o ewentualnych błędach w swoim działaniu. Jeżeli serwer zaprzestanie działania, to klient otrzyma jedynie błąd „Connection refused" (odrzucone połączenie).

2.2. Obsługa wyjątków w MPI

W MPI (Message Passing Interface) [7] możliwe jest zdefiniowanie funkcji, która jest wywoływana za każdym razem, gdy wystąpi wyjątek. Grupa funkcji MPI_Errhandler(_create, _set) odpowiada za zdefiniowanie funkcji obsługi oraz związanie jej z konkretnym komunikatorem. Błędy w bibliotece MPI dzielimy na dwa rodzaje:

- błędy programu - wywołanie funkcji MPI posiada złe parametry,
- błędy zasobów - program usiłuje zaalokować zasoby, których system nie jest w stanie udostępnić.

Jednakże sposób obsługi w MPI umożliwia wyłącznie obsługę błędów lokalnych, a ponadto błędy muszą być związane z konkretną operacją. Wiele funkcji MPI jest wykonywanych asynchronicznie (np. nieblokujące wysyłanie), to znaczy, że operacja jest kontynuowana po wyjściu z funkcji, która ją aktywowała [5]. W przypadku takich operacji funkcja obsługi błędu nie zostanie wywołana.

Aby sprawdzić działanie funkcji obsługi błędów w MPI, wykorzystaliśmy typowy błąd polegający na wysłaniu informacji do nie istniejącego procesu. Rozpatrzmy następujący kod programu:

```
...
MPI_Comm_size(MPI_COMM_WORLD, &iloscProcesow);
...
MPI_Send (&bufor, 1, MPI_INT, iloscProcesow+1, 0,
      MPI_COMM_WORLD);
```

...

Mechanizm umożliwia przechwycenie lokalnego błędu i wywołanie funkcji zdefiniowanej przez użytkownika. Brak jest możliwości wysłania błędu do innego procesu (np. mastera) w celu zasygnalizowania nieprawidłowości.

2.3. Obsługa zdalnych wyjątków dla języka C++/Java

Obsługa wyjątków w RMI (Remote Method Invocation) [4] bazuje na obsłudze udostępnianej przez C++/Java. Generalnie są wyróżniane dwie grupy wyjątków:

- wyjątki zdalne,
- wyjątki lokalne.

Wyjątki lokalne są obsługiwane jak normalne wyjątki C++/Java, natomiast wyjątki zdalne służą do sygnalizowania błędów podczas pracy serwera lub podczas komunikacji z serwerem. Na rysunku 1 przedstawiono strukturę dziedziczenia dla wyjątków zdalnych i lokalnych [6]. W przypadku wyjątków zdalnych informacja na temat źródła oraz przyczyn wyjątku jest często bardzo ograniczona, więc klient posiada niewielkie możliwości naprawy błędnej sytuacji.

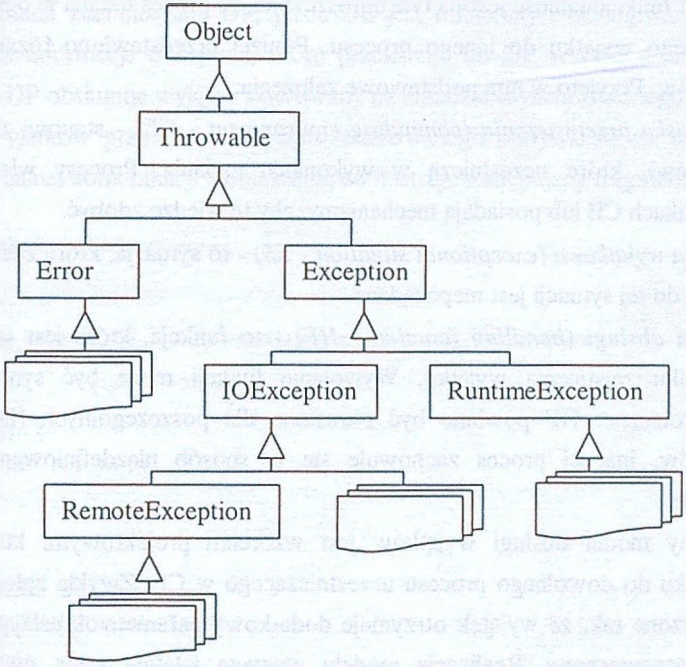
W naszym przypadku testowym wykorzystaliśmy serwer Apache oraz RMI w Java. W celu generacji wyjątku serwer wykonuje dzielenie przez zero albo jawnie zwraca zdalny wyjątek. Rozpatrzmy następujący fragment programu:

```
Serwer:
   public String odpowiedz1() throws RemoteException {
   ...
   RemoteException e = new RemoteException();
   throw e;
   };

  public String odpowiedz2() throws RemoteException {
   ...
   int a;
   a = 1/0;
   };

Klient:
   try{
     ...
     message = obj.odpowiedz1() /*obj.odpowiedz2()*/;
   }
   catch (RemoteException e) { ... }
   catch (Exception e) { ... }
```

Jak widać, możliwe jest przesłanie wyjątków między serwerem a klientem. Jednak stworzony mechanizm często powoduje gubienie informacji, gdyż zdalny wyjątek nie musi zawierać dokładnych informacji o błędzie. Ponadto zauważmy, że klient zostaje zablokowany w wołaniu metody do czasu jej zakończenia przez serwer.



Rys. 1. Struktura dziedziczenia dla Exception oraz RemoteException w Java
Fig. 1. Class Hierarchy for Exception and RemoteException in Java

2.4. Porównanie możliwości obsługi wyjątków

W tabeli 1 przedstawiono porównanie dostępnych strategii obsługi wyjątków w systemach rozproszonych. Zauważmy, że żaden z systemów nie implementuje możliwości przesłania asynchronicznej informacji w przypadku zaistnienia błędu. Istnieje konieczność rozbudowy procedur obsługi wyjątków o nowe możliwości.

Tabela 1

Możliwości obecnie dostępnych mechanizmów obsługi wyjątków

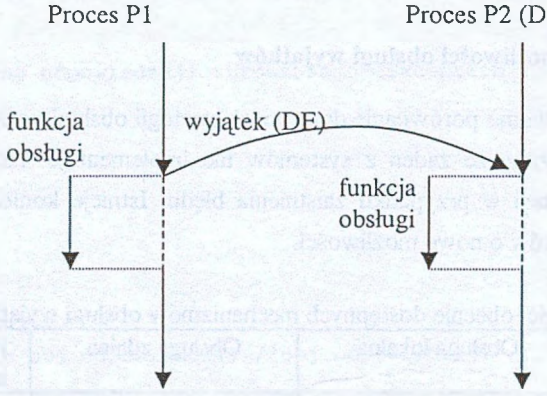
Środowisko programowania	Obsługa lokalna	Obsługa zdalna	Powiadomienie innego procesu
MPI	funkcja obsługi błędu	zwrócenie kodu błędu	brak
RPC	zwrócenie kodu błędu	zwrócenie kodu błędu	brak
RMI	try - catch - throw	try - catch - throw RemoteException	brak

3. Rozszerzony model obsługi wyjątków

Na podstawie modeli obsługi wyjątków dla różnych platform programowania można zauważyć, że wszystkie one dotyczą dwóch procesów: wołającego i wykonującego. W przypadku MPI funkcjonalność jest na tyle uproszczona, że proces nie ma w ogóle możliwości wysłania zdalnego wyjątku do innego procesu. Poniżej przedstawiono rozszerzony model obsługi wyjątków. Przyjęto w nim podstawowe założenia:

- *Środowisko przetwarzania (computing environment - CE)* - stanowi zbiór procesów (obiektów), które uczestniczą w wykonaniu zadania. Procesy wiedzą o innych uczestnikach CE lub posiadają mechanizmy, aby tę wiedzę zdobyć.
- *Sytuacja wyjątkowa (exceptional situation - ES)* - to sytuacja, która zdarza się rzadko, wejście do tej sytuacji jest niepożądane.
- *Funkcja obsługi (handling function - HF)* - to funkcja, która jest wykonywana w przypadku zaistnienia wyjątku. Wywołanie funkcji może być synchroniczne lub asynchroniczne. HF powinna być określona dla poszczególnych (lub wszystkich) wyjątków, inaczej proces zachowuje się w sposób niezdefiniowany lub kończy działanie.

Rozszerzony model obsługi wyjątków jest wzorcem projektowym, który umożliwia wysłanie wyjątku do dowolnego procesu uczestniczącego w CE. Zwykłe zgłoszenie wyjątku zostaje rozszerzone tak, że wyjątek otrzymuje dodatkowy parametr określający, dla jakiego procesu jest przeznaczony. Realizacja modelu wymaga zdefiniowania dwóch pojęć jak pokazano na rys. 2:



Rys. 2. Rozszerzony model obsługi wyjątków
Fig. 2. Extended exception handling model

- *Proces przeznaczenia (destination process DP)* - proces, który otrzymuje zdalny wyjątek. Zazwyczaj jest to proces, który musi być poinformowany o zaistniałym błędzie.
- *Wyjątek skierowany (directed exception DE)* - rozszerzona koncepcja wyjątku (ES). DE posiada zdefiniowany DP, gdzie DE jest odbierany i obsługiwany. DE posiada również informacje o błędzie, które przekazuje do DP w celu wykonania obsługi. Proces DP obsługuje wyjątek skierowany na zasadzie asynchronicznego przerwania.

Obsługa wyjątków przy użyciu wyjątku skierowanego odbywa się na zasadzie obsługi lokalnej oraz zdalnej komunikacji z obiektami, co ilustruje następujący fragment programu:

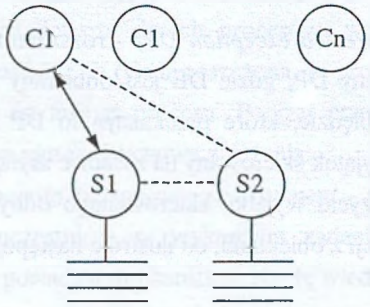
```
--- kontekst lokalny: ---
try {
  ...
  if (ES) throw e;     //lokalne przerwanie wykonania
  ...
}
catch (e) {
  throw DE (e, DP);
  ...
}

--- kontekst zdalny: ---
przypisz_funkcję_obsługi;
...
//asynchroniczne nadejście zdalnego wyjątku
...
funkcja_obsługi {
  ...
  wykonaj_akcje_naprawcze
  ...
};
```

Model rozszerza możliwości języków programowania rozproszonego. Jeżeli proces wysyła wyjątek, który dotyczy całego środowiska wykonania, to może zdefiniować zdalny proces przeznaczenia, który odbierze i obsłuży wyjątek. Informacja przenoszona z wyjątkiem zależy od środowiska pracy, może to być numer błędu, obiekt lub inna dana. Dotychczas sygnały POSIX [1] umożliwiały nawiązanie asynchronicznej komunikacji między procesami. Jednak mechanizm sygnałów dotyczy warstwy systemu operacyjnego i często występują problemy z jego zastosowaniem w językach wyższego poziomu.

Jako przykład zastosowania rozpatrzmy duplikowaną bazę danych. W modelu występują pewna ilość klientów (C_1, C_2, ..., C_n) oraz dwa serwery (S_1, S_2), z których każdy ma dostęp do

swojej kopii danych. Sytuacja wyjątkowa występuje, gdy klient C_i nie może być obsłużony przez serwer S_1 i zostaje obsłużony przez duplikowany serwer S_2 jak pokazano na rys. 3.



Rys. 3. Przykład rozszerzonego modelu obsługi wyjątków
Fig. 3. An example of extended exception handling

Możemy wyróżnić cztery przypadki jak pokazano poniżej. Procesy mogą komunikować się ze sobą na dwa sposoby: zwykłą komunikacją oraz przesyłaniem wyjątków.

$$C_1 \xrightarrow{data} S_1 \xrightarrow{data} C_1$$

$$C_1 \xrightarrow{data} S_{1(error)} \xrightarrow{exception} C_1 \xrightarrow{data} S_2 \xrightarrow{data} C_1$$

$$C_1 \xrightarrow{data} S_{1(error)} \xrightarrow{data} S_2 \xrightarrow{data} S_1 \xrightarrow{data} C_1$$

$$C_1 \xrightarrow{data} S_{1(error)} \xrightarrow{exception} S_2 \xrightarrow{dane} S_1 \xrightarrow{dane} C_1$$

Pierwszy przypadek jest poprawny i nie wymaga specjalnej obsługi. Drugi przypadek umożliwia obsługę przez wysłanie zdalnego wyjątku z serwera S_1 do klienta C_i (podobnie jak w bibliotece RMI). Dalej klient komunikuje się z serwerem S_2. Rozwiązanie pozwala na uzyskanie poprawnych danych, ale komplikuje kod klienta. Klient musi wiedzieć, że może zaistnieć wyjątek przysłany od S_1, a ponadto musi znać serwer S_2.

Trzeci przypadek umożliwia zaimplementowanie serwera odpornego na błędy, jednak komunikacja między dwoma serwerami odbywa się w zwykły sposób. To powoduje, że kod serwera jest niejasny, gdyż nie ma rozróżnienia na zwykłą komunikację i komunikację w przypadku błędów.

Czwarte rozwiązanie implementuje zdalne wysyłanie wyjątków między serwerami S_1 i S_2. Klient jest zwolniony z konieczności zapewnienia niezawodności, a jednocześnie serwery posiadają jasno zdefiniowane obszary funkcjonalne. S_2 obsługuje sytuacje wyjątkowe tylko w przypadku, gdy otrzyma wyjątek i nie musi rozszerzać swojej standardowej funkcjonalności.

Możliwość odbioru zdalnych wyjątków wiąże się z koniecznością zapewnienia bezpieczeństwa systemu. Poziom bezpieczeństwa zależy z pewnością od typu aplikacji i sposobu przetwarzania. Jak łatwo zauważyć, jeżeli proces może wymusić obsługę sytuacji

awaryjnej w systemie, to może również usiłować uszkodzić lub zniszczyć proces odbierający zdalne wyjątki. Jednym z rozwiązań jest przechowywanie listy procesów uczestniczących w CE i odbiór wyjątków jedynie od tych procesów.

W systemach rozproszonych wiele procesów może jednocześnie wykonać te same zadania. W przypadku wysyłania wyjątków może to prowadzić do zarzucenia procesu odbierającego nadmiar informacji [8]. Jako rozwiązanie tego problemu proponujemy, aby wyjątki, które nadejdą w trakcie obsługi, były ustawiane w kolejkę i oczekiwały na zwolnienie zasobów obliczeniowych. Ponadto procesy wysyłające zdalne wyjątki zostają zablokowane na instrukcji wysyłania do czasu obsłużenia zdalnego wyjątku przez odbiorcę.

Dotychczas w wielu środowiskach programowania (RMI) używano pojęcia zdalnego wyjątku. Wyjątki skierowane zdefiniowane powyżej różnią się od zdalnych wyjątków tym, że jawnie podajemy proces odbierający wyjątek, który nie musi być procesem wywołujący zdalną metodę. Wówczas proces sygnalizujący wyjątek będzie oczekiwał na wyniki takiej dodatkowej obsługi.

4. Realizacja modelu w bibliotekach programowania rozproszonego

4.1. Wyjątki skierowane w MPI

Model przetwarzania w bibliotece MPI dobrze pasuje do wykorzystania wyjątków skierowanych. Obecnie MPI udostępnia możliwość lokalnego wywołania funkcji obsługi, jednak nie umożliwia żadnego przesyłania informacji o błędach między procesami.

Użycie wyjątków skierowanych w MPI wymaga wprowadzenia do biblioteki dwóch mechanizmów:

- funkcji obsługi dla wyjątków skierowanych,
- wysyłania wyjątków skierowanych.

Proponowany przez nas projekt przyjmuje następujące nazwy funkcji:

MPI_Remote_errhandler_create (function, errhandler),

MPI_Remote_errhandler_set (communicator, errhandler),

MPI_Directed_exception_throw (process, errnumber).

Parametrami funkcji są odpowiednio: funkcja obsługi, uchwyt błędu, komunikator środowiska MPI, numer procesu przeznaczenia oraz numer błędu. Zauważmy, że przedstawiony model nie jest związany z żadnym sposobem lokalnej obsługi wyjątków. Jeżeli implementacja jest wykonywana w języku C++, możemy wykorzystać standardowe mechanizmy try-catch-throw. Natomiast, jeżeli wykonujemy implementację w języku C, gdzie

nie mamy typowej obsługi wyjątków, możemy wykorzystać lokalne funkcje obsługi lub kody błędów zwracane przez funkcje.

4.2. Wyjątki skierowane w RMI

Biblioteka RMI/Java wykorzystuje typowy mechanizm obsługi wyjątków try-catch-throw. Wyjątki mogą być wysłane z wielu różnych punktów systemu: w kliencie, w serwerze, w czasie komunikacji z serwerem. Jednak żaden z mechanizmów nie umożliwia wysłania wyjątku do obiektu innego niż klient lub serwer. Rozszerzony model wprowadza pojęcie obiektu przeznaczenia tak, aby klient mógł wysłać wyjątek do innego klienta lub serwer do innego serwera.

throw (Exception e, Destination d);
void theObject::handleException(Exception e);

Wyjątek jest obiektem, który zawiera informacje na temat błędu oraz kontekst wywołania. Gdy wyjątek dotrze do odbiorcy, ten wywołuje odpowiednią funkcję obsługi. Zdefiniowanie funkcji obsługi jest konieczne, gdyż skierowany wyjątek jest zdarzeniem asynchronicznym i musi zostać asynchronicznie obsłużony. Proces wysyłający musi ponadto wiedzieć, jaki proces może ten wyjątek odebrać, tę informację może zdobyć z wykorzystaniem usługi NamingService dla środowiska RMI.

4.3. Porównanie modeli wyjątków

Przedstawiony model wprowadza nowy sposób obsługi i reakcji na wyjątki w systemach rozproszonych. W tabeli 2 przedstawiono podstawowe różnice między zwykłymi i skierowanymi wyjątkami.

Tabela 2

Porównanie między zwykłym i rozszerzonym modelem wyjątków

Właściwość	Model try-catch-throw	Model rozszerzony	
		kontekst lokalny	kontekst zdalny
zdalne wysyłanie	brak	istnieje	---
parametry wywołania	wartość (typ)	wartość (typ), przeznaczenie	wartość (typ)
model obsługi	terminacja	terminacja	terminacja / wznowienie
sposób implementacji	try-catch-throw	try-catch-throw	try-catch-throw / funkcja obsługi
zdarzenie synchroniczne	tak	tak	nie

Praktyczne zastosowanie modelu wiąże się ze wzrostem złożoności aplikacji. Wynika to z faktu, że w tradycyjnym modelu bloki chronione i funkcje obsługi były wyraźnie zdefiniowane. W modelu rozszerzonym konieczne jest podwójne definiowanie funkcji obsługi - dla wyjątków lokalnych oraz dla wyjątków zdalnych. Innym problemem jest pobranie informacji na temat potencjalnych odbiorców wyjątków. Większość środowisk programowania udostępnia taką wiedzę, jednak nie wszystkie. W projekcie zaproponowanym dla MPI i RMI nie ma problemu ze zidentyfikowaniem innych procesów w środowisku wykonania, gdyż taka wiedza może zostać zgromadzona przy użyciu dostępnych mechanizmów.

Środowisko rozproszone jest dużo bardziej skomplikowane niż środowisko sekwencyjne. Użycie wyjątków skierowanych daje dodatkową możliwość komunikacji między procesami w przypadku zaistnienia błędów i wydaje się, że zastosowanie modelu w praktyce przyniesie korzyści w postaci zwiększenia niezawodności aplikacji.

5. Dedykowane procesy odbiorcze zdalnych wyjątków

Jednym z celów obsługi wyjątków jest rozdzielenie zwykłego i wyjątkowego wykonania programu. Zauważmy, że w przypadku wyjątków skierowanych konieczne jest definiowanie funkcji obsługi w każdym procesie, który może taki wyjątek odebrać. Stanowi to pewną niedogodność, gdyż mechanizmy naprawcze mogą być rozrzucone po wszystkich obiektach w systemie. Aby temu zapobiec należy wydzielić dedykowany obiekt (proces) odpowiedzialny za obsługę wyjątków. Obiekt ten będziemy określać mianem *nadzorcy* wyjątków.

Obiekt nadzorcy jest obiektem nadrzędnym w stosunku do aplikacji i pełni funkcję koordynatora zapewniającego zwiększoną niezawodność systemu. Funkcjonalność udostępniana przez nadzorcę może być podzielona na następujące czynności:

- odbiór i obsługa skierowanych wyjątków od innych procesów – wyjątki, które nie mogą zostać obsłużone przez procesy są kierowane do nadzorcy,
- przechowywanie informacji o stanie sytemu – nadzorca musi znać stan procesów oraz historię występujących błędów,
- udostępnianie danych o stanie aplikacji – na podstawie swoich informacji o stanie aplikacji nadzorca udostępnia innym procesom dane, które mogą pomóc w zwiększeniu niezawodności,
- wymuszenie odpowiednich akcji – nadzorca jest funkcjonalnie procesem sterującym i może wymuszać działania, które mają na celu zwiększenie niezawodności; w szczególności nadzorca wymusza zakończenie procesu, jeżeli uzna, że stan procesu jest niewłaściwy.

Przedstawiona funkcjonalność nadzorcy jest bazową funkcjonalnością, która może być rozszerzona o dodatkowe funkcje specyficzne dla aplikacji. Oczywiste jest, że mechanizmy naprawcze i sposoby reakcji na błędy zależą od rodzaju błędu i architektury aplikacji, jednak nadzorca udostępnia wzorzec, dzięki któremu możliwe jest proste zaimplementowanie wymaganej funkcjonalności.

Aby skutecznie zarządzać aplikacją, nadzorca przechowuje stan aplikacji jako swoje wewnętrzne dane (określane jako Baza Wyjątków). Baza Wyjątków stanowi strukturę, w której nadzorca przechowuje historię występujących wyjątków i dane na ich temat. Dane w Bazie Wyjątków zawierają:

- aktualny stan procesu,
- informacje o wyjątkach zgłoszonych przez proces,
- informacje o wyjątkach zgłoszonych na temat procesu przez inne procesy.

Zastosowanie mechanizmu nadzorcy pozwala na rozdzielenie zwykłego i wyjątkowego przepływu sterowania na poziomie całej aplikacji, co powoduje, że kod jest bardziej przejrzysty i łatwiejszy do zarządzania. Ponadto Baza Wyjątków daje projektantom aplikacji dodatkowo możliwość śledzenia i naprawy ewentualnych wyjątków, które mogą wystąpić podczas działania systemu.

6. Podsumowanie

W pracy usystematyzowano koncepcje obsługi wyjątków w systemach rozproszonych. Wzrost liczby aplikacji internetowych oraz ich wykorzystanie w coraz to bardziej odpowiedzialnych dziedzinach życia zwiększa znaczenie wiarygodności ich funkcjonowania. W wielu przypadkach nie wystarczy lokalna obsługa błędów pojawiających się sytuacjach wyjątkowych i konieczne staje się bardziej złożone podejście, sprowadzające się do zdalnej obsługi wyjątków. W pracy zasugerowano taką koncepcję wprowadzając pojęcie wyjątków skierowanych. Przeprowadzono również pierwsze badania symulacyjne, które potwierdzają pewne korzyści takiego podejścia, a mianowicie wzrost wiarygodności przy zachowaniu takiego samego poziomu wydajności wykonywanej aplikacji.

Praca została wykonana w ramach grantu KBN numer 4 T11C 005 25.

LITERATURA

1. Bach M. J.: The Design of the UNIX Operating System. Prentice-Hall Inc., 1986.
2. Bloomer J.: Power Programming with RPC. O'Reilly & Associates, Inc., 1992.

3. Buhr P. A., Mok W. Y. R.: Advanced Exception Handling Mechanisms. IEEE Trans. Software Eng, 2000.
4. Java Remote Method Invocation. Sun Microsystems, Inc. 2002.
5. MPI-2: Extensions to the Message Passing Interface. MPI Forum, 1997.
6. Shannon B.: J2EE Platform Specification. Sun Microsystems Inc., 2002.
7. Snir M.: MPI: The Complete Reference. The MIT Press, 1996.
8. Xu J., Romanovsky A., Tandell B.: Concurrent Exception Handling and Resolution in Distributed Object Systems. IEEE Trans. on PDS, 2000.

Recenzent: Dr inż. Wojciech Mikanik

Wpłynęło do Redakcji 29 maja 2003 r.

Abstract

Exception handling in distributed systems is a tedious task and requires a professional design and implementations. The paper presents exception handling models for various distributed computing environments [Fig. 1] together with the analysis of their applicability in practice. The main focus is the idea of assigning a destination to a thrown exception and allowing sending an exception to any remote process. The remote process specifies a handling function for remote exceptions as they arrive as asynchronous events [Fig. 2, 3]. We noticed that a dedicated process (called exception manager) could be specified for receiving and handling of remote exceptions. An implementation project of the idea is presented for MPI and RMI library.

Adresy

Paweł L. KACZMAREK: Politechnika Gdańska, Wydział Elektroniki, Telekomunikacji i Informatyki, ul. G. Narutowicza 11/12, 80-952 Gdańsk, Polska, pkacz@eti.pg.gda.pl .
Henryk KRAWCZYK: Politechnika Gdańska, Wydział Elektroniki, Telekomunikacji i Informatyki, ul. G. Narutowicza 11/12, 80-952 Gdańsk, Polska, hkrawk@eti.pg.gda.pl .