СОФИЙСКИ УНИВЕРСИТЕТ "Св. КЛИМЕНТ ОХРИДСКИ"
ФАКУЛТЕТ ПО МАТЕМАТИКА И ИНФОРМАТИКА
КАТЕДРА Изчислителни системи

ЗАПИСКИ

по

МРЕЖОВО ПРОГРАМИРАНЕ

Доц. д-р Стела Русева

2022

Тема 1. Увод в мрежовото програмиране

Мрежовото програмиране е обширна област с голям избор на различни технологии за желаещите да установят комуникация между няколко машини.

Днес TCP/IP протоколите са най-перспективната технология за построяване на мрежи. Това е обусловено от развитието на Интернет и от самото разпространение на Световната паяжина (World Wide Web). Уеб не е приложение и не е протокол, макар тя да използва и приложения (уеб браузъри и сървъри) и протоколи (например HTTP). Уеб е най-популярното сред потребителите на Интернет използване на мрежовите технологии.

TCP/IP осигурява за приложния слой:

- Хетерогенна среда, която осигурява взаимодействието на различни ОС;
- Скриване на хетерогенността от потребителя и приложенията;
- Простота за разширяемост и мащабиране;
- Взаимодействие на основата обмен на съобщения.

Клиент-сървърна архитектура

Съгласно парадигмата на клиент-сървърната архитектура един или няколко клиенти и един или няколко сървъра съвместно с базова операционна система и среда за взаимодействие образуват единна система, осигуряваща разпределени изчисления, анализ и представяне на данните.

Приложение на модела клент-сървър

Процесът на разработване на разпределени приложения е сложен и една от най-важните му задачи е решението как функционалността на приложенията да се разпредели между клиентската и сървърната част.

Алгоритъм на работа на клиент-сървърната архитектура

В класическия случай дадената схема функционира по следния начин:

- клиентът формира и изпраща заявка към сървъра;

- сървърът реализира необходимата обработка на данните, формира резултат и го предава на клиента;

- клиентът получава резултата, изобразява го на устройство за извеждане и чака следващите действия на потребителя.

Цикълът се повторя, докато потребителят не приключи работата със сървъра.

Макар че конкретизираме клиенти и сървъри не винаги е очевидно каква роля играе конкретната програма и не може еднозначно да се твърди, че едната програма обслужва другата. Но в случая с използването на TCP/IP разликите стават по-ясни.

Сървърът слуша зададен порт, за да открие входящи TCP съединения или UDP дейтаграми от един или няколко клиента. Можем да определим, че клиентът е този, който започва пръв диалога.

Ще разгледаме три типични случая за архитектура клиент-сървър. В първия случай клиентът и сървърът работят на една машина. Това е най-простата конфигурация, понеже отсъства физическа мрежа. Изпращаните данни се предават на TCP/IP стека, но не се слагат в изходящата опашка на мрежовото устройство, а се зациклят от системата и после се връщат обратно в стека, но вече в качеството си на приети данни.

На етапа на разработването такова местоположение на клиента и сървъра дава определени предимства, даже ако в реалността те ще работят на различни машини. Първо, по-лесно е да се оцени производителността на двете програми, понеже са изключени мрежови забавяния във времето. Второ, този метод създава идеална лабораторна среда, в която пакетите не се губят, не се забавят и винаги пристигат в правилния ред. Между другото, почти винаги може да се създаде такова натоварване, че UDP дейтаграмите да се губят. И накрая, разработването и дебъгването винаги е по-лесно и по-удобно на една машина.

Като следващ пример разглеждаме конфигурацията клиент и сървър, работещи на различни машини, но в една локална мрежа. Това също е реална мрежа и условията пак са близки до идеалните. Пакетите рядко се губят и практически винаги пристигат в правилния ред. Такава ситуация често се среща в практиката. При което някои приложения са предназначени да работят само в такава среда. Типичен пример е сървър за печат. В неголяма локална мрежа може да има такъв единствен сървър, обслужващ няколко машини. Едната от тези машини (или мрежовото програмно осигуряване на базата на TCP/IP е вградено в принтера) изпълнява ролята на сървър, който приема заявките за печат от клиентите на другите машини и ги слага в опашка за принтера.

В третия пример клиентът и сървърът работят на различни компютри, свързани в глобалната мрежа. Такава мрежа може да е Интернет, но най-важното е, че приложенията вече не се намират в една локална мрежа, понеже по маршрута на IP дейтаграмите има поне един маршрутизатор.

Такова обкръжение може да бъде „враждебно”, сравнено с първите два случая. Поради нарастването на трафика в глобалната мрежа започват да се препълват опашките, в които маршрутизаторът временно съхранява постъпилите пакети, докато не ги изпрати към получателя им. А когато в опашката вече няма място, маршрутизаторът отхвърля пакетите. Като резултат клиентът е длъжен да предава пакетите повторно (и се получава забавяне), което води до поява на дубликати и доставка на пакети в неправилен ред. Тези проблеми възникват прекалено често.

Обмен на данните по мрежата

Въпреки, че мрежата осигурява преместване на данните от една точка до друга, самата тя остава пасивна. Това означава, че мрежата не генерира и не анализира предаваните данни. Тя даже не съдържа някакви средства за обработка на информацията. Всички операции за обработка на данните се изпълняват от приложните програми.

При използване на мрежата приложенията работят в двойка (съвместно) и всяка двойка приложения използва мрежата просто за обмен на съобщения. Например ще разгледаме управлението на разпределена база от данни, която позволява на отдалечени потребители да се обслужват от централна база от данни. За да работи системата са необходими две приложения: едното трябва да работи на този компютър, на който се намира базата от данни, а другото - на отдалечен компютър. Приложението, изпълнявано на отдалечения компютър, изпраща заявка на приложението, работещо на компютъра на базата от данни. След пристигането на заявката при приложението, което се изпълнява на компютъра на базата от данни, се получава необходимата информация от базата от данни и се връща отговора. Само на тези две приложения е известен формата на съобщенията и неговия смисъл.

Обработка на данните в системите клиент-сървър

По какъв начин две програми могат взаимно да се открият в такава голяма мрежа, като Интернет? Както и в повечето други мрежи, в Интернет се използва прост

механизъм: първо се пуска едно приложение и се чака докато към него не се обърне друго приложение. На второто приложение следва да са му известни къде го очаква първото приложение.

Споразумението, в съответствие с което едно мрежово приложение очаква докато към него не се обърне друго приложение, лежи в основата на системите от типа клиент-сървър (или архитектурите за разпределени изчисления на принципа клиент-сървър). Ще разгледаме общия принцип и основната терминология на технологията клиент-сервър.

Програмата, която очаква заявки за установяване на съединение, се нарича сървър, а програмата, която се явява инициатор на тези заявки, се нарича клиент. За инициализация на заявката за установяване на съединение клиентската програма е необходимо да знае къде точно се изпълнява сървърната програма и да посочи това местоположение на мрежовото програмно осигуряване.

По какъв начин клиентът посочва местоположението на сървъра? В Интернет това местоположение се обозначава с двойка идентификатори (computer, application).

Тук computer обозначава машината, на която се изпълнява сървърната програма, а application - конкретната приложна програма на този компютър. При предаването по Интернет тези две стойности са представени във вид на двоични числа. Самите потребители обикновено не ползват двоичното представяне - те посочват необходимите стойности в символен вид. Потребителите въвеждат имена и числа, а мрежовото програмно осигуряване извиква за изпълнение функции, преобразуващи автоматично символните стойности в съответстващи двоични стойности.

Принципи за обмен на данни

В процеса на обмен на данни между повечето приложения в Интернет се изпълнява една и съща последователност от операции:

– Първо се зарежда за изпълнение сървърното приложение и се очаква заявка за установяване на съединение от клиента;

– Клиентът се обръща към сървъра, като посочва неговото местоположение и предава изискване да пристъпи към обмен на данни;

– Клиентът и сървърът си обменят съобщения;

– След приключването на предаването на данни клиентът и сървърът си съобщават за това, че е достигнат края (на файла), за да се прекрати обмена на данните (пакетите).

В протоколите са посочени общите операции, които следва да бъдат предвидени и на разработчиците на всяка операционна система е предоставена възможността да определят API интерфейса, използван в приложенията за изпълнението на тези операции. Затова стандартът на протокола може единствено да посочва какви операции са необходими за предаването на данните между приложенията, а API интерфейсът задава точните имена на всяка функция и типа на всеки параметър.

Този модел първоначално предполага неравноправност на взаимодействащите процеси и най-често се използва за организация на мрежови приложения. Основните разлики за процесите на клиента и сървъра, характерни за отдалеченото взаимодействие са:

- Сървърът, като правило работи постоянно, а клиентите могат да работят епизодично;
- Сървърът очаква заявки от клиентите, инициатор за взаимодействието е клиентът;

- Като правило клиентът се обръща към един сървър еднократно, като в същото време към сървъра могат едновременно да постъпват заявки от няколко клиента;
- Клиентът е длъжен да знае пълния адрес на сървъра преди началото на общуването, в същото време сървърът може да получи информация за пълния адрес на клиента от пристигналата вече заявка след началото на общуването;
- Клиентът и сървърът са длъжни да използват един и същ протокол на транспортния слой.

Неравноправността на процесите в модела клиент-сървър налага своя отпечатък на програмния интерфейс, използван между нивата на приложенията (процесите) и транспортния слой.

Макар че стандартите на протоколите позволяват на разработчиците на операционната система да избират всякакъв API интерфейс, в много операционни системи е приет API интерфейса на сокетите.

Интерфейс на приложното програмиране

Терминът *интерфейс на приложното програмиране* (API - Application Program Interface) се използва за описание на набор от операции на разположение на програмиста. API интерфейсът определя параметрите и смисъла на всяка операция.

Прост API интерфейс за операциите за обмен на данни по мрежата

API интерфейсът включва седем операции (функции), които са достатъчни за създаването на повечето мрежови приложения.

Операция	Описание
await_contact	Използва се от сървъра за преход в режим на очакване на заявка от клиента за установяване на съединение
make_contact	Използва се от клиента за предаване на сървъра на заявка за установяване на съединение
cname_to_comp	Използва се за преобразуване на името на компютъра в еквивалентна двоична стойност
appname_to_appnum	Използва се за преобразуване на името на програмата в еквивалентна двоична стойност
send	Използва се от клиента или сървъра за изпращане на данни
recv	Използва се от клиента или сървъра за получаване на данни
send_eof	Използва се от клиента или сървъра след приключването на предаването на данните

Сървърът започва работа с извикване на функцията await_contact за преход в режим на очакване на заявки от клиента. Клиентът извиква функцията make_contact за установяване на съединение. След установяването на съединението двете програми могат да обменят съобщения с помощта на функциите send и recv. Двете приложения трябва така да са програмирани, че всяко от тях да може да определи дали трябва да изпраща или да получава съобщения, ако и двете се опитват само да получават, ще останат блокирани завинаги. След приключване на предаването на данните едното приложение извиква функцията send_eof за изпращане на другото на признак за край на файла. В другото приложение функцията recv връща стойност 0, като указание за това, че е достигнат края на файла. След като двете страни извикат функцията send_eof, съединението се затваря.

Тема 2. UDP сокети

Интерфейсът на сокетите е API, който определя набор от функции, позволяващ разработване на приложения за използването им в TCP/IP мрежите. Първоначално сокетите са били вградени в операционната система UNIX.

Абстракция на сокетите

Сокетът може да се разглежда като краен пункт за предаване на данни по мрежата. Мрежовото съединение е процес на предаване на данни по мрежата между два компютъра или процеса. Когато програмите използват сокет, за тях той представлява абстракция, представляваща единия от краищата на мрежовото съединение. За установяването на съединение в абстрактния модел на сокетите е необходимо всяка от мрежовите програми да има свой собствен сокет. Връзката между два сокета може както да бъде ориентирана на съединение, така и да не бъде. Въпреки че разработчиците са модифицирали системния код на UNIX, интерфейсът на сокетите както по-рано използва концепцията за вход-изход на данните в UNIX. Мрежовият модел на интерфейса на сокетите използва цикъла отваряне-четене-запис-затваряне.

Когато на програмата й е нужен сокет, тя формира неговите характеристики и се обръща към API, изисквайки от мрежовото програмно осигуряване неговия дескриптор. Структурата на таблицата с описанието на параметрите на сокета съвсем незначително се различава от структурата на таблицата с описанието на параметрите на файла.

Дескриптор на сокет и дескриптор на файл

Ако дескрипторът на файла сочи към определен файл (вече съществуващ или току-що създаден) или устройство, то дескрипторът на сокета не съдържа някакви определени адреси или целеви пунктове на мрежовото съединение. Този факт, че дескрипторът на сокета не представлява определена мрежова точка (endpoint), съществено го отличава от всеки друг дескриптор на стандартната система за вход-изход. В повечето операционни системи дескрипторът на файла сочи към определен файл, намиращ се на твърдия диск. Програмите, работещи със сокети, първо създават сокет и чак след това го съединяват с целевата точка на другия край на мрежовото съединение.

Дескриптор на сокет

На практика “създаването на сокет” това е просто процес на заделяне на памет от системата за поставяне в нея на структура от данни, описваща дадения сокет. В UNIX всеки процес ползва таблицата с дескриптори на файловете. Функцията socket в UNIX получава дескриптор от таблицата с дескриптори на файловете. Дескрипторът представлява указател към вътрешна структура от данни. Съединението между двете мрежови програми носи в себе си следната информация:

-местен (локален) порт на протокола, обозначаващ програмата или процеса, изпращащ/приемащ съобщения или дейтаграми;

-адрес на локалния компютър, обозначаващ компютъра в мрежата, приемащ/изпращащ пакетите с данни;

-адрес на външен компютър, обозначаващ компютъра в мрежата, приемащ/изпращащ пакетите с данни;

-отдалечен порт на протокола, обозначаващ програмата или процеса-получател на данните;

-протокол, обозначаващ, по какъв начин програмата ще предава данните по мрежата.

Структурата от данни на сокета, съответстваща на дескриптора на сокета, съдържа информация за тези пет пункта. По какъв начин сокетът се явява реализация на абстрактния модел за крайната точка на мрежовото съединение? Структурата от данни на сокета съдържа всички елементи, необходими на крайната точка на мрежовото съединение. Структурата данни на сокета значително опростява процеса на мрежово съединение. Когато една програма желае да установи връзка с друга програма, програмата-предавател просто отдава своята информация на сокета, а интерфейсът на сокетите на свой ред я предава по-нататък в стека на мрежовите протоколи на TCP/IP. Преди това програмата е длъжна да създаде сокет, извиквайки системната функция socket(), а след това да го конфигурира.

По-лесният вариант за взаимодействие на отдалечени процеси представлява схемата за организация на общуване между клиент и сървър с помощта на дейтаграми, т.е. използването на UDP протокол.

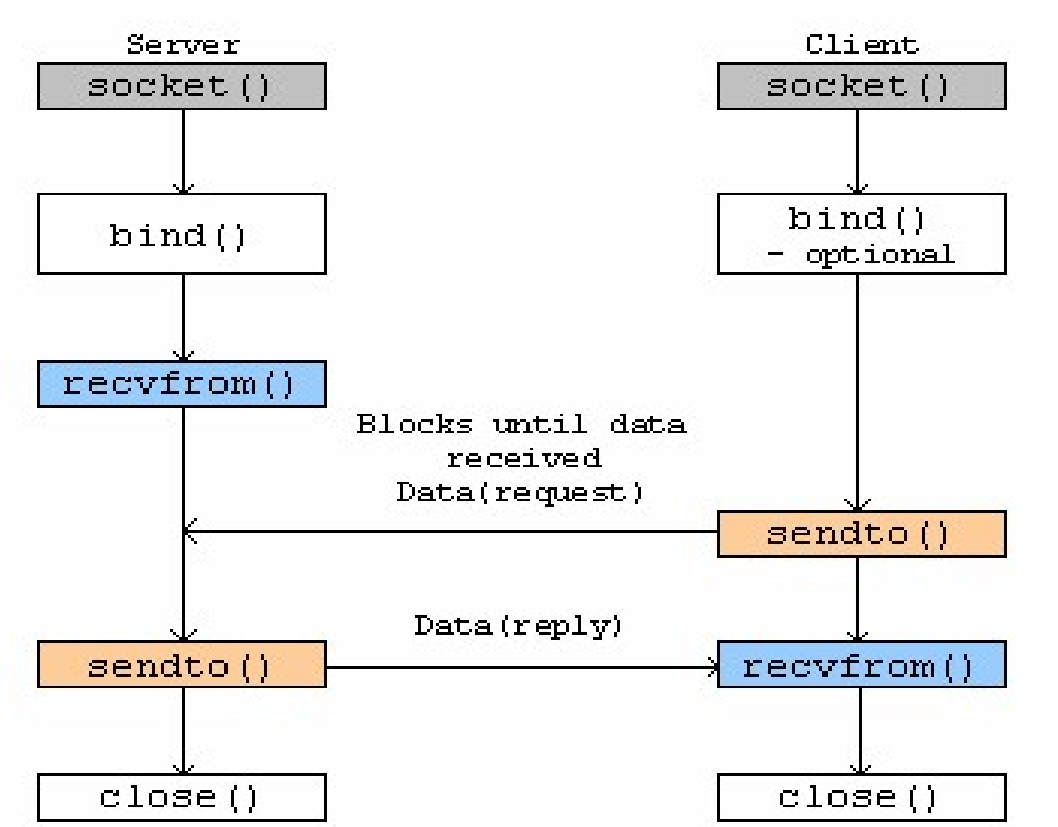


Схема на взаимодействието на клиент и сървър по UDP протокол

Създаването на сокет се реализира с помощта на системния примитив socket(). За свързването на създадения сокет към IP адрес и номер на порт (настройката на адреса) служи системния примитив bind(). На очакването за получаване на съобщение, неговото прочитане при необходимост, определянето на адреса на изпращача съответства системния примитив recvfrom(). За изпращането на дейтаграмата отговаря системният примитив sendto().

back-word == little-endian == host order .. A7 B2 C3 FF 07 CE F2 21 ..
fore-word == big-endian == network order .. FF C3 B2 A7 21 F2 CE 07 .. Motorola

В i80x86 е приета подредба на байтовете, при която младшите байтове на цяло число имат младши адреси. При мрежовата подредба на байтовете, приета в Интернет,

младши адреси имат старшите байтове на числото.

Функции за преобразуване на реда на байтовете

```
#include <netinet/in.h>
unsigned long int htonl(
   unsigned long int hostlong);
unsigned short int htons(
   unsigned short int hostshort);
unsigned long int ntohl(
   unsigned long int netlong);
unsigned short int ntohs(
   unsigned short int netshort);
```

Параметърът при тях е стойността, която трябва да се конвертира. Връщаната стойност е това, което се получава като резултат от конвертирането. Посоката на конвертирането се определя от подредбата на буквите h (host) и n (network) в наименованието на функцията, размерът на числото - от последната буква в наименованието, т.е. htons - това е host to network short, ntohl - network to host long.

Функции за преобразуване на IP адреси inet_ntoa(), inet_aton()

```
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
int inet_aton(const char *strptr,
   struct in_addr *addrptr);
char *inet_ntoa(struct in_addr *addrptr);
```

Тези функции осъществяват превод на IP адреси от символно представяне (във вид на четворка числа, разделени с точки) в числово представяне и обратно. Функцията inet_aton() преобразува символен IP адрес от числово представяне в мрежова подредба на байтовете. Функцията връща 1, ако в символен вид е записан правилен IP адрес и 0 в противен случай. Структурата от типа struct in_addr се използва за съхраняване на IP адресите в мрежова подредба на байтовете и изглежда така:

```
struct in_addr {
   in_addr_t s_addr;
};
```

Използва се адреса на такава структура, а не просто адрес на променлива от тип in_addr_t. Това, че се използва структура, състояща от една променлива, а не самата 32-битова променлива, се е получило исторически.

За обратното преобразуване се прилага функцията inet_ntoa(). Числовото представяне на адреса в мрежова подредба на байтовете е длъжно да бъде записано в структурата от тип struct in_addr, адресът на която addrptr предава на функцията като аргумент. Функцията връща указател към стринг, съдържащ символно представяне на адреса. Този стринг се поставя в статичен буфер, като при следващи извиквания новото съдържимо заменя старото.

Функция bzero()

```
#include <string.h>
void bzero(void *addr, int n);
```

Функция bzero запълва първите n байта, започвайки от адреса addr с нулеви стойности. Функцията нищо не връща.

Създаване на сокет. Системен примитив socket()

```
#include <sys/types.h>
#include <sys/socket.h>
int socket(int domain, int type,
        int protocol);
```

За създаване на сокет в операционната система служи системното извикване socket(). За транспортните протоколи на семейството TCP/IP съществуват два вида сокети: UDP сокет - сокет за работа с дейтаграми, и TCP сокет - потоков сокет. При създаването на сокет е необходимо точно да се специфицира неговия тип. Тази спецификация се реализира с помощта на три параметъра на примитива socket().

За транспортните протоколи на TCP/IP винаги в качеството на първи параметър се посочва предопределената константа AF_INET (Address family - Internet) или нейният синоним PF_INET (Protocol family - Internet).

Вторият параметър приема предопределени стойности: SOCK_STREAM за потоков сокет и SOCK_DGRAM - за дейтаграмните.

Понеже третият параметър в нашия случай не се отчита, в него ще слагаме стойността 0.

В случай на успешно приключване системният примитив връща файлов дескриптор (целочислена стойност по-голяма или равна на 0), който ще се използва като указател към създадения комуникационен възел при всички следващи мрежови извиквания. При възникване на някаква грешка се връща -1.

Адреси на сокетите. Настройка на адреса на сокета. Системен примитив bind()

```
#include <sys/types.h>
#include <sys/socket.h>
int bind(int sockd,
      struct sockaddr *my_addr,
      int addrlen);
```

Когато сокетът вече е създаден, тогава е необходимо да се настрои неговият адрес. За целта се използва системният примитив bind(). Първият параметър на примитива е длъжен да съдържа дескриптора на сокета, за който се прави тази настройка на адреса. Вторият и третият параметър задават този адрес.

Във втория параметър има указател към структурата struct sockaddr, съдържаща локалната част на пълния адрес.

Указателите от типа struct sockaddr * се срещат при много мрежови системни примитиви. Те се използват за предаване на информация за това към кой адрес е свързан или трябва да се свърже сокета. Структурата struct sockaddr е описана във файла <sys/socket.h> по начина:

```
struct sockaddr {
   short sa_family;
   char sa_data[14];
};
```

Такъв състав на структурата е обусловен от това, че мрежовите системни примитиви могат да се използват за различните семейства протоколи, които по различен начин определят адресните пространства за отдалечени и локални адреси на сокета. На практика този тип данни представлява само общ шаблон за предаване на системни примитиви на структури от данни, специфични за всяко семейство протоколи. Общ елемент за тези структури е само полето short sa_family (което в различните структури може да има различни имена, като важното е всички те да са от един тип и да са първите елементи от своите структури) за описанието на семейството

протоколи. Съдържимото на това поле на системния примитив се анализира за точното определяне на състава на постъпилата информация.

За работа с TCP/IP семейството протоколи се използва адрес на сокета от следния вид, описан във файла <netinet/in.h>:

```
struct sockaddr_in{
   short sin_family;
   /* Избраното семейство протоколи - винаги PF_INET */
   unsigned short sin_port;
   /* 16 битов номер на порт в мрежова подредба на байтовете */
   struct in_addr sin_addr;
   /* Адрес на мрежовия интерфейс */
   char sin_zero[8];
   /*Това поле не се използва. Трябва винаги да е запълнено с нули */
};
```

Първият елемент на структурата sin_family задава семейството протоколи. В него се записва предопределената константа PF_INET.

Отдалечената част на пълния адрес IP адрес се съдържа в структурата от типа struct in_addr.

За посочване на номер на порт е предназначен елемента на структурата sin_port, в който номерът на порта трябва да се съхранява в мрежова подредба на байтовете. Съществуват два варианта за задаване на номер на порт: фиксиран порт по желание на потребителя и порт, който произволно е зададен от операционната система. Първият вариант изисква посочване като номер на порт на положително предварително известно число и за UDP протокол обикновено се използва при настройването на адресите на сокетите и при предаването на съобщения с помощта на системния примитив sendto(). Вторият вариант изисква посочване в качеството на номер на порт стойността 0. В този случай операционната система сама свързва сокета към свободен номер на порт. Този механизъм обикновено се използва при настройването на сокетите в програмите на клиентите, когато за програмиста не е задължително предварително точно да се знае номера на порта.

Номерата на портовете от 1 до 1023 могат да се задават на сокетите само от процеси, работещи с привилегии на системен администратор. Като правило, тези номера са закрепени за системни мрежови услуги независимо от вида на използваната операционна система, за да могат потребителските клиентски програми да искат обслужване винаги от един и същ локален адрес. Съществуват редица масово използвани мрежови програми, които стартират процеси с пълномощия на обикновени потребители. За тези програми от корпорацията Internet по присвоено име и номер (ICANN) се заделя диапазон от адреси от 1024 до 49151, който не е желателно да се използва поради възможни конфликти. Номерата на портовете от 49152 до 65535 са предназначени за процеси на обикновени потребители.

IP адресът при настройката също така може да бъде определен по два начина. Той може да бъде на конкретен мрежов интерфейс, принуждаващ операционната система да приема/предава пакети само през този мрежов интерфейс, а може да бъде зададен за цялата изчислителна система (съобщенията ще може да се получават/изпращат през всеки един мрежов интерфейс). В първия случай в качеството на стойност на полето на структурата sin_addr.s_addr се използва числова стойност на IP адреса на конкретния мрежов интерфейс в мрежова подредба на байтовете. Във втория случай тази стойност трябва да е равна на стойността на константата INADDR_ANY, приведена към мрежова подредба на байтовете.

Третият параметър на системния примитив bind() трябва да съдържа

фактическата дължина на структурата, адресът на която се предава като втори параметър. Тази дължина се променя в зависимост от семейството протоколи и даже е различна в рамките на едно семейство протоколи. Размерът на структурата, съдържащ адреса на сокета за TCP/IP семейството протоколи може да бъде определен като sizeof(struct sockaddr_in). Системният примитив връща стойност 0 при нормално приключване и -1 при регистрирана грешка.

Системни примитиви sendto() и recvfrom()

```
#include <sys/types.h>
#include <sys/socket.h>
int sendto(int sockd, char *buff,
   int nbytes, int flags,
   struct sockaddr *to, int addrlen);
int recvfrom(int sockd, char *buff,
   int nbytes, int flags,
   struct sockaddr *from, int addrlen);
```

Системният примитив sendto е предназначен за изпращане на дейтаграми. Системният примитив recvfrom е предназначен за четене на пристигналите дейтаграми и за определяне адреса на изпращача. По премълчаване при отсътствие на пристигнали дейтаграми примитивът recvfrom се блокира докато се появи дейтаграма. Примитивът sendto може да се блокира при липса на място за дейтаграмата в мрежовия буфер.

Параметърът sockd е дескриптора на създадения по-рано сокет, т.е. стойност, връщана от системния примитив socket(), чрез който ще се изпраща или ще се получава съобщения.

Параметърът buff представлява адрес на област от паметта, започвайки от който ще се взема информация за предаване или разполагане на приетата информация.

Параметърът nbytes за системния примитив sendto определя количеството байтове, което трябва да бъде предадено, започвайки от адреса в паметта buff. Параметърът nbytes за системния примитив recvfrom определя максималния брой на байтовете, който може да бъде записан в приемния буфер, започвайки от адреса buff.

Параметърът to за системния примитив sendto определя указател към структура, съдържаща адреса на сокета на получателя на пакета, която трябва да бъде попълнена преди извикването. Ако параметърът from за системния примитив recvfrom не е равен на NULL, тогава за случая на установяване на връзка чрез пакети данни той определя указателя към структурата, в която ще бъде записан адреса на сокета на изпращача на информацията след приключване на примитива. В този случай преди извикването е необходимо тази структура да се занули.

Параметърът addrlen за системния примитив sendto трябва да съдържа фактическата дължина на структурата, адреса на която се предава като параметър to. За системния примитив recvfrom параметрът addrlen представлява указател към променлива, в която ще бъде записана фактическата дължина на структурата на адреса на сокета на изпращача, ако това е определено от параметъра from. Да отбележим, че преди извикването този параметър е длъжен да сочи към променливата, съдържаща максимално допустимата стойност на тази дължина. Ако параметърът from има стойност NULL, тогава и параметърът addrlen може да има стойност NULL.

Параметърът flags определя режимите на използване на системните примитиви. В нашия курс няма да разглеждаме неговото използване и затова задаваме за стойност на този параметър 0.

В случай на успешно приключване системният примитив връща броя на реално изпратените или приети байтове. При възникване на някаква грешка се връща -1.

Функции за работа с IPv6 адреси и DNS

Стандартът POSIX 1003.1-2001 препоръчва вместо функцията inet_aton() да се използва функцията inet_pton, понеже тя поддържа както IPv4, така и IPv6:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
int inet_pton(int af, const char *src, void *dst);
```

af: AF_INET (dst: struct in_addr*), AF_INET6 (dst: struct in6_addr*)

Функцията inet_pton() преобразува IP адрес, задаван с аргумента src от стрингов формат в мрежова подредба на байтовете в съответствие със зададеното семейство адреси af. Функцията връща положителна стойност при успешно приключване, 0 - ако стрингът не може да бъде интерпретиран като IP адрес и отрицателна стойност, ако af съдържа неподдържана стойност за семейството протоколи.

Стандартът POSIX 1003.1-2001 препоръчва вместо функцията inet_ntoa() да се използва функцията inet_ntop, понеже тя поддържа както IPv4, така и IPv6:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
const char *inet_ntop(
        int af, const void *src, char *dst, size_t cnt);
```

af: AF_INET (src: struct in_addr*), AF_INET6 (src: struct in6_addr*)

Функцията inet_ntop() преобразува IP адрес, задаван с аргумента src от мрежова подредба на байтовете в стрингов формат. Полученият ASCIIZ стринг се записва в зададения от потребителя буфер dst. Потребителят е длъжен да резервира поне INET_ADDRSTRLEN (или INET6_ADDRSTRLEN) байта за буфера dst. Функцията връща стойността на указателя dst или NULL - при грешка.

За получаване на информация за възлите в мрежата

```
#include <netdb.h>
#include <sys/socket.h>
struct hostent *gethostbyname(const char *cp);
struct hostent *gethostbyaddr(const char *addr, int len, int type);
```

Функцията търси в DNS информация за посочения хост. За gethostbyname() хостът се задава с име на домейн или с IP адрес в стрингов формат. За gethostbyaddr() хостът се задава с IP адрес в мрежова подредба, вторият аргумент определя дължината на адреса, а третият е AF_INET или AF_INET6. При грешка връща NULL. Функциите могат да търсят информация в /etc/hosts, в DNS, в LDAP, в NIS и др.

Формат на структурата hostent

```
struct hostent {
        char *h_name;         /*официално име на хоста*/
        char **h_aliases;     /*масив с псевдоними*/
        int h_addrtype;       /*тип на адреса - AF_INET*/
        int h_length;         /*дължина на адреса в байтове*/
        char **h_addr_list;   /*масив с адреси*/
};
```

Масивите h_aliases и h_addr_list са ограничени от елемент със стойност 0. Адресите се съхраняват в мрежова подредба на байтовете. Функциите gethostbyname() и gethostbyaddr() връщат указател към структура, поместваща в областта с данни на ядрото на ОС. Ако за gethostbyname() като параметър се зададе IP адресът на хоста, тогава той ще се копира в полето h_name (няма да се изпълни DNS заявка). Ако трябва да се получи името на домейна според IP адреса, може да се ползва gethostbyaddr().

Функции за работа с БД за информация за мрежовите услуги и протоколи

```
#include <netdb.h>
struct servent *getservbyname(
        const char *name, const char *proto);
struct servent *getservbyport(
        int port, const char *proto);
```

Функцията getservbyname() търси в БД (/etc/services) информация (номер на порт) за зададената мрежова услуга. Функцията getservbyport() връща информация за мрежовата услуга според номера на нейния порт. Параметърът proto може да бъде NULL, тогава търсенето става за всеки протокол. При грешка връща NULL.

```
struct servent {
        char *s_name;          /*официално име на услугата*/
        char **s_aliases;      /*масив с псевдонимите*/
        int s_port;            /*номер на порта*/
        char *s_proto;         /*протокол*/
};
```

Масивът s_aliases е ограничен от елемент със стойност 0. Номерът на порта s_port се съхранява в мрежова подредба на байтовете. Функциите getservbyname() и getservbyport() връщат указател към структура, разположена в областта с данни на ядрото на ОС.

Функции за работа с БД за информация за протоколите

```
#include <netdb.h>
struct protoent *getprotobyname(const char *name);
struct protoent *getprotobynumber(int proto);
struct protoent {
        char *p_name;          /*официално име*/
        char **p_aliases;      /*псевдоними*/
        int p_proto;           /*номер на протокола*/
};
```

Четири типа данни, отнасящи се до мрежите

Тип на данните	Файл	Структура	Функции за търсене по ключ
възли	/etc/hosts	hostent	gethostbyaddr, gethostbyname
мрежи	/etc/networks	netent	getnetbyaddr, getnetbyname
протоколи	/etc/protocols	protoent	getprotobyname, getprotobynumber
услуги	/etc/services	servent	getservbyname, getservbyport

За всеки от четирите типа данни съществува собствена структура (изисква се включване на <netdb.h>). За всеки от четирите типа са определени три функции: getXXXent (чете следващия запис във файла и при необходимост затваря файла), setXXXent (отваря файла, ако все още не е отворен и преминава в началото на файла) и endXXXent (затваря файла). За всеки от четирите типа данни има фукции за търсене по ключ (keyed lookup). Те последователно преминават през файла като извикват getXXXent и търсят елемент, съвпадащ с аргумента. Търсенето по ключ има вида: getXXXbyYYY (като gethostbyname и gethostbyaddr).

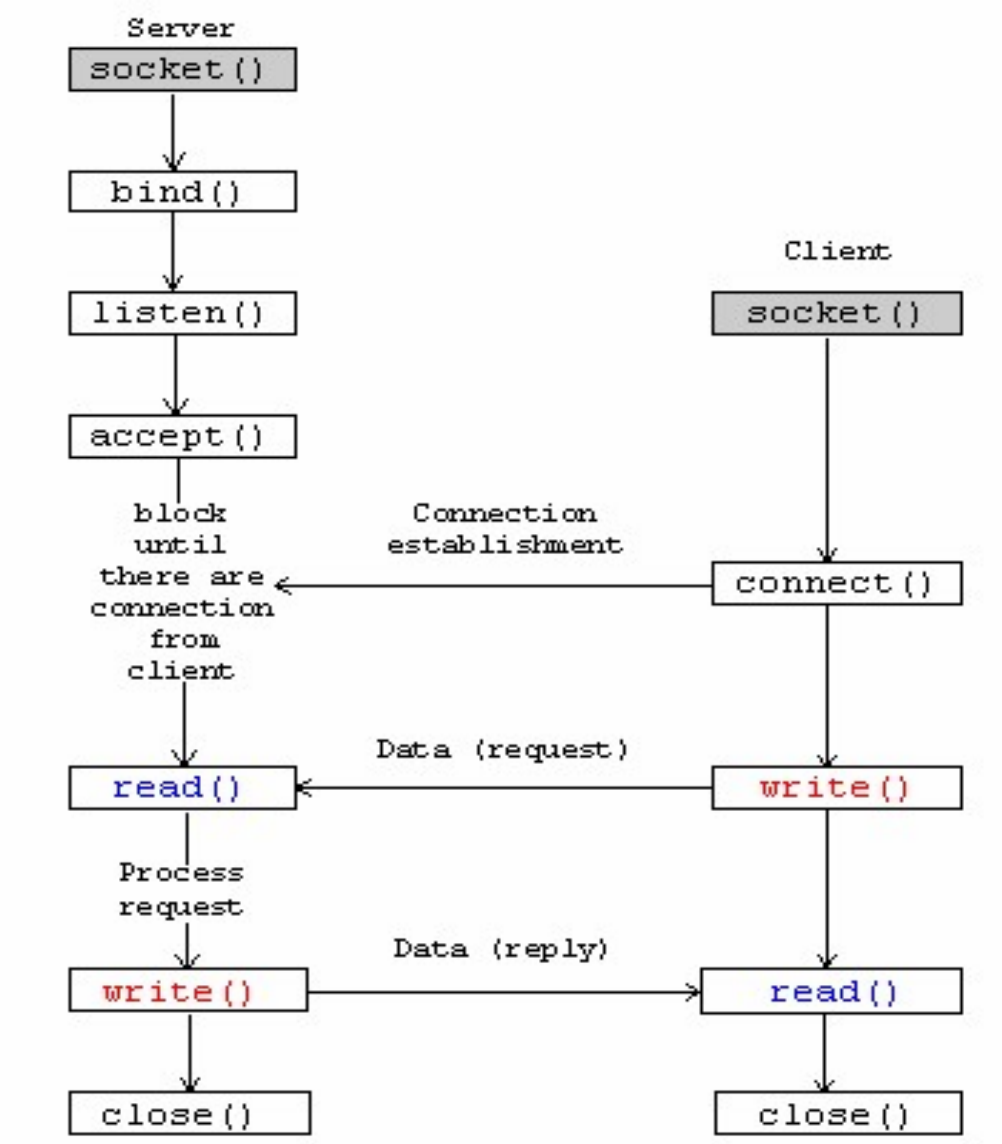


Схема на взаимодействието на клиента и сървъра по TCP протокол

Организация на връзката между процеси чрез установяване на логическо съединение

За създаването на сокетите се използва системния примитив socket(). След което наборите от системните извиквания вече са различни. За свързването на сървъра към зададен IP адрес и номер на порт се използва системния примитив bind(). За клиентския процес това свързване е обединено с процеса на установяване на съединение със сървъра в нов системен примитив connect() и е скрито от потребителя. Вътре в това извикване операционната система осъществява настройка на сокета на избрания от нея порт и на адреса, на който да е мрежовия интерфейс. За въвеждане на сокета на сървъра в пасивно състояние и за създаване на опашка от съединения служи системния примитив listen(). Сървърът очаква съединения и получава информация за адреса на присъединилия се към него клиент с помощта на стемния примитив accept(). Понеже установеното логическо съединение изглежда от гледна точка на процесите като канал за връзка, позволяващ предаването на данни в двете посоки, с помощта на потоковия модел, за изпращането и четенето на информацията двата системни примитива

използват вече известните системни примитиви read() и write(), а за затваряне на съединението - close(). При работа със сокети извикванията read() и write() притежават същите особености на поведението, както и при работа с програмни канали (pip'ове и FIFO).

Установяване на логическо съединение на клиенската страна

Системен примитив connect()

Извикването connect() скрива вътре в себе си настройката на сокета на избран системен порт и произволен мрежов интерфейс (на практика това е примитивът bind() с нулев номер на порт и IP адрес INADDR_ANY). Примитивът се блокира докато не бъде установено логическо съединение или не изтече определен интервал от време, който може да се регулира от системния администратор.

```
#include <sys/types.h>
#include <sys/socket.h>
int connect(int sockd,
   struct sockaddr *servaddr,
   int addrlen);
```

Системният притив connect служи за организация на връзката на клиента със сървъра. Най-често той се използва за установяване на логическо съединение, макар че може да бъде използван и при свързване с дейтаграми (connectionless).

Параметърът sockd представлява дескриптора на създадения по-рано комуникационен възел, т. е. стойността, която е върнал системния примитив socket().

Параметърът servaddr представлява адрес на структура, съдържаща информация за пълния адрес на сокета на сървъра. Той има тип указател към структура-шаблон struct sockaddr, която е длъжна да бъде конкретизирана в зависимост от използваното семейство протоколи и попълнена преди извикването.

Параметърът addrlen е длъжен да съдържа фактическата дължина на структурата, адреса на която се предава в качеството на втори параметър.

При установяване на виртуалното съединение системният примитив не приключва до неговото установяване или до изтичането на установеното в системата време - timeout. При използването му в connectionless връзка примитивът приключва незабавно.

Системният примитив връща стойност 0 при нормално приключване и -1 при възникнала грешка. В случай, че в качеството на IP адрес е посочен несъществуващ адрес или адрес на изключена машина, тогава програмата ще съобщи за грешка при работата на примитива connect().

Установяване на логическо съединение на сървърната страна (подготовка)

Създаване на опашка от заявки за съединения

Създаване на пасивен сокет

Системен примитив listen()

```
#include <sys/types.h>
#include <sys/socket.h>
int listen(int sockd, int backlog);
```

Системният примитив listen се използва от сървъра за превод на сокета в пасивен режим и установяване на дълбочината на опашките за съединения.

Параметърът sockd представлява дескриптор на създадения преди това сокет, който е длъжен да бъде приведен в пасивен режим, т. е. стойността, която е върнал системния примитив socket().

Параметърът backlog определя максималния размер на опашките за сокетите,

намиращи се в състояние на установени и полуотворени съединения.

Системният примитив връща стойност 0 при нормално приключване и -1 при възникване на грешка.

Последният параметър на различните UNIX-подобни операционни системи и даже за различните версии на една и съща система може да има различен смисъл. Някъде това е сумарната дължина на двете опашки, в други случаи се отнася за опашката на полуотворените съединения (например за Linux до версия на ядрото 2.2), за трети - за опашката на установените съединения (например за Linux от версия на ядрото от 2.2 нагоре), а някъде изобщо се игнорира.

Установяване на логическо съединение на сървърната страна

Създаване на активен сокет (нов сокет)

Приемане на напълно установеното съединение

Системен примитив accept()

Системният примитив accept() позволява на сървъра да получи информация за напълно установените съединения. Ако опашката на напълно установените съединения не е празна, тогава той връща дескриптора на първия присъединен сокет в тази опашка, като едновременно го изтрива от опашката. Ако опашката е празна, тогава примитивът изчаква появяването на напълно установено съединение. Системното извикване също така позволява на сървъра да научи адреса на клиента, установил съединението.

```
#include <sys/types.h>
#include <sys/socket.h>
int accept(int sockd,
   struct sockaddr *cliaddr,
   int *clilen);
```

Параметърът sockd съдържа дескриптора на създадения и настроен сокет, предварително приведен в пасивен (слушащ) режим с помощта на системния примитив listen().

Параметърът cliaddr служи за получаването на адреса на клиента, установил логическото съединение и трябва да съдържа указател към структурата, в която ще бъде записан този адрес.

Параметърът clilen съдържа указател към цяла променлива, която след връщането от извикването ще съдържа фактическата дължина на адреса на клиента. Преди извикването тази променлива трябва да съдържа максимално допустимата стойност на тази дължина. Ако параметърът cliaddr има стойност NULL, тогава и параметърът clilen може да има стойност NULL.

Системният примитив връща при нормално приключване дескриптора на присъединения сокет, създаден при установяване на съединението, за последващото общуване на клиента и сървъра, и стойност -1 при възникване на грешка.

Създаване на програми с паралелна обработка на заявките на клиентите

Постъпващите заявки сървърът може да обработва последователно - заявка след заявка или паралелно. За обработката на всяка от тях се стартира нов процес или нишка. Като правило, сървърите, ориентирани за комуникация клиент-сървър с помощта на установяване на логическо съединение (протокол TCP), обработват заявките паралелно, а сървърите, ориентирани на комуникация клиент-сървър без установяване на логическо съединение (протокол UDP), обработват заявките последователно.

В приведения пример сървърът осъществява последователна обработка на заявките от различни клиенти. При такъв подход клиентите могат дълго да престояват

след установяването на съединението, очаквайки обслужване. Затова обикновено се прилага схема за псевдопаралелна обработка на заявките. След приемане на установеното съединение сървърът поражда процес-дете, на който предава работата с клиента. Процесът-родител затваря присъединения сокет и преминава в очакване на ново съединение.

Конструкцията на сървъра за паралелна обработка предполага, че за всяка нова заявка от клиента се създава ново копие на сървърния процес. Ако сървърът за паралелна обработка е многонишков, тогава неговата производителност значително се увеличива. Вместо създаването на нов процес или техните превключвания, многонишковият сървър за паралелна обработка просто образува нов поток, който се изпълнява значително по-бързо. Ако хостът е оборудван с няколко процесора, тогава сървърът ще работи още по-бързо, понеже всеки поток ще може да се изпълнява на отделен процесор.

Схема на работа на TCP сървър с паралелна обработка на заявките

Ще разгледаме какво се случва, когато се появява поредна заявка, а сървърът още не е приключил обслужването на предишната. С други думи, когато клиентската програма се опитва да изпрати още една заявка, като в същото време последователният сървър още не е приключил с обработката на предишната, или когато клиентът изпраща преди момента, когато паралелният сървър е приключил със създаването на нов процес, обработващ предишната заявка.

В този случай сървърът може да отхвърли или да игнорира постъпилата заявка. С цел обработката да бъде по-ефективна съществува примитивът listen. Той действа колкото е възможно по-бързо и разполага всички постъпили зявки в опашката. Примитивът listen не само привежда сокета в пасивен режим на очакване, но и го подготвя за обработка на множество едновременно постъпващи заявки. С други думи, в системата се организира опашка на постъпилите заявки, и всички заявки, очакващи обработка от сървъра, се записват в нея, докато освободилият се сървър не избере тази заявка. Ако опашката при постъпването на нова заявка се окаже препълнена, сокетът отхвърля съединението и клиентската програма ще получи съобщение за грешка.

След като се е изпълнил примитивът accept, паралелният сървър ще създаде нов (породен) процес и ще му предаде задачата за обслужването на новата заявка. По какъв начин се създава процес-дете зависи от операционната система. В UNIX например за целта се извиква примитива fork. Независимо от това, как се създава породен процес, процесът-родител му предава копие на новия сокет. След това процесът-родител затваря собственото копие на сокета и отново извиква примитива accept.

Функцията fork поражда обслужващ процес за обмен на данни с клиента. Наличието на два процеса позволява бързото приемане на заявки от клиентите и паралелното им изпълнение.

Принципи за проектиране на паралелен сървър

За мрежовото съединение на сокета е необходимо да има две крайни точки. Крайните точки на мрежовото съединение определят адресите на взаимодействащите процеси. До тогава докато мрежовите процеси са съединени с различни точки на мрежовото съединение, те могат да работят на един и същ порт на протокол, без да си пречат. Сървърът за паралелна обработка създава нов процес за всяко съединение. С други думи, на хоста постоянно работи единствен главен сървър, очаквайки заявка от някакъв процес в мрежата. В друг момент от време, на същия този хост както преди работи един главен сървър, а заедно с него и множество подчинени. Всеки подчинен сървър работи с уникален адрес на конкретния, съединен с него процес.

Сокетът на главния сървърен процес (прослушващ заявките от всеки мрежов адрес) поражда породен сървърен процес, който е със своя опашка и обработва заявката. Сокетът, прослушващ всеки мрежов адрес, не може да има отворено съединение. За да се установи мрежовото съединение е необходимо да има двойка крайни точки, всяка от които да притежава зададени адреси. Понеже сокетът на главния сървърен процес се занимава само със слушане на порта и приемане на заявките за съединение (винаги прослушва заявките от всеки адрес, той е в състояние да обработва само току-що постъпилата заявка). За целта веднага след accept сървърът извиква fork за създаване на процес-дете. После се анализира стойността, която е върнал този примитив. В родителския процес върнатата стойност съдържа идентификатора на породения процес, а в породения тази стойност е равна на нула. Използвайки този признак, се преминава към поредното извикване на accept в родителския процес, а породеният процес обслужва клиента и приключва.

Синхронно мултиплексиране (на въвеждането и извеждането)

Приложението може да отваря няколко сокета и да взаимодейства чрез тях с други приложения. За обслужването на всеки сокет може да се поражда отделен процес, но в този случай в системата ще има много процеси и производителността на компютъра рязко ще се намали. Постоянният анализ на състоянието на всеки сокет също така ще доведе до забавяне на работата на приложенията. Затова в такива приложения е целесъобразно да се използва функцията select, която позволява да се задържи изпълнението на приложението дотогава, докато не се случи някакво събитие със сокета. Тази функция осигурява синхронна схема за мултиплексиране. Тя често се използва от сървърите и се определя като nonblocking sockets. Тази функция позволява да се проследява едновременно състоянието на множество (мулти) дескриптори и след това да се осигури възможност на програмата да продължи да работи с един от избраните дискриптори.

Функцията има следното описание:

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
int select(int n, fd_set *readfds, fd_set *writefds,
        fd_set *exceptfds, struct timeval *timeout);
FD_CLR(int fd, fd_set *set);
FD_ISSET(int fd, fd_set *set);
FD_SET(int fd, fd_set *set);
FD_ZERO(fd_set *fds);
```

При грешка функцията връща -1. При успешно приключване – броя на дескрипторите, готови за операции без блокиране. Функцията връща 0, ако е изтекло времето за очакване (*timeout*).

Ако поне един сокет е готов за изпълнение на зададената операция, тогава select връща ненулева стойност, а всички дескриптори, довели до сработването на функцията се записват в съответните множества. Това позволява анализирането на съдържащите се в множествата дескриптори и изпълнение с тях на необходимите действия.

Функцията select работи с три множества дескриптори, всяко от които е от тип fd_set. В множеството readfds се записват дескрипторите на сокетите, от които се налага да се четат данните (в това множество се добавят и слушащите сокети). Множеството writefds трябва да съдържа дескрипторите на сокетите, в които ще се

записва, а exceptfds - дескрипторите на сокетите, които се налага да се контролират за възникване на грешки. Ако няма такова множество, тогава може да се подаде вместо указател към него NULL.

Параметърът timeout показва какво максимално време трябва да очаква процеса за постъпване на информация за сокетите. Той е от тип struct timeval, определен в sys/times.h и има следния формат:

```
struct timeval {
   int tv_sec;     /* секунди */
   int tv_usec;   /* микросекунди */
};
```

На практика не е възможно да се достигне такава точност на измерването на времето при използването на select. Реалната точност е около 100 милисекунди.

За работа с множествата от дескриптори са предвидени функциите FD_XXX. Тяхното използване напълно скрива детайлите на вътрешното устройство на fd_set.

FD_ZERO(fd_set *set) - изчиства множеството set;

FD_SET(int fd, fd_set *set) - добавя дескриптора fd в множеството set;

FD_CLR(int fd, fd_set *set) - изтрива дескриптора fd от множеството set;

FD_ISSET(int fd, fd_set *set) – проверява дали се съдържа в множеството set дескриптора fd.

Интерфейсът на сокетите използва битови маски за да определи множеството от сокети, за което да бъде установено наблюдение.

В n е необходимо да се запише максималната стойност за броя на дескрипторите за всички множества плюс единица. Този параметър определя броя на дескрипторите на сокетите, които ще се проверяват. Обикновено като стойност на n се конкретизира константата FD_SETSIZE, задаваща, че е необходимо да се анализират всички отворени сокети.

Номерата на проследяваните сокети се установяват в променливи от типа fd_set, които представляват масив от 256 бита и имат следното описание:

```
typedef struct fd_set {
        unsigned long fds_bits [8];
} fd_set;
```

Записаната единица в съответстващия разряд на променливата от типа fd_set, задава номерата на дескрипторите на сокетите, подлежащи на анализ.

В аргумента fd се задава номера на дескриптора на сокета, а в аргумента set следва да стои адреса на променлива от тип fd_set, където се съхраняват номерата на анализираните сокети.

Ако информацията пристигне преди изтичането на таймаут-а, тогава select веднага ще върне управлението на процеса, като ще зададе с двоични маски, от кои сокети е постъпила информация. Например, ако потребителят е пожелал да проверява за получаване на данни от сокети с номера 0, 1 и 2, тогава в параметъра readfds той трябва да подаде двоична маска 7. Когато select върне управлението, двоичната маска, ще бъде заменена с маска, показваща номерата на сокетите, в които са постъпили данни.

Ако параметърът timeout има стойност 0, тогава процесът ще приключи когато пристигне от някой сокет сигнал за промяна на състоянието му.

Стойността на функцията select фиксира броя на сокетите, от които е постъпила информация, а в променливите readfds, writefds, exeptfds ще се съдържат номерата на дескрипторите на тези сокети.

Тема 4. Windows Sockets API (WSA)

Класическата реализация на сокетите се нарича Berkeley sockets или BSD sockets (Berkeley Sockets API, 1983 г.). Впоследствие почти всички ОС по един или друг начин са наследили тази реализация. За всяка ОС степента на поддръжката на сокетите е различна. Стопроцентова преносимост имат SOCK_STREAM за TCP протокол и SOCK_DGRAM за UDP протокол.

Някои основни разлики между socket API и Winsock API:

- В MSWindows наборът от заглавни файлове е драстично намален. Може да се включи само един файл winsock.h или winsock2.h в случай, че ще се използват разширените възможности на Winsock 2.
- В MSWindows е необходимо библиотеката Winsock явно да се инициализира преди да се използват нейните функции. Това се прави с функцията WSAStartup. Също така следва да се извика функцията WSACleanup при приключването на работата със сокетите. Приключването на процеса с функцията ExitProcess автоматично не освобождава ресурсите на сокетите.
- В GNU Linux дескрипторите на сокетите имат тип int. В MSWindows сокетите не са файлови дескриптори, затова за тях е въведен тип SOCKET (обявен като u_int).
- В MSWindows за работа със сокети не се използват функции на файловия вход-изход (read и write). В MSWindows всички извиквания write и read трябва да се заменят с send и recv съответно. Вместо close се използва closesocket.
- В MSWindows глобалната променлива errno не се използва. Вместо това кодът на последната грешка се запазва от системата за всеки поток отделно. За да бъде получен този код се използва функцията WSAGetLastError.
- В MSWindows са въведени допълнителни константи, които следва да се използват вместо конкретни числа. Така стойностите, връщани от функциите на Winsock, следва да се сравняват с константите INVALID_SOCKET или SOCKET_ERROR, а не с -1.

За съжаление разликите при socket API и MSWinsock не се ограничиват само от приведения списък. Например могат да възникнат проблеми с функциите, нямащи пряко отношение към socket API. В MSWindows няма аналог на fork и за организация на паралелно обслужване на клиентите се налага използването на други средства.

Преди да се използва кода на сокетите се налага да добавят съответните библиотеки и header-и. Кодът за инициализация и създаване на сокетите в Unix подобните системи и MSWindows изглеждат различно, понеже разработчиците на MSWindows са изнесли кода на мрежовата подсистема в отделна библиотека, затова първо програмистът трябва да направи допълнителна инициализация.

Създаване на сокет:

```
// Listing 1 (Linux & FreeBSD)
// int socket (int domain, int type, int protocol);
#include <sys/types.h>
#include <sys/socket.h>
  int sd;           // нашият дескриптор
  // тук няма инициализация, понеже socket() е системно извикване
        sd = socket (PF_INET, SOCK_DGRAM, 0);
// Listing 1 (MSWindows)
// SOCKET socket (int pf, int type, int protocol);
```

```
#pragma comment (lib, "ws2_32.lib");    // търсим нужната библиотека
#include <winsock2.h>        // winsock2.h: typedef u_int SOCKET
  WORD wVersion;          // версията на winsock интерфейса
  WSADATA wsaData;        // тук ще записваме данните за сокета
  wVersion = MAKEWORD (2, 0);      // задаваме версията на winsock
  SOCKET sd;          // нашият дескриптор на сокета
  int wsaInitError = WSAStartup (wVersion, &wsaData);    // инициализираме winsock
  if (wsaInitError != 0)
    // казваме, че са възникнали проблеми и излизаме
    exit (1);
  else
    // щом инициализацията е минала успешно, тогава създаваме сокет
          sd = socket (PF_INET, SOCK_DGRAM, 0);
```

Кодът следва да се компилира като конзолно приложение.

По стандарта за сокетите на системите Unix кодът на грешките за тези примитиви се записва в глобална библиотечна променлива errno. Разработчиците на winsock2 не са съгласни с подобна политика, затова в MSWindows системите за получаването на кода на последната фиксирана грешка следва да се използва int WSAGetLastError(void). От обявяването на функцията socket() се вижда, че в случая с MSWindows системите тя може да връща само неотрицателни числа, докато в класическия BSD socket() се връщат такива от диапазона на int. За сигнализация за грешка winsock socket() връща 0, а BSD socket() връща -1. Налага се внимателнно да се обработват грешките. Реализацията winsock изисква на всяко извикване на WSAStartup да съответства извикване на WSACleanup(). Налага се да се почисти това, което е оставила след себе си библиотеката и след сокета:

Затваряне на сокет и деинициализация на библиотеките:

```
// Linux & FreeBSD
       int close (int sd);
// Windows
  int closesocket (SOCKET sd);
         int WSACleanup (void);
```

Реализацията на winsock изисква да почистим цялата използвана памет с нейните методи. За целта извикваме две функции, първата (closesocket()) унищожава нашия дескриптор (closesocket() незабавно приключва всички операции, стоящи в опашката за изпълнение от сокета), втората почиства цялата памет, която е била заделена за нормалното функциониране на winsock. За различна от MSWindows система това е просто затваряне на сокета с класическия метод, използвайки системното извикване close().

Винаги трябва да се проверяват стойностите, връщани от функциите, работещи в мрежата. Мрежовият код е най-нестабилната част на приложението, вследствие на архитектурата на мрежовите протоколи и Интернет.

Да видим как изглежда името на сокета:

```
struct sockaddr_in
{
  unsigned char  sin_len;  // само за FreeBSD
  unsigned char  sin_family;
```

```
  unsigned short   sin_port;
  struct in_addr  sin_addr;
  char    sin_zero[8];  // отсъства в Linux
};
 struct in_addr
{
  unsigned long int  s_addr;      // следва да е с дължина 4 байта (int32)
        };
```

В общия случай трябва да създадем структура sockaddr_in, да я занулим и да запълним в нея три полета: sin_family, sin_port и sin_addr. Ето как се прави това:

```
// Listing 2 (Linux & FreeBSD)
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#define PORT 5000      //  5000
struct sockaddr_in addr;
memset (&addr, 0, sizeof (struct sockaddr_in)); // зануляването е наложително за
                                                // максимална преносимост
addr.sin_family = AF_INET;
addr.sin_port = htons (PORT);
       addr.sin_addr.s_addr = inet_addr ("192.168.0.1");  // за този пример

// Listing 2 (MSWindows)
#include <string.h>
#include <winsock2.h>
#define PORT 5000
struct sockaddr_in addr;
memset (&addr, 0, sizeof (struct sockaddr_in));
addr.sin_family = AF_INET;
addr.sin_port = htons (PORT);
addr.sin_addr.s_addr = inet_addr ("192.168.0.1");
```

Настройка на сокета:

```
// Linux & FreeBSD
 int bind (int s, const struct sockaddr *addr, int addrlen);
// Windows
 int bind (SOCKET s, const struct sockaddr *name, int namelen);
```

Изпращане на съобщения с UDP протокол:

```
// Linux & FreeBSD
int sendto (int s, const void *msg, int len, int flags, const struct sockaddr *to, int tolen);
// Windows
int sendto (SOCKET s, const char *buf, int len, int flags, const struct sockaddr *to, int tolen);
```

Методи за адресните структури:

```
// Listing 3 (MSWindows & Linux & FreeBSD)
void _sock_addr::set_port (unsigned short port)
 {
```

```
    address->sin_port = htons (port);
 }
 void _sock_addr::set_ip (const char *ip)
{
   address->sin_addr.s_addr = inet_addr (ip);
}
```

С тези методи можем да използваме една и съща адресна структура за изпращане на дейтаграми с различни адреси, като попълваме намера на порта и адреса на получателя. После изпращаме пакетите. Това ни позволява да избегнем допълнителни (ненужни) операции за заделяне/освобождаване на памет.

Определяне адреса на сокета:

```
// Listing 4 (Windows & Linux & FreeBSD)
struct sockaddr_in *name = new struct sockaddr_in;
int namelen = sizeof (struct sockaddr_in);
int error = getsockname (sd, (struct sockaddr *) name, &namelen);
if (error == -1) // SOCKET_ERROR в WINDOWS
{
   // обработка на грешките
}
cout << "The socket IP address is: " << inet_ntoa (name->sin_addr) << endl
       << "The socket port number is: " << ntohs (name->sin_port) << endl;
```

Получаване на съобщения с UDP протокол:

```
// Linux & FreeBSD
  int recvfrom (int s,  void *buf,  int len, int flags, struct sockaddr *from, int *fromlen);
// Windows
        int recvfrom (SOCKET s, char *buf, int len, int flags, struct sockaddr *from, int
*fromlen);
```

Функция listen():

```
// Linux & FreeBSD
  int listen (int s, int backlog);
// Windows
        int listen (SOCKET s, int backlog);
```

Изпращане на заявка към сървъра:

```
// Linux & FreeBSD
  int connect (int s, const struct sockaddr *server_addr, int namelen);
// Windows
        int connect (SOCKET s, const struct sockaddr *server_addr, int namelen);
```

Примерен код за изпращане на заявка:

```
  #define SERVER_ADDRESS    "192.168.0.1"
  #define SERVER_PORT    10000
  struct sockaddr_in addr;
  addr.sin_family = AF_INET;
  addr.sin_port = htons (SERVER_PORT);
  addr.sin_addr.s_addr = inet_addr (SERVER_ADDRESS);
  int namelen = sizeof (struct sockaddr_in);
  int error = connect (sd, (struct sockaddr *) &addr, namelen);
```

```
 if (error == -1) // SOCKET_ERROR в WINDOWS
   // неуспешно
 else
       // приемане-изпращане на данни
```

Допускаме, че в опашката е постъпила клиентска заявка (клиентът е извикал connect()). Ние трябва да я приемем и да я обслужим.

За целта извикваме функцията accept():

```
// Linux & FreeBSD
 int accept (int s, struct sockaddr *addr, int *addrlen);
// Windows
       SOCKET accept (int s, struct sockaddr *addr, int *addrlen);
```

Обявяването на send() изглежда по следния начин:

```
// Linux & FreeBSD
 int send (int s, const void *msg, int len, int flags);
// Windows
       int send (SOCKET s, const char *buf, int len, int flags);
```

Предаване на данни в сокет при Berkeley Sockets API:

write(sock,buf,len) <=> send(sock,buf,len,0) <=> sendto(sock,buf,len,0,NULL,0);

Функцията send() връща броя на предадените, а не на предаваните байтове. TCP протоколът гарантира успешна доставка на данните до получателя само в случай, че съединението няма да се прекъсне. Ако съединението се прекъсне преди края на предаването, данните ще останат недоставени, но кодът няма да уведоми за това. А грешка ще сработи само в случай, че съединението се е разпаднало преди извикването на функцията send(). Ако send() върне число, по-малко от len, тогава се налага отново да извикаме send(), като му предадем в качеството на втори параметър указател към останалата част от данните и по съответен начин да променим len.

Обявяването на recv() изглежда по начина:

```
// Linux & FreeBSD
 int recv (int s, void *buf, int len, int flags);
// Windows
       int recv (SOCKET s, char *buf, int len, int flags);
```

Получаване (приемане) на данни от сокет при Berkeley Sockets API:

read(sock,buf,len) <=> recv(sock,buf,len,0) <=> recvfrom(sock,buf,len,0,NULL,0);

Функцията recv() връща броя на приетите байтове или грешка. Ако recv() върне 0, това не означава, че са били приети 0 байта. Това означава, че е затворено съединението. Приложението трябва да гарантира, че са получени всички байтове.

Всички разгледани до тук операции със сокети са синхронни. Програмата, използваща такива сокети, трябва сама от време на време да проверява по един или друг начин дали са пристигнали данните, установена ли е връзката и т.н. Асинхронните сокети позволяват на програмата да получава уведомяване за събитията/сигналите/ съобщенията, случили се със сокета: постъпване на данните, освобождаване на място в буфера, затваряне и т.н. (event driven program парадигма). Такъв вариант на работа по-добре подхожда на събитийно-ориентираните програми, типични за MSWindows. Поддръжката на асинхронни сокети се базира на съобщения, които се обработват от прозоречно-ориентирани приложения. Програмистът посочва какво съобщение на кой прозоречен интерфейс да пристига при възникване на събитие/сигнал/съобщение за интересуващия го сокет (callback functions).

Тема 5. Използване на сокети в Java

Ще разгледаме пример за клиент-сървърно приложение на езика Java.

Приложението функционира по следния алгоритъм:

1. Клиентът прочита от стандартното устройство за въвеждане (клавиатурата) низ от символи и изпраща този низ през свой сокет.

2. Сървърът приема (получава) низа през свой сокет.

3. Сървърът преобразува всички символи на низа в горен регистър (в главни).

4. Сървърът изпраща модифицирания низ на клиента.

5. Клиентът получава низа и го отпечатва (изобразява) на стандартното устройство за извеждане (дисплея).

Първо ще разгледаме взаимодействието с използване на TCP протокол.

Текст на програмата TCPClient.java:

```
import java.io.*;
import java.net.*;
class TCPClient {
 public static void main(String argv[]) throws Exception
 {
   String sentence;
   String modifiedSentence;
   BufferedReader inFromUser = new BufferedReader(
            new InputStreamReader(System.in));
   Socket clientSocket = new Socket ("hostname", 6789);
   DataOutputStream outToServer =
         new DataOutputStream(clientSocket.getOutputStream());
   BufferedReader inFromServer = new BufferedReader(
         new InputStreamReader(clientSoket.getInputStream()));
          sentence = inFromUser.readLine();
          outToServer.writeBytes(sentence + '\n');
          modifiedSentence = inFromServer.readLine();
   System.out.println("FROM SERVER: " + modifiedSentence);
   clientSocket.close();
  }
}
```

Програмата TCPClient създава три потока от данни и един сокет. Сокетът е с име clientSocket. Потокът inFromUser е входящ поток от данни за програмата и е свързан със стандартното устройство за въвеждане. Всички символи, въвеждани от клавиатурата на потребителя, попадат в поток inFromUser. Другият входящ поток от данни на програмата, inFromServer, е свързан със сокета и той предава символите от сървърната страна на приложението. Изходящият поток от данни на програмата TCPClient има име outToServer и също така е свързан със сокета. Този поток предава символите към сървъра за обработка.

Първите два реда съдържат имената на двата Java пакета java.io и java.net:

import java.io.*; import java.net.*;

Пакетът java.io съдържа класовете на входящите и изходящите потоци от данни. Обикновено тези класове са BufferedReader и DataOutputStream, които са използвани в програмата. Пакетът java.net съдържа класовете, поддържащи работата с компютърната мрежа (Socket и ServerSocket). Обектът clientSocket е породен от класа Socket.

class TCPClient {

```
  public static void main(String argv[]) throws Exception
    {......}
}
String sentence;
String modifiedSentence;
```

Приведените два реда представляват обявяване на обекти от тип String. Обектът sentence е предназначен за съхраняване на низа, въвеждан от потребителя и предаван на сървъра. Обектът modifiedSentence съдържа низа, който е пристигнал (получен) от сървъра и извеждан на стандартното устройство за извеждане.

```
BufferedReader inFromUser =
    new BufferedReader(new InputStreamReader(System.in));
```

Така се създава потоков обект inFromUser от тип BufferedReader. Инициализацията на потока от данни става чрез обекта System.in, свързващ потока със стандартното устройство за въвеждане. След изпълнението на тази команда към TCP клиента започват да се предават символите, въвеждани от потребителя от клавиатурата.

```
Socket clientSocket = new Socket ("hostname", 6789);
```

В приведения ред става създаването на обект clientSocket от тип Socket. Освен това, се инициализира TCP съединение между клиента и сървъра. Думата hostname е необходимо да се замени с името на хоста, като се запазят кавичките. Преди началото на установяването на TCP съединението клиентът прави DNS запитване за IP адреса на хоста. Стойността 6789 е номер на порт (може да се избере друго число, като клиентската и сървърната страна са длъжни да използват един и същ номер на порт).

```
DataOutputStream outToServer =
    new DataOutputStream(clientSocket.getOutputStream());
BufferedReader inFromServer =
    new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
```

Така става създаването на потоковите обекти, свързани със сокета. Потокът outToServer осигурява извеждането на програмата през сокета, а потокът inFromServer е предназначен за приемане на данните от сървърната страна.

```
sentence = inFromUser.readLine();
```

Този ред записва последователността от символи, въвеждани от потребителя, в променливата sentence. Въвеждането приключва когато потребителят натисне клавиша Enter. За записването на символите в променливата се използва потоковия обект inFromUser.

```
outToServer.writeBytes(sentence + '\n');
```

Тук низът sentence, снабден със символ „възврат на каретката” (CR, carriage return), се разполага в изходящия поток outToServer. След което се предава през сокета по виртуалния канал, съединяващ клиента със сървъра.

```
modifiedSentence = inFromServer.readLine();
```

Отговорът на сървъра се приема във входящия поток inFromServer, откъдето приетият низ се копира в променливата modifiedSentence. Записването на символите в низа modifiedSentence продължава докато не се получи символ „възврат на каретката”.

```
System.out.println("FROM SERVER: " + modifiedSentence);
```

Тук се извежда на дисплея полученият от сървъра низ.

```
clientSocket.close();
```

Така сокетът се затваря, а следователно и TCP съединението.

Текст на програмата TCPServer.java:

```
import java.io.*;
import java.net.*;
```

```
class TCPServer {
  public static void main(String argv[]) throws Exception
   {
     String clientSentence;
     String capitalizedSentence;
     ServerSocket welcomeSocket = new ServerSocket (6789);
     while (true) {
       Socket connectionSocket = welcomeSocket.accept();
     BufferedReader inFromClient = new BufferedReader(
        new InputStreamReader(connectionSocket.getInputStream()));
     DataOutputStream outToClient =
        new DataOutputStream(connectionSocket.getOutputStream());
     clientSentence = inFromClient.readLine();
     capitalizedSentence = clientSentence.toUpperCase() + '\n';
     outToClient.writeBytes(capitalizedSentence);
    }
  }
}
```

ServerSocket welcomeSocket = new ServerSocket (6789);

Така се създава обектът welcomeSocket от тип ServerSocket. Обектът welcomeSocket представлява сокет, с помощта на който клиентът установява първоначален контакт със сървъра. За това сокетът използва порт с номер 6789, съвпадащ с номера на порта на сокета на клиента. TCP протоколът установява пряк виртуален канал между сокета clientSocket на клиентската страна и connectionSocket на сървърната страна, след което клиентът и сървърът могат свободно да осъществят обмен на информация.

После програмата създава няколко потокови обекта, аналогични на clientSocket.

capitalizedSentence = clientSentence.toUpperCase() + ’\n’;

Тази команда изпълнява получаване на низа, предаван на клиента, привежда низа в горен регистър с помощта на метода toUpperCase() и добавя в края символ „възврат на каретката”.

Да разгледаме взаимодействието с използването на UDP протокол.

Текст на програмата UDPClient.java:

```
import java.io.*;
import java.net.*;
class UDPClient {
  public static void main(String args[]) throws Exception
  {
    BufferedReader inFromUser =
         new BufferedReader(new InputStreamReader (System.in));
    DatagramSocket clientSocket = new DatagramSocket();
    InetAddress IPAddress = InetAddress.getByName("hostname");
    byte[] sendData = new byte[1024];
    byte[] receiveData = new byte[1024];
    String sentence = inFromUser.readLine();
    sendData = sentence.getBytes();
    DatagramPacket sendPacket =
         new DatagramPacket(sendData, sendData.length, IPAddress, 9876);
    clientSocket.send(sendPacket);
```

```
        DatagramPacket receivePacket =
              new DatagramPacket(receiveData, receiveData.length);
        clientSocket.receive(receivePacket);
        String modifiedSentence = new String(receivePacket.getData());
        System.out.println("FROM SERVER:" + modifiedSentence);
        clientSocket.close();
       }
}
```

Програмата UDPClient създава един поток от данни и един сокет. Сокетът е от тип DatagramSocket и е с име clientSocket. Потокът inFromUser е входящ за програмата и е свързан със стандартното устройство за въвеждане. Когато потребителят въвежда символи, те попадат в програмата през входящия поток от данни. При използването на UDP протокол нито един от потоците не е свързан със сокета. Изпращането на байтовете се осъществява с пакети с помощта на обект от тип DatagramSocket.

DatagramSocket clientSocket = new DatagramSocket();

Тази команда създава обект clientSocket от тип DatagramSocket, но за разлика от програмата TCPClient не инициира установяване на TCP съединение. По тази причина конструкторът DatagramSocket() не приема в качеството на аргументи името на хоста на сървъра и номера на порта. Така действието на приведения ред се свежда до създаване на сокет на клиента.

InetAddress IPAddress = InetAddress.getByName("hostname");

Този ред създава DNS заявка за IP адреса на хоста с името hostname (думата hostname е необходимо да се замени с реалното име на хоста). На практика работата с DNS заявката се осъществява с метода getByName(), който приема в качеството на аргумент името на хоста и връща получения IP адрес. IP адресът се записва в обекта IPAddress от тип InetAddress.

byte[] sendData = new byte[1024];

byte[] receiveData = new byte[1024];

Байтовите масиви sendData и receiveData са предназначени за съхраняване на предаваната и приеманата информация съответно.

sendData = sentence.getBytes();

Стрингът sentence с помощта на метода getBytes() се преобразува в масив от байтове, който се копира в променливата sendData.

DatagramPacket sendPacket =
  new DatagramPacket(sendData, sendData.length, IPAddress, 9876);

Така се създава пакет sendPacket, предназначен за изпращане към сървъра, използващ UDP сокет. В пакета влизат данните, съдържащи се в променливата sendData, размерът на тези данни, IP адреса на хоста получател и номера на порта на приложението (9876). Пакетът има тип DatagramPacket.

clientSocket.send(sendPacket);

Методът send() на обекта clientSocket изпраща пакета sendPacket (clientSocket представлява клиентският сокет на приложението). В случая за UDP протокол е необходимо да се създаде пакет, съдържащ адреса на сървъра. След изпращането на пакета клиентът очаква получаване на пакет от сървъра.

DatagramPacket receivePacket =
   new DatagramPacket(receiveData, receiveData.length);

Така клиентът създава променлива за съхраняване на пакета receivePacket от тип DatagramPacket.

clientSocket.receive(receivePacket);

Клиентът се намира в състояние на очакване докато не започне да приема пакети от сървъра. Приеманите символи се записват в променливата receivePacket.

String modifiedSentence = new String(receivePacket.getData());

В този ред се извличат данните от обекта receivePacket и се преобразува типа от масив от байтове в низ с последващо присвояване на променливата modifiedSentence.

System.out.println("FROM SERVER:" + modifiedSentence):

Този ред отпечатва на дисплея на клиента низа modifiedSentence.

Текст на програмата UDPServer.java:

```
import java.io.*;
import java.net.*;
class UDPServer {
     public static void main(String args[]) throws Exception
      {
     DatagramSocket serverSocket = new DatagramSocket(9876);
     byte[] receiveData = new byte[1024];
     byte[] sendData = new byte[1024];
    while(true)
       {
      DatagramPacket receivePacket =
            new DatagramPacket(receiveData, receiveData.length);
      serverSocket.receive(receivePacket);
      String sentence = new String(receivePacket.getData());
      InetAddress IPAddress = receivePacket.getAddress();
      int port = receivePacket.getPort():
      String capitalizedSentence = sentence.toUpperCase();
      sendData = capitalizedSentence.getBytes();
      DatagramPacket sendPacket =
            new DatagramPacket(sendData, sendData.length, IPAddress, port);
      serverSocket.send(sendPacket);
        }
      }
}
```

Програмата UDPServer създава единствен сокет с име serverSocket от тип DatagramSocket. Същият тип е използван в програмата клиент на приложението. Аналогично със сокета не е свързан нито един от потоците с данни. Програмата UDPServer няма потоци - сокетът приема и предава пакети на процеса.

DatagramSocket serverSocket = new DatagramSocket(9876);

Този ред създава обект serverSocket от тип DatagramSocket с номер на порт 9876. Предаването и приемането на всички данни от сървъра се осъществява с помощта на този обект. В случая за няколко клиента всички те ще осъществяват обмен на данните със сървъра посредством единствен сокет serverSocket.

String sentence = new String(receivePacket.getData());

InetAddress IPAddress = receivePacket.getAddress();

int port = receivePacket.getPort();

Тези редове осъществяват разопаковане на пакета, получен от клиента. Първата команда извлича от пакета данните и ги записва в стринговия обект sentence. Следващата - извлича IP адреса на изпращача, а третата - номера на порта. Извличането на IP адреса и номера на порта е обусловено от необходимостта да се идентифицира клиента за последващо предаване на модифицирания низ.

Тема 6. RPC и RMI

Алтернатива на сокетите е Remote Procedure Call (RPC). Вместо директната работа със сокети, програмистът работи като с локална процедура. RPC е технология за отдалечено извикване на процедури, позволяваща работа с услуги, разположени на отдалечен компютър така, както те биха се изпълнявали локално. Основната идея е създаването на фамилия от обекти, които могат да се намират върху различни машини и които могат да комуникират помежду си чрез стандартни мрежови протоколи.

RPC е удобен механизъм, облекчаващ взаимодействието на операционните системи и приложенията в мрежата. Този механизъм е надстройка над системата за обмен на съобщения на ОС, затова в редица случаи той позволява по-удобно и прозрачно организиране на взаимодействието на програмите в мрежата, обаче полезността му не е универсална. RPC е относително старо (от началото на "модерните" разпределени архитектури - 1976 г.). Системите на базата на RPC са базират на извикване на отдалечени процедури, а не на предаване на потоци от данни.

Концепция на отдалеченото извикване на процедури

Идеята за отдалечено извикване на процедури се състои в разширяването на добре известния механизъм за предаването на управлението и на данните вътре в програмите, изпълнявани на една машина и този - за предаване на управлението и на данните по мрежата. Средствата на отдалеченото извикване на процедурите са предназначени за облекчаване на организацията на разпределените изчисления. Най-голяма ефективност при използването на RPC се постига в тези приложения, в които съществува интерактивна връзка между отдалечените компоненти с малки времена на отговори и относително малко количество на предаваните данни. Такива приложения се наричат RPC-ориентирани.

Реализацията на отдалечените извиквания е съществено по-сложна от реализацията на извикванията на локални процедури. Понеже извикващата и извикваната процедури се изпълняват на различни машини, затова те имат различни адресни пространства, което създава проблеми при предаването на параметри и резултати, особено ако машините не са идентични. RPC параметрите не бива да съдържат указатели към клетки на паметта и стойностите на параметрите се копират от единия компютър за предаване към другия. Следващата разлика на RPC от локалното извикване е, че задължително се използват слоевете на протоколния стек, което не бива да се вижда явно, както при определянето на процедурите, така и в самите процедури.

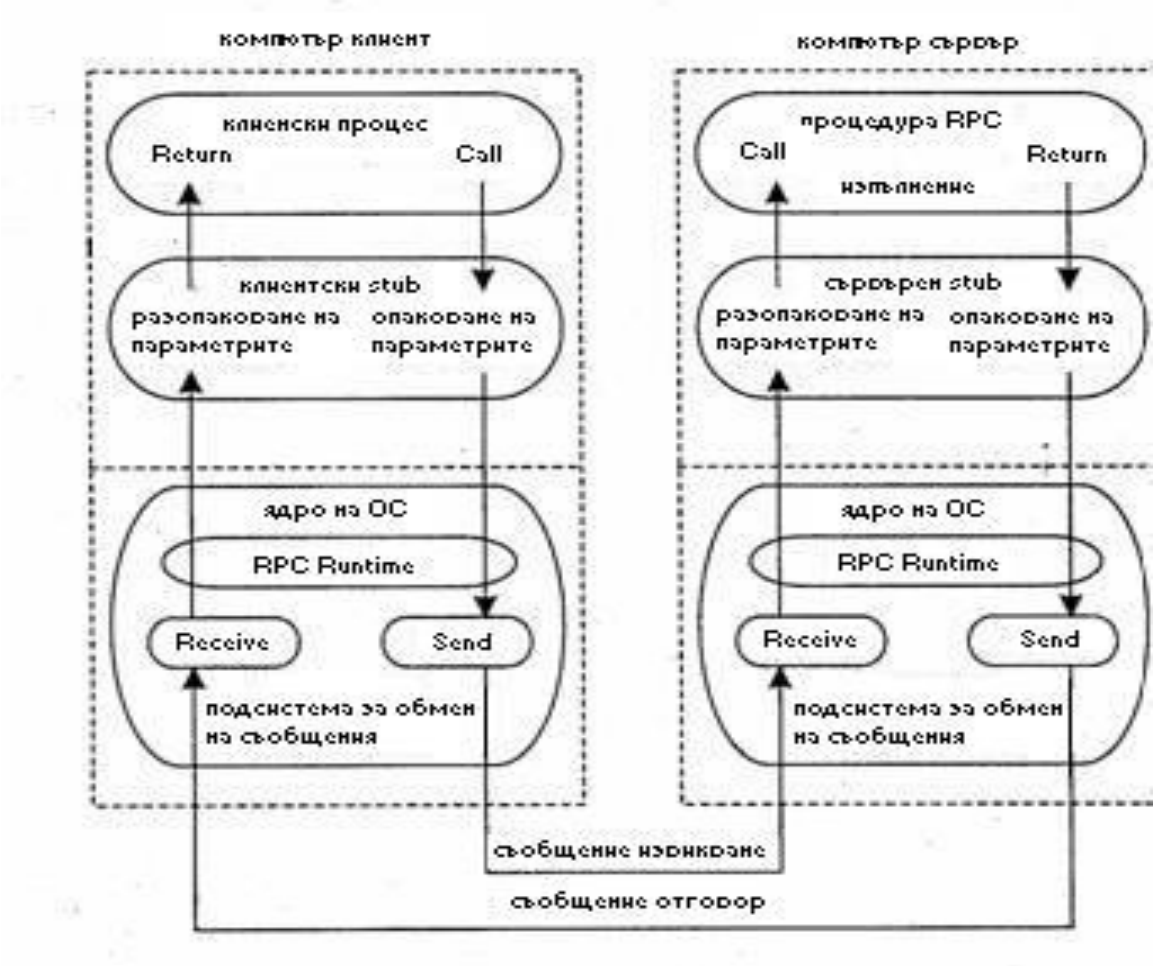
Отдалечеността внася допълнителни проблеми. Изпълнението на извикващата програма и на извикваната локална процедура на една машина се реализира в рамките на един процес. В реализацията на RPC участват минимум два процеса - по един на всяка машина. Ако единият от тях приключи аварийно, могат да възникнат следните ситуации: при авария на извикващата процедура процедурите на отдалеченото извикване ще "осиротеят"; при аварийно приключване на отдалечените процедури извикващите процедури ще станат "родители с убито потомство", като ще чакат безрезултатно отговора от отдалечените процедури.

Освен тези проблеми съществуват и редица други, свързани с неоднородността на езиците за програмиране и на операционните среди: структурите от данни и структурите на извикването на процедурите, поддържани от конкретния език за програмиране, не се поддържат точно така, както от другите езици. Тези и някои други проблеми разрешава широко разпространената технология RPC, лежаща в основата на много разпределени операционни системи.

Ще разгледаме изпълнението на извикването на локална процедура в автономно работеща машина. За да осъществи извикването извикващата процедура записва параметрите в стека в обратен ред. След като приключи изпълнението, записва резултата в регистър и връща управлението на извикващата процедура, която извлича параметрите от стека, като го връща в първоначалното му състояние.

Базови операции на RPC

Решението за избора на механизъм за предаване на параметрите се взема от разработчиците на езика. Понякога това зависи от типа на предаваните данни. Идеята, заложена в основата на RPC, се състои в това, така да се направи извикването на отдалечена процедура, че да изглежда по възможност аналогично (същото), както при извикването на локална процедура. С други думи, за да се направи RPC прозрачно на извикващата процедура да не се налага да знае, че извикваната процедура се намира на друга машина и обратното. Когато извикваната процедура на практика е отдалечена, в библиотеката се записва вместо локалната процедура друга версия на процедурата, наричана клиентски stub. Подобно на оригиналната процедура, stub-ът се извиква като се използва последователността, както се реализира прекъсване при обръщение към ядрото на ОС. Но за разлика от оригиналната процедура не се записват параметрите в регистри и не се заявяват данните от ядрото, а само се формира съобщение за изпращане към ядрото на отдалечената машина.



RPC механизмът постига прозрачност по следния начин. Когато извикваната процедура наистина е отдалечена, в библиотеката на процедурите вместо локалната реализации на оригиналния код на процедурата се разполага друга версия на

процедурата, наричана клиентски stub. На отдалечения компютър, който изпълнява ролята на сървър за процедурите, се поставя оригиналния код на извикваната процедура и още един stub, наречен сървърен stub. Предназначението на клиентския и сървърния stub е да се организира предаване на параметрите на извикваната процедура и връщането на стойностите на процедурата по мрежата, при което кодът на оригиналната процедура, разположена на сървъра, трябва да е напълно съхранен. Stub-овете използват за предаването на данни по мрежата средствата на подсистемата за обмен на съобщения, т.е. съществуващите в ОС примитиви send и receive. Понякога в подсистемата за обмен на съобщения се заделя програмен модул, организиращ връзката на stub-овете с примитивите за предаване на съобщения, наричан RPCRuntime модул.

Етапи на изпълнение на процедура RPC

Подобно на оригинална процедура, клиентският stub се извиква чрез обичайното предаване на параметри чрез стека, но след това вместо изпълнението на системното извикване, работещо с локален ресурс, се реализира формиране на съобщение, съдържащо името на извикваната процедура и нейните параметри. Тази операция се нарича операция за опаковане на параметрите. След това клиентският stub се обръща към примитива send за предаването на това съобщение на отдалечения компютър, на който е разположена реализацията на оригиналната процедура. След като получи съобщението от мрежата, ядрото на ОС на отдалечения компютър извиква сървърния stub, който извлича от съобщението параметрите и извиква по обичайния начин оригиналната процедура. Сървърният stub за получаването на съобщението е длъжен предварително да извика примитива receive, за да определи ядрото от кого е пристигналото съобщение. Сървърният stub разопакова параметрите на извикването, записани в съобщението и по обичайния начин извиква оригиналната процедура, като й предава параметрите чрез стека. След приключване на работата на процедурата сървърният stub опакова резултата от нейната работа в ново съобщение и с помощта на примитива send предава съобщението по мрежата на клиенткия stub, а той връща по обичайния начин резултата и управлението на извикващата процедура. Нито извикващата процедура, нито оригиналната извиквана процедура са се променили от това, че са работили на различни компютри.

Когато ядрото получи управлението, то превключва контекстите, съхранява регистрите на процесора и картата на паметта (дескрипторите на страниците), установява нова карта на паметта, която ще се използва за работа в режима на ядрото. Понеже контекстите на ядрото и на потребителя се различават, ядрото трябва точно да копира съобщението в своето собствено адресно пространство, така че да му е достъпно, да запомни адреса на целта (евентуално и други полета на заглавието), а после да го предаде на мрежовия интерфейс. С това приключва работата на клиентската страна. Включва се таймерът за предаването и ядрото може или да изпълни циклично запитване за наличие на отговор или да предаде управлението на планировчика, който да избере някой друг процес за изпълнение. В първия случай се ускорява изпълнението на заявката, но липсва мултипрограмиране.

На сървърната страна постъпващите битове се приемат от апаратурата или от вградената или от оперативната памет. Когато се получи цялата информация се генерира прекъсването. Обработчикът на прекъсванията проверява правилността на данните на пакета и определя за кой stub следва да ги предаде. Ако в никой от stub-овете не се очаква този пакет, обработчикът е длъжен или да го сложи в буфера или изобщо да го игнорира. Ако съществува очакващ stub, тогава съобщението се копира за него. Накрая се изпълнява превключване на контекстите и като резултат се

възстановяват регистрите и картата на паметта (приемат тези стойности, каквито са имали в момента, преди извикването на receive).

От тук започва работата си сървърният stub. Той разопакова параметрите и ги записва по съответен начин в стека. Когато всичко е готово се изпълнява извикването на сървъра. След изпълнението на процедурата на сървъра резултатите се предават на клиента. Затова се изпълняват всички описани вече етапи, но в обратен ред.

Отдалечено извикване на методи (RMI - Remote Method Invocation)

Отдалечен обект е обект, методите на който могат да бъдат извикани от друга Java машина. Обект от този тип се задава чрез един или повече отдалечени интерфейси, които представляват Java интерфейси, декларирящи отдалечените методи.

Отдалеченото извикване на методи е извикването от друга Java машина на метод от отдалечения обект, дефиниран в отдалечения интерфейс. Извикването става по същия начин, както в локалната Java машина.

За да може програмата да извика методи на друго приложение се създава сървърно приложение. Методите на сървъра могат да бъдат извикани от друга самостоятелна клиентска програма. Сървърното приложение създава отдалечени обекти, чиито указатели трябва да са достъпни за клиента. Сървърът очаква клиентските заявки, които извикват достъпните методи на обектите на сървъра. Като правило, клиентът получава отдалечени указатели на един или на няколко обекта на сървъра, на които той може да извиква методите. Технологията RMI предоставя механизъм, чрез който клиентът и сървърът могат да комуникират един с друг предавайки информация както от клиента към сървъра, така и от сървъра към клиента. Такива приложения могат да се наричат приложения на базата на разпределени обекти.

Приложенията с разпределени обекти трябва да локализират отдалечения обект. За да стане достъпен отдалеченият обект е необходимо той да се регистрира с вградения механизъм rmiregistry, тогава указателят към този обект ще стане достъпен за клиента. На приложението могат да се предават указатели в хода на нормалното функциониране. Клиентът е длъжен да установи връзка с отдалечения обект. Всички детайли за установяването на връзката са възложени на RMI средствата, разработчикът се обръща към методите на отдалечения обект точно така, както с обикновените методи. RMI предоставя средства за предаване на кода на обектите и за предаването на данните. Извикващата програма може да предава обекти на отдалечените обекти.

RMI позволява да се зарежда байт-кода на класа на обекта, ако този клас не е описан и не е определен на виртуалната машина, която ще получава кода. Типът и цялото поведение на обекта, който преди е бил достъпен само за една виртуална машина, ще бъдат предадени на друга отдалечена виртуална машина. Поведението на обекта при това няма да бъде променено.

Както всички приложения, разпределените приложения, построени на базата на RMI, са изградени от интерфейси и класове. Интерфейсите определят методите, класовете имплементират методите, описани в интерфейсите, а също така могат да съдържат допълнителни методи, неописани в интерфейсите. В разпределените приложения имплементациите се разполагат на различни виртуални машини. Обектите, методите на които са достъпни от други виртуални машини, се наричат отдалечени обекти. За да може обект да стане отдалечен той трябва да имплементира отдалечен интерфейс. Отдалечен интерфейс се създава на базата на интерфейса Java.rmi.Remote. Освен типичните изключителни ситуации, всеки метод на отдалечения интерфейс е длъжен да съдържа в своето описание фрагмент throws, свързан с изключение от типа java.rmi.RemoteException. В RMI модела отдалеченият обект, предаван от една виртуална машина към друга, се различава от "неотдалечения" обект. Вместо да се

предаде копие на обекта на отдалечената машина, RMI предава класа stub на отдалечения обект. Класът stub работи в качеството на прокси-обект, т. е. обект, който представлява отдалечен обект за тази машина, където е бил предаден. Приложението, извикващо отдалечен метод, фактически извиква метод от класа stub, разположен на тази машина, където е извикващото приложение. Класът stub на отдалечения обект имплементира същия набор от интерфейси, както и отдалечения обект. Това позволява да се приведе обектът stub към всеки един тип в съответствие с имплементираните интерфейси.

Процесът на създаване на разпределено приложение на основата на RMI се състои от следните етапи:
1. Разработка и имплементация на компонентите на разпределеното приложение.
2. Компилиране на кода, получаване на съответстващите класове stub.
3. Регистрация на класовете, за да станат класовете достъпни за отдалечените програми.
4. Стартиране на приложенията.

Разработка на компонентите

При това се осъществяват следващите стъпки:
• О*писание на отдалечения интерфейс.* Отдалеченият интерфейс описва методите, които могат да бъдат извиквани от клиента на отдалечен обект;
• *Имплементация на отдалечените обекти.* Отдалечените обекти могат да имплементират един или няколко отдалечени интерфейси. Класът на имплементациите може да имплементира и други интерфейси (локални или отдалечени) и други методи, достъпни локално. Ако в качеството на параметри се използват типове, съответстващи на локалните класове или ако методите връщат стойности на типовете, описани в локалните класове, тогава тези класове също трябва да се имплементират;
• *Имплементация на клиента.* Клиентът може да бъде имплементиран по всяко време, даже след като отдалечен обект вече е бил разположен и регистриран.

Компилация и създаване на класове stub. Етапът на компилация се състои от две стъпки. Първата стъпка използва компилатора javac. С негова помощ се компилират изходните файлове, сред които и файловете, съдържащи имплементация на отдалечените интерфейси (сървърни класове) и клиентски класове. На втория етап се използва компилаторът rmic. С негова помощ се създават класовете stub, съответстващи на отдалечените обекти. Класовете stub на отдалечените обекти се използват за създаване на прокси-обекти на машините, обръщащи се към отдалечените обекти.

Регистрация на класовете. Всички създадени класове, свързани с отдалечения интерфейс, класовете stub и другите класове трябва да са достъпни за клиента. Следва те по съответен начин да се регистрират на сървъра или да се разположат в такива каталози, откъдето да могат да бъдат получени свободно от клиента.

Стартиране на приложението. За да се стартира разпределеното приложение е необходимо да се стартира регистъра на отдалечените обекти, после сървъра, след което и клиентът на приложението.

Клиентският стъб изпълнява следните действия: Определя местоположението на сървъра, за който е предназначено извикването (binding); Извиква процедурата и опакова аргументите за нея в съобщение (marshaling); Полученото съобщение се преобразува в поток от байтове (serialization) и се изпраща по мрежата към сървъра.

След като получи съобщението-отговор от сървъра, го разопакова и го връща на клиента в качеството на резултат от работата на процедурата.

Тема 7. SGML, HTML и XML

SGML (Structured Generalized Markup Language)

Стандартият обобщен маркиращ език представлява родоначалник на всички хипертекстови езици. Езиците HTML и XML са получени от SGML (макар и по различен начин). HTML е приложение на SGML, а XML е подмножество на SGML, разработено е с цел опростяване на процеса за машинното четене на документи. SGML определя базовия синтаксис, като също така дава възможност за създаване на собствени елементи (от тук е и термина обобщен в наименованието на езика). За да се използва SGML за описание на определен документ, трябва да се обмислят съответстващия набор от елементи и структурата на документа. Например за да се опише една книга, трябва да се използват създадени елементи с наименования, като например: BOOK, PART, CHAPTER, INTRODUCTION, A-SECTION, B-SECTION, C-SECTION и т.н.

Наборът от най-често използваните елементи за описание на документ от определен тип, се нарича SGML приложение. SGML приложението включва в себе си правила, установяващи начините за организация на елементите, а също така и други особености на тяхното взаимодействие.

SGML е наследник на създадения през 1969 г. в IBM език GML (Generalized Markup Language). Първоначално SGML е бил разработен за съвместно използване на машинно-четими документи в много от правителствените и космическите проекти в САЩ. Той широко се е използвал в печатната и издателска сфера, но неговата сложност е затруднила масовото му разпространение и приложение.

Основните части на SGML документа са:

- SGML декларация, която определя какви символи и ограничители могат да се използват в приложението;
- Document Type Definition (DTD) - определя синтаксиса на конструкциите за маркиране. DTD може да включва допълнителни определения, като препратки към символи (мнемоники);
- Спецификацията на семантиката, свързана с препратките, също така дава ограничения за синтаксиса, които не могат да бъдат изразени вътре в DTD;
- Съдържимото на SGML документа трябва да включва поне коренов елемент.

HTML (Hypertext Markup Language)

Авторът на уеб, професор Тим Бърнърс-Лий, през периода 1989 - 1991 г. развива идеите за хипертекст, предложени още през 40-те и 60-те години на миналия век от Буш и Нелсън. Професор Тим Бърнърс-Лий със своите асистенти разработват първоначалните версии на HTML езика, на HTTP протокол, на уеб сървър и браузър. Така се получава, че фундаментът на Интернет е авторство на един единствен човек. Целта тогава е била осъществяване на комуникация между физиците в ЦЕРН.

Езикът HTML предоставя фиксиран набор от елементи, които могат да се използват за разместване на компоненти на уеб страници. Примери за такива елементи са заглавия, абзаци, списъци, таблици, изображения и препратки. Например HTML отлично подхожда за създаване на лична страница.

Всеки елемент започва с начален таг: текст, заключен в ъглови (счупени) скоби (< >), който съдържа името на елемента и допълнителна информация. Повечето елементи завършват със затварящ таг, който повторя съответния начален таг, изключение прави употребата на наклонена черта (/) пред името на елемента. Елементът съдържание представлява текст, разположен между начален и затварящ тагове. Много елементи съдържат вложени елементи.

Браузърът, изобразяващ HTML страницата, разпознава всеки от тези стандартни елементи и ги изобразява в съответния формат. Например обикновено браузърът изобразява заглавието H1 с най-голям размер на шрифта, заглавието H2 – с по-малък размер на шрифта, а елементът P – с още по-малък размер на шрифта. Елементът LI се изобразява като абзац на текст в състава на маркиран списък. Елементът A браузърът преобразува в препратка (подчертан текст), на който потребителят може да кликне, за да премине към друго място на текущата страница или на друга страница.

Макар че наборът от HTML елементи е бил съществено разширен в сравнение с първата версия на HTML, езикът HTML както и преди не е пригоден за представянето на много от типовите документи. Следва представяне на примери за документи, които не могат да бъдат адекватно описани с помощта на езика HTML:

- *Документ, който не съдържа типови компоненти* (заглавия, абзаци, списъци, таблици и т.н.). Например в HTML езика няма елементи, необходими за изобразяване на музикални символи или математически уравнения.
- *База от данни,* например каталог с книги. Може да се използва HTML страница за съхраняване и изобразяване на информация от статична база данни (например списък от книги и тяхното описание). Обаче, ако се наложи да се направи сортировка, филтрация, търсене и обработка на информацията, ще трябва да се използва друг готов софтуер (като например за програма, работеща с бази от данни, може да се приложи Microsoft Access). В HTML езика не са предвидени съответстващи елементи.
- *Документ, който трябва да се представи във вид на йерархична структура*. Да допуснем, за написването на една книга, тя трябва да се разбие на части, глави, раздели A, B, C и т.н. Също така трябва впоследствие да може програмата да използва дадената структура на документа за създаване на заглавия, за оформяне на различни нива в структурата с помощта на различни стилове, извлечения от определени раздели, и още например обработка на информацията с други техники. Обаче елементът от тип заглавие в HTML съдържа само описание на собствения текст. Например вътре в елемента от тип заглавие не се задават вложени елементи на текста, които са свързани с разделите на документа. Тези елементи не могат да бъдат използвани за представяне на йерархична структура на документа.

Езикът XML позволява да се преодолеят тези ограничения.

XML (Extensible Markup Language) решава тези проблеми

Изглежда, че SGML езикът напълно подхожда за описание на уеб документи. Обаче разработчиците от консорциума World Wide Web Consortium (W3C) са решили, че той е прекалено сложен и фундаментален за да представя ефективно информацията в уеб. Гъвкавостта и изобилието от средства, поддържани от SGML, затруднява написването на програмно осигуряване, необходимо за обработка и изобразяване на SGML информацията в браузърите. Възниква идеята да се приспособи част от SGML езика специално за разполагане на информация в уеб. През 1996 г. групата XML Working Group разработва клон на езика SGML, като го именова разширяем език за маркиране на хипертекст.

Ето как го описват неговите създатели: "Разширяемият тагов език Extensible Markup Language (XML) е съставна част на езика SGML… Той е предназначен за по-лесно използване на езика SGML в уеб и за изпълнение на задачи, които към настоящия момент се реализират с езика HTML. XML е разработен с цел усъвършенстване на използването и на взаимодействието на езиците SGML и HTML." Това е записано в спецификацията на версия 1.0 XML на създадения XML от Working Group през февруари на 1998 г.

XML е опростена версия на SGML, приспособена за уеб. Както и SGML, XML дава възможност за разработване на собствени набори от елементи за описание на определен документ. Както и при SGML, в тялото на програмата може да бъде определено XML приложение (или речник), което съдържа набор от най-често използваните елементи с общо предназначение и структурата на документа, която може да бъде използвана за описание на документ от определен тип (например документи, съдържащи математически формули или векторна графика).

Синтаксисът на XML е значително по-прост, от този на SGML, което подпомага възприятието на XML документите, а също така и написването на програмите браузъри, кодовете и уеб страниците за достъп и представянето на информацията за документа.

Описанието на езика XML представлява описание на оператори, написани така, че да се спазва определен синтаксис. Когато се създава XML документ, тогава вместо да се използва ограничен набор от определени елементи, могат да се създават собствени елементи и да им се присвояват всякакви имена по избор. Именно поради тази причина езикът XML е разширяем (extensible). Следователно можем да използваме XML за описание на практически всеки документ, от музикална партитура до база от данни.

За описание на бази от данни в XML са предвидени възможности за работа с няколко формата (например формат .mdb Access или .dbf dBase).

Езикът XML е построен на принципа на отворените и достъпни стандарти.

Имената на елементите в XML документа не са определени от езика XML. Тук само се задават тези имена при създаването на определен документ. За елементите могат да се изберат всякакви коректно зададени имена.

Когато се присвояват имена в XML документ, добре е, по възможност, те да са най-информативни. Едно от предимствата на XML документа е това, че на всеки фрагмент с информация може да бъде присвоено информативно описание.

Увод в XML

Таговете се състоят от текст затворен между знаците за по-малко (”<”) и по-голямо (”>”), както в <tag>. Таговете биват два вида: за отваряне на област (като <start>) и за затваряне на дадена област. Таговете за затваряне на дадена област изглеждат почти по същия начин, като тези, използвани за отваряне на област, само че имат наклонена черта (”/”) точно след отварящия знак за по-малко (като </start>). SGML също така дефинира атрибути за таговете, които представляват различна информация, сложена в дадения таг, както когато се задава адреса на една картинка в HTML: <img src=”picture.jpg”>.

SGML е използван за да се дефинира DTD за HTML и още се използва за да се пишат DTD-а за XML. Проблемът на SGML е, че позволява да се използва неправилен синтаксис, което правеше създаването на четец за HTML трудна задача:

- Някои тагове не изискваха затварящ таг, като <img>. Добавянето на затварящ таг води до грешка.
- Таговете можеха да се слагат във всякакъв ред. Например Това е пробен низ , се третира като правилно разположение, въпреки че разположението на таговете не е симетрично.
- Някои атрибути не изискват стойности, като nowrap в <td nowrap>.
- Атрибутите могат да бъдат дефинирани както с кавички, така и без. Това означава, че <img src=”picture.jpg”> и <img src=picture.jpg> се третираха по един и същ начин.

Всички тези бележки правят създаването на SGML базиран език наистина трудна задача. И тук е мястото където се намесва XML.

XML не се стреми към свободата, която SGML предлага. Вместо това дефинира следните правила:

- Всеки отварящ таг трябва задължително да има и затварящ.
- Ако за даден таг не е предвиден затварящ такъв, тогава в края на въпросния таг се добавя ”/”, като в <img />.
- Таговете трябва да се затварят в точният ред, в който се отварят, като Това е пробен низ .
- Всички атрибути задължително изискват стойности.
- Всички стойности на атрибутите трябва задължително да са обградени в кавички.

Тези прости правила, по които е изграден XML, подпомогнаха да се избегнат всичките проблеми около SGML. Поради това само за първите шест години от XML се появиха различни нови езици, базирани на него: MathML, SVG, RDF, RSS, SOAP, XSLT, XSL-FO и съответно това доведе до реформиране на HTML в XHTML.

Днес XML е една от най-бързо развиващите се технологии в света. Неговата основна цел е да представя информация в структуриран вид, използвайки обикновен текст. XML файловете обаче не са като базите от данни, където информацията също се съхранява в структуриран вид.

Съществуват три основни начина за да се съобщи на браузъра как да обработва и да изобразява всеки от създадените XML елементи:

- *Таблица със стилове.* С помощта на този метод се свързва таблица със стилове с XML документа. Таблицата със стиловете представлява отделен файл, съдържащ инструкции за форматирането на отделните XML елементи. Може да се използва както каскадна таблица със стилове (Cascading Style Sheet – CSS), която също така се прилага и за HTML страници, така и разширяема таблица във формата на езика на стиловите таблици (Extensible Stylesheet Language – XSL), притежаваща значително по-големи възможности от CSS и е разработена специално за XML документи.
- *Свързване на данните.* Този метод изисква създаване на HTML страница, свързване към нея на XML документ и установяване на взаимодействието на стандартните HTML елементи на страницата, такива като SPAN или TABLE, с XML елементите. Впоследствие HTML елементите автоматично изобразяват информацията от свързаните с тях XML елементи.
- *Написване на скрипт.* При този метод се създава HTML страница, тя се свързва с XML документа и така се осигурява достъп до отделните XML елементи с помощта на специално написан скрипт (с JavaScript или с Microsoft Visual Basic Scripting Edition [VBScript]). Браузърът възприема XML документа като обектен модел на документа (Document Object Model – DOM), съставен от голям набор от обекти, свойства и команди. Написаният код позволява да се осъществява достъп, изобразяване и манипулиране с XML елементите.

DOM (Document object model)

Самостоятелно DOM е може би една от най-големите иновации в Интернет, след като HTML беше първоначално използван за да се свържат отделните документи заедно. DOM дава на програмистите неограничен достъп до HTML, осигурявайки им възможност да го видят като XML документ. DOM представлява по-високо ниво

DHTML (динамичен HTML), пресъздаден от Майкрософт и Нетскейп в един истински мултиплатформен самостоятелен език.

Другите DOM-ове

Всеки език, базиран на XML, като XHTML и SVG, може да се възползва от ядрото на DOM, защото технически тези езици са си XML. Въпреки това, много езици дефинират свои собствени DOM-ове за да разширят ядрото на XML и да си добавят собствени спецификации.

Едновременно с разработването на DOM за XML, W3C разработва и DOM, специлиазиран за XHTML. Този DOM добавя в спецификацията, че всеки елемент се представя от свой собствен тип, като HTMLDivElement, което отговаря на . Това правило има много малко изключения, за много малко елементи, за които не се изискват специални свойства или методи, различни от тези, които всеки елемент получава по принцип.

XML ще замени ли HTML?

Към настоящия момент отговорът на този въпрос е отрицателен. HTML както и преди остава основен език за съобщения за браузъра, как той да изобразява информацията в уеб.

Без да заменя HTML, XML се използва съвместно с него като съществено разширява възможностите на уеб страниците за:

- виртуално представяне на документи от всякакъв тип;
- сортировки, филтрации, подреждане, търсене и манипулиране с информация с други методи;
- представяне на информацията в структуриран вид.

Както заявяват самите разработчици, XML е бил създаден за взаимодействие с HTML и за съвместното им използване.

XML приложения, повишаващи качеството на XML документите

Освен XML приложенията за описание на определен клас документи съществуват няколко XML приложения, които могат да се използват вътре в XML документ от всеки тип. Тези приложения подпомагат създаването на документа и подобряват неговото качество. Примери за такива приложения са:

- Extensible Stylesheet Language (XSL) позволява създаването на мощни стилови таблици с използването на синтаксиса на XML.
- XML Schema позволява разработването на подробни схеми за вашите XML документи с използването на стандартния XML синтаксис, което представлява по-мощна алтернатива на използването на DTD.
- XML Linking Language (XLink) дава възможност за свързването на XML документите. Той поддържа множествени целеви указатели и други полезни функции, осигуряващи значително по-голяма свобода, в сравнение с механизма за организация на препратки в HTML.
- XML Pointer Language (XPointer) позволява да се определят гъвкави целеви указатели.

При съвместно използване на XPointer и XLink могат да се организират препратки към всяко едно място в целевия документ, а не само преходи към специално отделени пунктове.

XML е не само полезен инструмент за описание на документи, но и служи като база за изграждане на приложения и разширения.

Тема 8. JavaScript

JavaScript е относително прост обектно-ориентиран език, предназначен за създаване на неголеми клиентски и сървърни приложения за Интернет. Програмите, написани на езика JavaScript, се вграждат в състава на HTML документите и се разпространяват заедно с тях. Браузърите разпознават вградените в текста на документа скрипт кодове и ги изпълняват. JavaScript е интерпретируем език за програмиране. Примери за програми на JavaScript са програми, проверяващи въведени потребителски данни или изпълнение на някакви действия при отварянето или затварянето на документа. Такива програми могат да реагират на действията на потребителя, като натискане бутона на мишката, въвеждане на данни във форма или преместване на мишката по страницата. JavaScript програмите могат да управляват самия браузър и атрибутите на документа.

Езикът JavaScript прилича по синтаксис на езика Java, с изключение на обектния модел. Той не притежава свойства, като статични типове данни и строга типизация. В JavaScript, за разлика от Java, понятието класове не е основа на синтактичните конструкции на езика. Основата представлява неголям набор от предопределени типове данни, поддържани от системата: числови, булеви и стрингови; функции, които могат да бъдат както самостоятелни, така и методи на обекти (метод в терминологията на JavaScript е функция/подпрограма); обектен модел с голям набор от предопределени обекти със свои свойства и методи, а също така и правила за задаване на нови обекти в програмата на потребителя.

Възможности на езика JavaScript

С помощта на JavaScript може динамично да се управлява визуализацията и съдържанието на HTML документите. Може да се запише в изобразявания на дисплея документ произволен HTML код в процеса на синтактичния анализ на заредения в браузъра документ. С помощта на обекта Document могат да се генерират документи "от нулата" в зависимост от предишните действия на потребителя или някакви други фактори.

JavaScript позволява да се контролира работата на браузъра. Например обектът Window поддържа методи, позволяващи извеждане на дисплея на диалогови прозорци, създаване, отваряне и затваряне на нови прозорци, задаване на размери на прозорците и т.н.

JavaScript позволява взаимодействие със съдържимото на документите. Обектът Document и съдържащите се в него обекти позволяват на програмите да прочитат части от HTML документа и понякога да взаимодействат с тях. Самият текст да се прочете е невъзможно, но може например да се получи списък на хипертекстовите препратки, които са налични в дадения документ. Много възможности за взаимодействие със съдържимото на документите осигурява обекта Form и обектите, които той може да съдържа: Button, Checkbox, Hidden, Password, Radio, Reset, Select, Submit, Text и Textarea.

Ядрото на JavaScript

Ядрото на JavaScript съдържа набор от основни обекти като Array, Date и Math, и основен набор от елементи на езика като операции, структури за управление и оператори. Ядрото на JavaScript може да бъде разширено за различни цели чрез допълване с нови обекти.

Клиентският JavaScript разширява ядрото на езика, предоставя обекти за управление на браузъра (Netscape Navigator или друг уеб браузър) и DOM. Например

клиентските разширения дават на приложението възможност да се разполагат елементи в HTML форма и да се реагира на действията на потребителя, като щракване на желаното място с мишката, въвеждане на данни във форма и навигация по страниците.

Сървърният JavaScript разширява ядрото на езика, предоставяйки обекти, свързани със стартирането на JavaScript на сървъра. Например сървърните разширения дават на приложението възможност за съединяване с релационната БД, да се съхранява информацията между извикванията на приложението или да се изпълняват различни действия с файловете на сървъра.

JavaScript позволява да се създават приложения, работещи по цялата мрежа Интернет. Клиентските приложения работят на браузъра, а сървърните приложения - на сървъра. Използвайки JavaScript, могат да се създават динамични HTML страници, обработващи потребителския вход и наличните данни, използвайки специални обекти, файлове и релационни бази от данни. Например клиентският JavaScript включва обекта form за представянето на формата на HTML страницата, а сървърният JavaScript включва обекта database за връзка с външна релационна БД. С помощта на функционалността на JavaScript LiveConnect може да се организира взаимодействие на кодовете на Java и JavaScript. От JavaScript може да се инстанциират Java обекти и да се получи достъп до техните public методи и полета. От Java може да се получи достъп до обекти, свойства и методи на JavaScript.

Клиентският и сървърният JavaScript имат следните общи елементи:

Ключови думи;

Синтаксис за оператори и граматика;

Правила за написване на изрази, променливи и литерали;

Базов е обектния модел (макар че клиентският и сървърният JavaScript имат различни предопределени обекти);

Предопределени обекти и функции, такива като Array, Date и Math.

Клиентски JavaScript

Уеб браузърите могат да интерпретират операторите на клиентския JavaScript, внедрен в HTML страницата. Когато браузърът (или клиентът) заявяват такава страница, сървърът изпраща на клиента по мрежата пълното съдържание на документа, включително HTML и операторите на JavaScript. Браузърът прочита страницата отгоре надолу, като изобразява резултата от работата на HTML и изпълнява операторите на JavaScript. Този процес води до резултата, който вижда потребителя.

Операторите на клиентския JavaScript, вградени в HTML страницата, могат да реагират на потребителските събития, такива като щракване на желаното място с мишката, въвеждане на данни във формуляр и навигация по страниците. Например може да се напише функция на JavaScript за проверка на въвежданата от потребителя информация във формата дали е коректна. Без да се предава по мрежата вградения JavaScript в HTML страницата могат да се проверят въведените данни и да се изведе диалогов прозорец, ако потребителят е въвел неверни данни.

Сървърен JavaScript

На сървъра също така може да се внедрява JavaScript в HTML страници. Сървърните оператори могат да се съединяват с релационните бази от данни на различните производители, да разделят информацията между потребителите на приложенията, да се получи достъп до файловата система на сървъра или да се

взаимодейства с други приложения посредством LiveConnect и Java. HTML страниците със сървърен JavaScript могат да съдържат също така клиентски JavaScript.

За разлика от страниците с чисто клиентски JavaScript, HTML страниците, използващи сървърен JavaScript, се компилират в байт кодови изпълними файлове. Тези приложения се задават за изпълнение на уеб сървъра от JavaScript. Поради тази причина създаването на приложения на JavaScript е процес от два етапа. На първия етап се създават HTML страници (които могат да съдържат оператори както за клиентския, така и за сървърния JavaScript) и JavaScript файлове. След това се компилират всички тези файлове в един изпълним блок. На втория етап страницата на приложението се заявява от клиентския браузър. Сървърът използва изпълнимия блок за преглед на изходната страница и за динамично генериране на HTML страница, връщана на клиента. Тя изпълнява всички открити на страницата оператори на сървърния JavaScript. Изпълнението на тези оператори може да добави нови HTML оператори или оператори на клиентския JavaScript в HTML страницата. Сървърът изпраща след това окончателния вариант на страницата по мрежата на Navigator на клиента, който изпълнява клиентския JavaScript и изобразява резултата.

Услугата Session Management Service на сървърния JavaScript съдържа обекти, които могат да се използват за работа с данни, съществуващи между клиентските заявки за няколко клиента или за няколко приложения. Услугата LiveWire Database Service на сървърния JavaScript предоставя обекти за достъп до бази от данни, служещи за интерфейс на Structured Query Language (SQL) сървърите.

За да могат да се ползват приложенията на JavaScript на сървъра, се налага да се зареди машина за изпълнение на JavaScript в Server Manager, като в Programs се избира сървърен JavaScript. След появяването на промпта "Activate the JavaScript application environment?" се задава Yes и OK.

На сървърната страна се работи с ресурсите на сървъра. Обектите на тази среда трябва да могат да манипулират с релационната база от данни и с файловата система на сървъра. HTML страницата не се изобразява на сървъра. Тя се заявява на сървъра за визуализация при клиента. Заявената страница може да съдържа клиентски JavaScript. Ако заявената страница е част от JavaScript приложение, сървърът може да генерира тази страница от готови компоненти.

Предварителната обработка на данните при клиента може да намали изискванията към пропускателната способност на мрежата. Обикновено подходите за разпределяне на приложенията между сървъра и клиента са различни. Някои задачи могат да се изпълняват само на клиента или само на сървъра, а други могат да се изпълняват на някой произволен от тях.

Услугата JavaScript Session Management Service предоставя обекти за съхраняване на информация, а клиентският JavaScript е преходящ. Клиентските обекти съществуват само докато потребителят има достъп до страницата. Сървърите могат да обединяват информация от много клиенти и от много приложения и могат да съхраняват големи обеми от данни в база от данни.

JavaScript и Java

JavaScript и Java имат фундаментални разлики. JavaScript поддържа по-голямата част от синтаксиса на изразите в Java и базовите конструкции за управление на потока.

JavaScript има обектен модел на базата на прототипи вместо по-общия обектен модел на базата на класове. Моделът на базата на прототипи предоставя възможност за динамично наследяване, т.е. това, което се наследява, може да се различава за различните обекти. Функциите могат да бъдат свойства на обектите, изпълнявани като нетипизирани методи.

JavaScript е език, свободен по форма, сравнен с Java. Не е задължително обявяването на всички променливи, класове и методи. Но следва да се отчита дали тези методи са public, private или protected, и не се налага да се реализират интерфейси.

Типът на променливите, параметрите и функциите return не е зададен явно.

Модел на обектите в JavaScript - обекти на Navigator

JavaScript е разработен от корпорацията Netscape междуплатформен обектно-ориентиран скриптов език през 1995 г. Всички операции, които могат да се изпълнят в програмата на JavaScript, описват действия над добре известни и разбираеми обекти, които представляват елементи на работната област на програмата Netscape Navigator и контейнери на езика HTML. Всъщност обектната ориентираност на JavaScript тук и свършва. Има само обекти с набор от свойства и набор от функции с тези обекти. Последните се наричат методи. Освен методите съществуват и други функции, които повече приличат на функции от традиционните езици за програмиране и позволяват да се работи със стандартните математически типове или да се управлява процеса на изпълнение на програмата. Също така в JavaScript има събития - аналог на програмните прекъсвания. Програмистът използва събития за изпълнение на определени части от програмния код на скрипта. Тези събития са ориентирани за работа в WWW, например зареждане на страница в работната област на Navigator или избор на хипертекстов линк. Всички вградени обекти на JavaScript водят началото си от работната област на Netscape. Комбинирайки браузърния и документния обектни модели (DOM), JavaScript позволява да се манипулира с всички елементи на страницата като с обекти от прозореца надолу по йерархията. Освен тези класове обекти потребителят може да създава и свои собствени.

Методи и свойства на обектите. Управление на потока от изчисления

Всеки един от класовете има функции за управление на обектите на класа - методи. Основните са тези, които позволяват преназначаване на смисъла на обектите. Това обикновено се прави с операция за присвояване. Всички типове оператори, които се поддържат от по-разпространените езици за програмиране, са реализирани в JavaScript. Освен операциите с числа и описание на стандартни класове в JavaScript има команди за управление на потока от изчисления (като break; continue; for; while, if..else и т.н.). Операторът за обявяване на променлива е var. Типът на променливата се определя от присвоената й стойност.

Вградени обекти

В JavaScript съществуват вградени обекти, които осигуряват някои много полезни методи. Най-често използваните вградени обекти са String, Date и Math. Обектът String предлага набор от методи и функции за обработка на текстови низове. Обектът Math притежава методи за изпълнение на различни математически функции, а обектът Date дава възможност за работа с времена и дати. Най-често в програмата на JavaScript се използват вградени масиви, най-вече графични образи (Images) и хипертекстови препратки (Links).

Типът Array е въведен за осигуряване на възможност за манипулиране с най-различни обекти, които се визуализират от Navigator-а. Например списък на всички хипертекстови препратки на дадената страница на уеб сайта, списък от всички графични образи на дадената страница, списък от всички аплети на дадената страница, списък на всички елементи на формата и т.н. Потребителят може да създаде и свой собствен масив, използвайки конструктора Array(). За масивите са определени три метода: join, reverse, sort.

Създаване на обекти в JavaScript

Всяка функция може да бъде използвана като конструктор:

```
function Dog() {}
d = new Dog();
```

Типичен конструктор:

```
function Dog(name) {
        this.name = name;
}
```

Тук не се получава както в C++ обект, представляващ екземпляр на класа, а се определя функция конструктор. Първо new създава нов празен обект. После изпълнява извикване към него на функцията с нов празен обект като стойност this. Така се създават шаблони за обекти. В JavaScript името на конструктора е име на класа, който се моделира. В определянето на Dog е зададена променливата на екземпляра с името name. Всеки обект, създаден с използването на Dog в качеството на функция конструктор, ще има собствено копие на променливата на екземпляра name.

В JavaScript обектът се създава като копие на съществуващ екземпляр (т.е. прототип) на обекта. Всички свойства и методи на обекта прототип стават свойства и методи на обекта, създаден от конструктора на този прототип. Обектите наследяват своите свойства и методи от своя прототип. В JavaScript всяка функция има свойство prototype, сочещо към обекта прототип. Обектът прототип има свойство constructor, сочещо обратно към самата функция. Когато функцията се използва за създаване на обект с new, полученият обект ще наследи свойството Dog.prototype. Обектът Dog.prototype има свойство constructor, сочещо обратно към функцията Dog. Така всеки обект Dog ще има свойство constructor, сочещо към функцията Dog. Dog.prototype е обект създаден неявно с извикването на функцията конструктор Object:

```
Dog.prototype = new Object();
```

Диалогови прозорци за предупреждение, въвеждане на данни и потвърждение

JavaScript осигурява възможност за комуникация с потребителя посредством три прозореца: alert (извежда съобщение за предупреждение и бутон), confirm (извежда съобщение и два бутона), prompt (извежда съобщение, поле за текст и два бутона).

Добавяне на JavaScript към уеб страници

Съществуват три основни начина за добавяне на JavaScript към уеб страници:

- чрез вграждане на JavaScript код в уеб страници. Когато се налага използване на малък скрипт, най-добрият избор е да се вгради в HTML документа. В този случай JavaScript кодът се записва в тялото на документа (след тага <BODY>). За обозначаване на програмния текст се използва таг <SCRIPT>;

- чрез поставяне на JavaScript код в секция <HEAD> на документа. Ако се използва код многократно на дадена страница, той може да се оформи във вид на функция и да се запише в секцията <HEAD> на документа;

- чрез връзка към JavaScript код, съхраняван във външен файл. Когато се използва JavaScript в няколко документа е рационално той да бъде записан в отделен текстов файл и след това да се импортира в отделните документи. Текстът във файла трябва да съдържа необходимия JavaScript код. В тага <SCRIPT>, където трябва да се разположи импортирания текст от файла се използва атрибут scr, чиято стойност е наименованието на файла с код на JavaScript. Пример за използване на външен файл: <SCRIPT SCR=" extern.js" TYPE="text/javascript".

Тема 9. CGI, FastCGI, ISAPI, Java servlet

CGI е едно от най-разпространените средства за създаване на динамични уеб страници. Това не е отделен език, а множество от спецификации, които позволяват на уеб сървъра да комуникира с други приложения и програми. Също така е стандарт, който определя начина, по който уеб сървърите могат да използват външни програми. CGI програмите се изпълняват в реално време. Това дава възможност за генериране на динамично съдържание. Заявката към URL адрес на дадена CGI програма се формира след като потребителят избере хипервръзка или подаде формуляр. За да направи заявката браузърът използва HTTP. Когато уеб сървърът получи заявка, той изпълнява CGI програма, като й предава изпратените от браузъра данни. Когато CGI приложението извърши обработката на данните, обикновено то генерира нови данни под формата на уеб страница, която се връща на браузъра.

CGI определя протокола за обмен на данни между сървъра и програмата. CGI определя реда на взаимодействието на сървъра с приложната програма, при което сървърът е инициращата страна. CGI определя и механизма на реален обмен на данни и управляващи команди в това взаимодействие, което не е определено от HTTP. Такива понятия, както метод на достъп, променливи на заглавието, MIME, типове данни, са заимствани от HTTP и правят спецификацията прозрачна за тези, които знаят този протокол. CGI скрипт е програма написана в съответствие със спецификацията Common Gateway Interface. CGI скриптовете могат да бъдат написани на избран език за програмиране (C, C++, PASCAL, FORTRAN и т.н.) или на команден език (shell, cshell, команден език на MS-DOS, Perl и т.н.). Най-разпространените езици, използвани за CGI са tcl, AppleScript, C, Java, Visual Basic и Perl.

Общата схема на работа на CGI се състои от следните елементи:

- *Получаване от уеб сървъра на информация от клиента (браузъра).* Данните се предават от формата на HTML документа към уеб сървъра. Формата се задава с помощта на таговете <FORM>...</FORM> и се състои от набор от полета за въвеждане, изобразявани от браузъра във вид на графични елементи за управление: селекторни бутони, опции, полета за въвеждане на текст, управляващи бутони и други.
- *Анализ и обработка на получената информация.* Данните, извлечени от HTML формата, се предават за обработка на CGI програмата. Те не винаги могат да бъдат обработени от CGI програмата самостоятелно. Например те могат да съдържат заявка към база от данни, която CGI програмата не може да чете. В този случай CGI програмата, на база на получената информация, формира заявка към компетентната програма, изпълнявана на същия компютър.
- *Създаване на нов HTML документ и препращането му на браузъра.* След обработката на получената информация CGI програмата създава динамичен, виртуален HTML документ, или формира препратка към вече съществуващ документ и предава резултата на браузъра.

Схема на кодиране

За всеки елемент на формата, имащ име, зададено с атрибута NAME, се фомира двойката "name=value", където value е стойността на елемента, въведена от потребителя или зададена по премълчаване.

Всички двойки се обединяват в стринг, в качеството на разделител служи символа "&". Кодираната информация се предава на сървъра с помощта на методите GET или POST. При използване на метода GET данните на формата се предават на сървъра в състава на URL заявката.

CGI скрипт

Предназначението на CGI програмите е да се създаде нов HTML документ, използвайки данните, съдържащи се в заявката и той да се върне обратно на клиента. Интерфейсът на CGI програмата не налага ограничения за избора на езика за програмиране. Последният трябва да притежава: *Средства за обработка на текст*. Необходими са за декодиране на входящата информация; *Средства за достъп до променливите на средата на обкръжението*. С тяхна помощ данните се предават на входа на CGI програмата; *Възможност за взаимодействие с други програми*. Необходима е за достъп до СУБД, програми за обработка на графика и други.

Механизми за обмен на данните

CGI спецификацията описва четири набора от механизми за обмен на данни: чрез променливите на обкръжението; чрез командния ред; чрез стандартния входящ поток; чрез стандартния изходящ поток.

Променливи на средата на обкръжението (мета променливи)

При стартиране на външна програма сървърът създава специфични променливи на обкръжението, чрез които предава на приложението както служебна информация, така и данни. Коректно е предварително да се помисли за обема на данните, предаван на скрипта и да се избере съответен метод за достъп. Размерът на променливите на обкръжението също е ограничен и ако е необходимо да се предават много данни е добре веднага да се избере метода POST, т.е. предаване на данни чрез стандартния входящ поток.

Формат на стандартния входящ поток

Стандартният входящ поток се използва за предаването на данни към скрипта с метода POST. Обемът на предаваните данни се задава от променливата на обкръжението CONTENT_LENGTH, а типът на данните - от промеливата CONTENT_TYPE. Например ако от HTML формата трябва да се предаде заявка от типа: a=b&b=c, тогава CONTENT_LENGTH = 7, CONTENT_TYPE = application/x-www-form-urlencoded, а първият символ в стандартния вход ще бъде символът "a". Следва да се помни, че краят на файла от сървъра към скрипта не се предава и затова четенето приключва според броя на прочетените символи.

Формат на стандартния изходящ поток

Стандартният изходящ поток се използва от скрипта за връщане на данните към сървъра. При това отговорът се състои от заглавие и данни. Резултатът от работата на скрипта може да се предава на клиента без преобразувания от страна на сървъра, ако скриптът осигурява построяване на пълно HTTP заглавие, в противен случай сървърът модифицира заглавието в съответствие с HTTP спецификацията. Заглавието на съобщението е длъжно да се отделя от тялото на съобщението с празен ред. Обикновено в скриптовете се задават само три полета на HTTP заглавието: Content-type, Location, Status. Content-type се задава само в случай, че скриптът сам генерира документ и го връща на клиента. В този случай реален документ във файловата система на сървъра не остава. При използването на такива скриптове следва да се отчита, че не всички сървъри и клиенти отработват така, както очаква разработчикът на скрипта. При задаване на Content-type: text/html, някои клиенти не реализират сканиране на получения текст за наличие в него на графика. Обикновено в Content-type се посочват текстови типове text/plain и text/html. Location се използва за преадресация. Понякога

преадресацията помага да се преодолее ограничението на сървъра или клиента за обработка на вградена графика или сървърна предварителна обработка. В този случай скриптът създава файл на диска и подава адреса му в Location. Така сървърът реално предава съществуващ файл. Полето Status позволява на CGI скрипта да върне статуса на обработката.

FastCGI интерфейсът е клиент-сървърен протокол за взаимодействие между уеб сървъра и приложенията, по-нататъшно развитие на технологията CGI. В сравнение с CGI е по-производителен и по-безопасен.

FastCGI сваля много от ограниченията на CGI програмите. Недостатъкът на CGI програмите е този, че те трябва да бъдат презаредени от уеб сървъра за всяка заявка, което води до понижаването на производителността. FastCGI, вместо да създава нови процеси за всяка една нова заявка, той използва постоянно заредени процеси за обработването на множеството от заявки. Това икономисва време.

Докато CGI програмите взаимодействат със сървара чрез STDIN и STDOUT на стартирания CGI процес, FastCGI процесите използват Unix Domain Sockets или TCP/IP за свързване със сървъра. Това дава следните предимства пред обикновените CGI програми: FastCGI програмите могат да бъдат стартирани не само на същия този сървър, но и където е необходимо в мрежата. Също така е възможна обработка на заявките от няколко паралелно работещи процеса.

FastCGI е отворен унифициран стандарт, разширяващ CGI интерфейса и позволяващ създаването на високопроизводителни уеб приложения без използването на специфичните API за уеб сървър. Целта на тази спецификация от гледна точка на FastCGI приложението е да се опише интерфейса на взаимодействието между него и уеб сървъра (който също така реализира FastCGI интерфейса). Спецификацията е написана за случая на използването на Unix платформата, макар че описанието й не зависи от порядъка на следването на байтовете и затова в общия случай може да се използва и на други платформи.

Интерфейсът FastCGI е разширение (подобрение) на CGI. FastCGI е проектирана за поддръжка на «дългоживущи» приложения - услуги. Това е главната разлика от традиционната реализация на CGI 1.1, където за обработката на всяка заявка се създава процес, еднократно използван, след което той приключва.

Стартовите условия за FastCGI процеса са «по-твърди» в сравнение с CGI, който при зареждане има достъп до променливите на обкръжението и може да използва стандартните потоци за вход-изход: STDIN, STDOUT, STDERR. Връзката между уеб сървъра и FastCGI процеса се реализира через единствен сокет. Този процес трябва да слуша за входящи съединения от уеб сървъра.

След приемането на съединението от уеб сървъра FastCGI процесът си обменя данни като използва прост протокол, решаващ две задачи: организация на двупосочния обмен в рамките на съединението (за емулация на STDIN, STDOUT, STDERR) и организация на няколко независими FastCGI сесии в рамките на едно съединение.

Internet Server API, API за IIS Microsoft (Internet Information Server)

ISAPI позволява на програмистите да разработват уеб приложения, които работят много по-бързо, отколкото CGI програмите, защото те са по-тясно интегрирани в уеб сървъра. Ако уеб сървърът е създаден на базата на Microsoft Internet Information Server, вместо CGI програми могат да се използват ISAPI приложения. Както CGI програмите, така ISAPI разширенията получават данни от браузъра, обработват ги и изпращат отговор на браузъра във вид на динамично генериран HTML документ.

ISAPI приложенията са DLL библиотеки, директно взаимодействащи с IIS API.

Интерфейсът ISAPI е предназначен за програмиране на приложения (API) и информационни услуги в Интернет (IIS). ISAPI се състои от класове за поддръжка и структури, участващи в програмната експлоатация на IIS. Уеб приложенията, използващи ISAPI за взаимодействие с IIS, реализират това взаимодействие на уеб сървъра на MSWindows по най-ефективния начин. Цялото програмно осигуряване на уеб приложенията на Microsoft пряко или косвено използва технологията ISAPI. Технологиите Microsoft Application Server Pages (ASP) и .NET Framework са построени като приложения на ISAPI.

Преобразуването на CGI уеб приложенията за използване на ISAPI увеличива производителността на уеб приложенията. ISAPI разширенията се зареждат в пространство на IIS процеса, затова възелът няма нужда да създава нов процес за всяка HTTP заявка. Понеже MSWindows зарежда динамично библиотеката в пространството на паметта еднократно при първото извикване на функцията в DLL и я съхранява там за неопределен интервал от време, затова ISAPI разширението остава заредено и не се изтрива докато IIS сървърът не се изключи или не се изтрие екземпляра от паметта.

Недостатък на тази технология е сложността за работа и компилация на ISAPI. Компилацията на кода в интегрираната среда за разработване (Integrated Design Environment, IDE) Visual Studio.NET е доста сложна и понеже IIS представлява процес с няколко нишки, резултатите от компилирането могат да бъдат непредсказуеми. Най-малката грешка в ISAPI приложението катастрофично повлиява на производителността на IIS. Сравнено с другите среди за разработване ISAPI е силно чувствителен към грешки в създаваното уеб приложение. ISAPI разширенията са най-честото използване на ISAPI. Филтрите са сложни за създаване и сферата на използването им е ограничена.

ISAPI разширението се създава във вид на DLL библиотека. Обръщението към тази библиотека се изпълнява в HTML документите, аналогично на обръщението към CGI програмите, от техните форми или препратки, създадени с таговете <FORM> и <A>. Когато потребителят се обръща към ISAPI разширението съответстващата DLL библиотека се зарежда в адресното пространство на Microsoft Information Server и става негова съставна част. Понеже ISAPI разширението работи в рамките на сървърния процес на Microsoft Information Server, а не в рамките на отделен процес, то може да ползва всички ресурси, достъпни на сървъра. Затова производителността на сървъра се запазва на високо ниво, даже ако разширението на сървъра се използва едновременно от много клиенти.

HTTP заявките се предават директно на ISAPI разширението с помощта на URL препратки или на данни, изпращани от HTML формата. ISAPI разширението може да се извика косвено посредством свързване на файл с определено ISAPI разширение в IIS. При установяването на свързване на файловете се изпълняват действия, аналогични на асоцииране на файлове за отговор на сървъра с конкретното ISAPI разширение.

ISAPI филтрите, както и ISAPI разширенията са реализирани във вид на DLL библиотеки. ISAPI филтрите са способни да контролират целия поток от данни между браузъра и уеб сървъра на нивото на HTTP протокола.

ISAPI филтрите (всеки в отделна *.dll) не могат да се изпълняват самостоятелно, подобно на ISAPI разширенията. Филтрите се зареждат при стартирането на IIS, според данните на метабазата. Филтрите работят в едно адресно пространство с IIS, което им осигурява висока скорост на работа. Те могат да се установят както за всеки IIS, така и за отделни негови сайтове. Възможността за анализ и при необходимост модификация на входящите и на изходящите потоци от данни на ISAPI филтрите ги прави изключително гъвкав и мощен механизъм. Но при лошо програмиране могат значително да намалят производителността на IIS. Филтрите на ISAPI се извикват независимо от другите заявки на IIS. Може да се настрои реагиране на ISAPI филтрите

на заявки според приоритети. Това ги отличава от останалите филтри, зареждани в IIS. Филтрите се използват от специализирани приложения, свързани с IIS, и обикновено изпълняват следните задачи: шифриране; журналиране (анализ и протоколиране на заявките според HTTP заглавията); пренасочване на заявките според HTTP заглавията; автентификация; компресия на данните.

Java Servlet

Сървлетът представлява Java клас, който се използва като разширение на възможностите на сървъри, предоставящи базирани на заявка-отговор програмен модел услуги. Сървлетът е Java интерфейс, реализацията на който разширява функционалните възможности на сървъра. Макар че сървлетите могат да обслужват всякакви заявки, обикновено те се използват за разширяване на уеб сървъри. Обикновено всеки сървлет е предназначен за извършване на една услуга. Един сървлет е една сървърна програма, която обслужва HTTP заявки и връща резултати като HTTP отговори. От тази гледна точка той много прилича на CGI (единствената прилика). Сървлетите имат директен достъп до голямо множество от JAVA API. Те могат да съхраняват статусна информация. Сървлетите наследяват всички достойнства на Java.

За да се разработват сървлети се използва JSDK (Java Servlet Development Kit). Той съдържа всички необходими класове и интерфейси.

Класовете и интерфейсите образуват два пакета:

- javax.servlet (GenericServlet class) - доставя базовия интерфейс;
- javax.servlet.http (HttpServlet class) - доставя класове, изведени от генетичния сървлетен интерфейс, които предлагат специфични средства за обслужване на HTTP заявки.

Когато един потребител зададе една заявка към един сървлет, тогава сървърът ще използва отделна нишка за обработката на заявката. Това има положително въздействие върху производителността, понеже отделните заявки не създават отделни процеси (както е при CGI). Нишките изискват по-малко ресурси. Друго предимство на сървлетите е тяхната преносимост. Сървлетите работят в процеса на уеб сървъра и затова могат да бъдат по-бързи от CGI реализациите.

В класа HttpServlet са определени методите doGet и doPost за реакция на заявките от тип GET и POST на клиента. Тези методи се извикват от метода service() на класа HttpServlet, който се извиква при пристигане на заявка на сървъра. Методът service() първо определя типа на заявката и после извиква съответния метод.

Методите doGet и doPost приемат в качеството на параметри обектите HttpServletRequest и HttpServletResponse, които осигуряват възможност за реализация на взаимодействието между клиента и сървъра. Методите на интерфейса HttpServletRequest осигуряват достъпа до данните в заявката. Методите на интерфейса HttpServletResponse осигуряват връщането на резултатите на уеб клиента в HTML формат. HttpServletRequest е обект, съдържащ информация за заявката от уеб браузъра, която може да бъде: login на потребителя и някакви стойности от заглавната част, изпратена от уеб браузъра; някакви параметри, изпратени като част от заявката; някакви информационни параметри, осигурени от deployment descriptor; да доставя сесии (session) и бисквитки (cookies). HttpServletResponse е обект, който позволява: добавяне на cookies; Header стойности; пренасочване към нови URL; създаване на HTTP за уеб браузър; изпращане на съобщения за грешки.

При имплементацията на сървлет от общ тип, може да се използва или да се разшири класа GenericServlet, а класа HTTPServlet предоставя методи за обслужване на HTTP заявки, като doGet и doPost. За да извърши необходимата обработка и да предостави отговор на заявката Java сървлетът използва различни уеб компоненти.

Тема 10. ASP.NET, PHP, JSP

Технология ASP.NET. Общи сведения

ASP.NET е технология за създаване на уеб приложения и уеб услуги на Microsoft. Тя е съставна част от платформата на Microsoft.NET и е развитие на по-старата технология Microsoft ASP. За първи път е публикувана през януари 2002 г. ASP.NET външно прилича на по-старата технология ASP, което позволява на разработчиците относително лесно да преминат към ASP.NET. В същото време вътрешното устройство на ASP.NET съществено се различава от ASP, понеже тя е основана на платформата .NET и следователно използва всички нови възможности, предоставяни от тази платформа. ASP.NET не е платформа.

ASP е първата технология на Microsoft, позволяваща динамично създаване на уеб страници на сървърната страна. ASP работи на операционните системи MSWindows NT и на уеб сървър Microsoft IIS. ASP не е език за програмиране, а е само технология за предварителна обработка, позволяваща свързване на програмни модули по време на процеса на формиране на уеб страницата. Относителната популярност на ASP е основана на лесното използване на скриптовите езици (VBScript или JScript) и възможността за използване на външни COM (Component Object Model) компоненти. Към настоящия момент технологията ASP (появила се през 1996 г.) се счита за остаряла и е заменена от ASP.NET. За ASP.NET е характерна възможността за разделяне на кода, описващ дизайна, от кода, реализиращ логиката на приложенията.

Теоретично при ASP.NET може да се пишат програми на всеки език (.NET език по избор), за който има съответстващ компилатор. Обаче на практика за създаването на ASP.NET приложения се използват основно Visual Basic.NET и C#.NET.

ASP и ASP.NET

Също като ASP, ASP.NET работи на уеб сървър и предоставя възможност за разработка на интерактивни, динамични, персонализирани уеб сайтове.

Има огромна разлика между ASP и ASP.NET. ASP.NET е част от средата за разработка .NET Framework. Приложения за ASP.NET могат да се пишат на всички езици за програмиране, които се компилират до Common Intermediate Language (CIL) код.

.NET Framework компилира кодовете на всички .NET езици (като Visual Basic .NET, Visual C++.NET, Visual C#.NET) в междинен CIL байт код (преди наричан MSIL Microsoft Intermediate Language, междинен език на Microsoft), използвайки CIL компилатор. Когато се компилира .NET код в CIL код, се получава независим от операционната система код. И вече при следващата стъпка от CIL код с JIT (Just-In-Time) компилатор .NET Framework преобразува CIL кода в машинен език, специфичен за конкретната операционна система и чак тогава .NET приложението може да бъде изпълнено на тази система.

Един от основните проблеми на ASP е смесването на HTML с бизнес логика, което прави страницата трудна за разбиране, поддръжка и дебъгване. Файлът става неимоверно голям и сложен, което забавя целия процес на разработване на приложението. Една от основните цели на ASP.NET е била да се справи с този проблем. Понеже потребителския интерфейс и бизнес логиката са две отделни неща ASP.NET ги разделя и се справя с всяко едно от тях поотделно. Благодарение на това разделение логиката за всяка форма (страница) става ясно отделена от клиентския интерфейс.

Програмирането за потребителския интерфейс се разделя на две различни части:

1. За визуализация се използва HTML код, записан във файл с разширение .aspx.
2. Бизнес логиката се дефинира в отделен файл (с разширение .cs за C#, с разширение .vb за VBScript), съдържащ конкретната имплементация на определен език.

Файлът, съдържащ бизнес логиката, се нарича изпълним код на уеб формата (Code Behind). Зад всяка уеб форма стои богатия обектен модел на .NET Framework. Всяка уеб форма се компилира до клас в асемблито (.DLL файлове) на проекта. Класът, генериран от .aspx файл, наследява един междинен клас вместо директно от Page класа. Междинният клас се нарича изпълним клас и там могат лесно да се добавят методи, да се обработват събития и т.н. Чрез изпълнимия код представянето е разделено от логиката, което прави .aspx страниците по-лесни за поддръжка.

ASP.NET има предимство в скоростта, сравнена със скриптовите технологии, понеже при първото обръщение кодът се компилира и се разполага в специален кеш, и впоследствие само се изпълнява (без загуба на време за парсинг, оптимизация и др.).

Архитектура на ASP.NET

Всяко ASP.NET приложение съдържа една или повече уеб форми. ASP.NET уеб формите абстрахират едно ниво над HTML. Генерираните страници са независими от операционната система и браузъра.

Основният компонент на ASP.NET е уеб формата - уеб страницата, която потребителят вижда в браузъра си под формата на HTML. Вместо целият HTML да се въвежда от програмиста, ASP.NET предлага едно по-високо ниво на абстракция, предлагайки богат избор от уеб контроли, подобни на тези при Windows формите.

Всяко ASP.NET приложение се изгражда от една или повече уеб форми, които могат да взаимодействат помежду си, като по този начин правят приложението интерактивно.

Уеб формата е програмируема уеб страница, която служи за потребителски интерфейс на ASP.NET приложение. Тя се състои се от HTML, код и контроли, които се изпълняват на уеб сървъра. Потребителят вижда резултата, като получава HTML, генериран от уеб сървъра. Кодът и контролите, които описват уеб формата, не напускат сървъра. ASP.NET е съвкупност от класове, които работят съвместно, за да обслужват HTTP заявки. Всичко е клас, зареден от асемблии. Работният процес на ASP.NET е отделен самостоятелен процес от IIS. Страниците в ASP.NET са винаги компилирани до .NET класове. Съдържат се в асемблита.

Традиционните уеб страници могат да изпълняват код на клиента, за да извършват по-прости операции, докато ASP.NET уеб формите могат също да изпълняват код от страна на сървъра, с който да осъществяват достъп до бази от данни, достъп до ресурсите на самия сървър, генериране на допълнителни уеб форми или да се възползват от вградената система за сигурност на сървъра.

Поради това, че ASP.NET уеб формите не разчитат на скрипт от страна на клиента, както и това, че всяка уеб форма в крайна сметка се превръща в чист HTML, те не са зависими от клиентския браузър или операционна система. Това позволява направата на една уеб форма, която да работи върху практически всяко устройство, което разполага с Интернет свързаност и уеб браузър, т.е. т.нар. Smart Client.

ASP.NET не е приспособен за малки проекти. Моделът на .NET класовете е сложен.

Предимства на ASP.NET пред ASP

- Компилируемият код се изпълнява по-бързо, повечето грешки се откриват още по време на разработването;
- Значително е подобрена обработката на грешките по време на изпълнението на стартираната готова програма с използването на блоковете try..catch;
- Потребителските елементи за управление (controls) позволяват да се отделят често използваните шаблони, такива като менюто на сайта;
- Използването на метафори, вече приложими в Windows приложенията, например такива като елементи за управление и събития;
- Разширяем набор от елементи за управление и библиотеки от класове позволява по-бързото разработване на приложенията;
- ASP.NET се опира на многоезиковите възможности на .NET, което позволява писането на кода на страниците на VB.NET, Delphi.NET, Visual C#, J# и т. н.;
- Възможност за кеширане на цялата страница или на нейни части за увеличаване на производителността;
- Възможност за кеширане на данните, използвани на страницата;
- Възможност за разделяне на визуалната част и бизнес логиката на различни файлове;
- Разширяем модел за обработка на заявките;
- Разширяем събитиен модел;
- Разширяем модел на сървърните елементи за управление;
- Наличие на master страници за задаване на шаблони за оформяне на страниците;
- Поддръжка на CRUD-операции при работа с таблици через GridView;
- Вградена поддръжка на AJAX;
- ASP.NET има предимство със скоростта в сравнение с другите технологии, основани на скриптове.

ASP.NET MVC

ASP.NET MVC е платформа, създадена от Microsoft, която служи за изработване на уеб приложения, използвайки модела Model-View-Controller (MVC). Платформата използва C#, HTML, CSS, JavaScript и бази от данни. ASP.NET MVC е съвременно средство за изграждане на уеб приложения, което не замества изцяло уеб формите. Платформата включва нови тенденции в разработката на уеб приложения, притежава много добър контрол върху HTML и дава възможност за създаване на всякакви приложения. ASP.NET MVC може да бъде лесно тествана и допълвана, защото е изградена от отделни модули, които са изцяло независими едни от други. Чрез платформата се създават цялостни приложения, които се стартират, а не единични скриптове (като при PHP например).

За пръв път ASP.NET MVC е била представена през октомври 2007 г. от Scott Guthrie на първата ALT.NET конференция в Остин, Тексас. Въпреки че Guthrie заявява в своя блог, че ASP.NET MVC ще бъде пусната в употреба като напълно поддържана функция на ASP.NET през първата половина на 2008, първата официална версия на продукта се появява на 13 март 2009 г.

Моделът представлява част от приложението, което реализира бизнес логиката, също известна като домейн логика. Домейн логиката обработва данните, които се предават между базата от данни и потребителския интерфейс. Например, в една система за инвентаризация, моделът отговаря за това дали елемент от склада е наличен. Моделът може да бъде част от заявлението, което актуализира базата от данни когато даден елемент е продаден или доставен в склада. Често пъти моделът съхранява и извлича официална информация в базата от данни.

Изглед модела позволява да се оформят няколко изгледа от един или повече модела от данни или източници в един обект. Този модел е оптимизиран за потребление и изпълнение.

Контролерите са класове, които се създават в MVC приложението. Намират се в папка Controllers. Всеки един клас, който е от този тип, трябва да има име, завършващо с наставка "Controller". Контролерите обработват постъпващите заявки, въведени от потребителя и изпълняват подходящата логика за изпълнение на приложението. Класът контролер е отговорен за следните етапи на обработка:

- Намиране и извикване на най-подходящия метод за действие (action method) и валидиране, че може да бъде извикан.
- Взимането на стойности, които да се използват като аргументи в метода за действие.
- Отстраняване на всички грешки, които могат да възникнат по време на изпълнението на метода за действие.
- Осигуряване на клас WebFormViewEngine по подразбиране за отваряне на страници с изглед от тип ASP.NET.

В ASP.NET приложения, които не ползват MVC Framework, взаимодействията с потребителя са организирани и контролирани чрез страници. За разлика от тях в приложения с MVC framework взаимодействията с потребителя са организирани чрез контролери и методи за действие. Контролерът определя метода за действие и може да включва толкова методи за действие, колкото са необходими.

Методите за действие обикновено имат функции, които са пряко свързани с взаимодействието с потребителя. Примери за взаимодействие с потребителите са въвеждане на URL адрес в браузъра, кликване върху линк и подаването на формуляр. Всяко едно от тези потребителски взаимодействия изпраща заявка към сървъра. Във всеки един от тези случаи, URL адресът от заявката съдържа информация, която MVC Framework използва за да включи метод за действие.

Предимството на MVC пред класическите уеб форми е в по-детайлния контрол върху генерирания html код. Размерът на страниците е значително по-малък, но времето за разработване на приложение с MVC е по-голямо. Уеб формите все още се използват, тъй като позволяват доста бързо да се разработи необходимата функционалност, но са приложими основно за Intranet приложения, докато MVC се предпочита за Интернет приложения.

PHP технология

PHP е замислен в края на 1994 г. от Rasmus Lerdorf. Ранните версии са били използвани на неговата лична страница с цел проследяване на това кой е преглеждал интерактивното му резюме. Първата използвана версия е станала достъпна в началото на 1995 г. и е била известна като Personal Home Page Tools. Тя се е състояла от много прост синтактичен анализатор, който е обработвал само няколко специални макрокоманди и редица програми, които впоследствие за били използвани за лични страници, като книги за гости, броячи и някои други допълнения.

Към 1996 г. PHP е бил използван поне от 15 000 уеб сайта по целия свят. През средата на 1997 г. тази цифра е подскочила на над 50 000. През средата на 1997 г. е променено разработването в PHP: синтактичният анализатор е бил отново пренаписан от Zeev Suraski и Andi Gutmans и този нов синтактичен анализатор станал основа за PHP версия 3.

PHP е рекурсивно съкращение на Hypertext Preprocessor. PHP е скрипт-машина, която комбинира множество готови уеб инструменти и интерпретатор на форми (Form

Interpreter) за генериране на бърз, лесен и с отворен код език за писане на скриптове. PHP работи практически на всеки уеб сървър с малки различия в кода. Най-голямото предимство на PHP е, че е безплатен с отворен изходен код. Той не изисква специален сървър както при ASP и ColdFusion. Той е по-бърз от JSP и е по-лесен за изучаване от Perl.

В момента PHP се използва от милиони домейни по целия свят и популярността му продължава да расте. С PHP може да създават и редактират файлове, да се събира и обработва информация от формуляри, да се изпращат данни с електронна поща, да се управляват записи в бази от данни, да се съхраняват данни в променливи по време на сесии и други. PHP предлага висока скорост на изпълнение и използва много малка част от системните ресурси и не забавя работата на хост машината.

PHP се прилага в неголеми проекти, по които работи неголяма група или сам човек. Той обикновено е изборът за по-малки авторски сайтове или за сайтове от малкия или средния бизнес. PHP работи по-бързо от ASP.NET. Скоростта се осигурява от това, че всички PHP приложения работят в едно адресно пространство, докато при ASP.NET за сметка на сложния модел от класове се налагат многократни проверки на данните, като всяко приложение се намира в отделно адресно пространство. Първият подход е по-бърз, но по-малко надежден, вторият е по-надежден, но за него се плаща.

Бързодействието на PHP + MySQL се осигурява от това, че разработващите ги групи тясно си сътрудничат. Същото се отнася до ASP.NET + MS SQL.

Петата версия на PHP предоставя възможности за обектно ориентирано програмиране, но не е получила голямо разпространение. В PHP 4.0 няма строга типизация, обявяването на променливите не е задължително и съществуват опасни конструкции. Синтаксисът на PHP е насочен към бързо и просто решаване на типичните проблеми, при което цялата отговорност е на програмиста. Често пъти откриването на сериозни грешки в кода става след месеци от приключване на проекта.

За подобряването на производителността в PHP 4.0 има няколко ключови нововъведения, такива като поддръжка на сесии, буферизация на изхода, по-безопасни методи на обработка на въвежданата от потребителя информация и няколко нови езикови конструкции. В петата версия на PHP измененията включват обновяване на ядрото на Zend (Zend Engine 2), което съществено е увеличило ефективността на интерпретатора. Въведена е поддръжка на езика XML. Напълно са преработени функциите на ООП, които започнали много да напомнят модела, използван в Java. В частност, е въведен деструктор, отворени, затворени и защитени членове и методи, окончателни членове и методи, интерфейси и клониране на обекти. Впоследствие във версиите са въведени пространство от имена и цял ред достатъчно сериозни изменения, количествено и качествено сравними с тези, които са се появили при прехода към PHP 5.0.

Съществуват множество модули (разширения) за PHP, които добавят различни функционалности и позволяват много по-бързо и ефективно разработване. Такива допълнителни функционалности към езика са: функции за обработка (създаване, редактиране) на изображения; функции за работа с низове и регулярни изрази; функции за работа с XML съдържание; функции за работа със сокети; функции за дата и час; математически функции; функции за управление на сесии и работа с бисквитки (cookies); функции за компресия и шифриране/дешифриране; функции за COM и .NET за MSWindows; функции за SOAP; функции за работа с различни бази от данни; функции за работа с принтер; функции за създаване на приложения с графичен потребителски интерфейс, базирани на библиотеката GTK; функции за изпращане на e-mail съобщения; хранилище за разширения и приложения на PHP: PEAR.

Java Server Pages

Технологията Java Server Pages (JSP) позволява смесването на обикновена статична HTML страница с динамично генерираното съдържимо на сървлетите. JSP и Servlet са средства, разработени от Sun Microsystems за създаване на уеб страници с динамично съдържание. Те се базират на езика Java, лесни са за написване и осигуряват пълен достъп до обектно ориентираните възможности на Java. Както и за всички приложения на Java, те са независими от платформата, на която се изпълняват.

JSP страниците предоставят такива възможности, каквито не могат да бъдат изпълнени от сървлетите. JSP документите автоматично се превеждат в сървлети. По-удобно е да се пише и модифицира обикновен HTML код, отколкото работа с множество от оператори println, които генерират HTML. Разделението на съдържимото и представянето позволява разделянето на задачите между различните уеб разработчици. Експертите по дизайн на уеб страници могат да създават HTML, използвайки обичайния инструментариум и оставяйки място за програмистите на сървлети, които да добавят впоследствие към страницата динамично съдържание.

JSP технологията позволява лесно създаване на уеб страници със статично и динамично съдържание. Уеб дизайн фирмите рядко използват подобен метод на работа, но използването на Java приложения често е наложително. JSP разширява динамичните възможности на Java Servlet технологията, базирана на Java, предоставяйки интуитивен начин за представяне на статична информация. В основата на JSP стои нуждата от разделяне на технологиите в едно приложение – именно имплементиране на MVC архитектура. Езикът за създаване на JSP страници, които представляват текстови документи, описващи как да бъде обработена заявката и как да бъде конструиран отговорът. Това включва и HTML, JavaScript, както и други езици за представяне на информация. Езикът за достъп до сървърни обекти UEL (Unified Expression Language) е изразителен език, който обединява използваният от предишните версии на JSP и JSF технологиите език за достъп до обектите.

Механизъм за добавяне на разширения към JSP езика

Предпочитаното разширение за JSP страници е .jsp. Една страница може да бъде съставена от няколко други страници или фрагменти. Разширението за JSP фрагмент е .jspf.

JSP елементите на JSP страница могат да бъдат описани с два синтаксиса - стандартен JSP и XML, като една страница може да съдържа само един тип синтаксис. JSP страница, използваща XML се нарича JSP документ и е XML валиден документ. Като такъв може да бъде обработвана със средства за работа с XML.

JSP файловете се преобразуват в сървлети. Всяка уеб страница може да бъде генерирана в резултат на изпълнение на обикновена Java програма (Java сървлет), но е по-удобно и бързо да се пише Java скрипт в HTML, който добавя динамични елементи. JSP се използва като алтернатива на сървлетите и в комбинация със сървлетите. Това опростява аспекта на сървлет програмирането или това е писане на html към поток. Позволява се да се смесва Java код директно с html, като резултатът се записва към потока автоматично.

JSP страниците обработват заявките като сървлетите. Затова жизненият им цикъл и много от възможностите на JSP (основно динамичните) са свързани с Java Servlet технологията. Когато към JSP страница е изпратена заявка, контейнерът проверява дали JSP сървлета на страницата е актуален. Ако не е, тогава страницата се превежда в сървлет клас, който се компилира. Резултатите на JSP страниците винаги се буферират автоматично.

Тема 11. CORBA

Понастоящем съществуват няколко модела за разпределени компоненти, като основните могат да се представят по следния начин:

- CORBA на Object Management Group.
- JEE / JavaBeans на фирмата Sun Microsystems.
- COM+ / .NET на фирмата Microsoft.

Тези три технологии са решение от областта Enterprise Integration Application (EIA).

Целта е:

- интеграция на услугите за приложенията на базата на различни платформи;
- осигуряване на взаимодействие: прозрачен механизъм за общуване и обмен на данни между елементите на разпределените изчислителни системи.

Утвърдилият се подход към интеграция на приложенията се изразява в описанието на външните интерфейси, форматите на съобщенията и спецификациите, използвани за предаването на тези съобщения. В процеса на интегрирането в съвременното разбиране на тази задача е въвлечено понятието Quality-of-Service, т.е. всичко, което има връзка с безопасността, надеждността и транзакционната цялостност. Но това е твърде малка част. Интеграцията на програмните приложения, както и самите приложения, реализират потребностите на бизнеса, затова е необходимо привличането в интеграционния модел на елементи за описание на бизнес процеси.

Интеграцията предполага добра воля и потребност за взаимодействие. Едностранната интеграция е невъзможна, неефективна и “незаконна”. Приложенията, както и техните създатели и потребителите трябва да се придържат към определени норми. Целта на стандартите е получаването на изгода от предоставените възможности за интеграция.

CORBA OMG

В края на 90-те години въпросите за мащабирането са станали толкова популярни, че се е изисквала стандартизация. CORBA и разпределените мрежови програмни приложения са “израствали” заедно. Първи OMG (Object Management Group) в края на 90-те години стартират с няколко спецификации, формализиращи понятието бизнес обект, средства за управление на бизнес обекти и среди за функционирането им (Business Object Facility (BOF) и Business Object Component Architecture (BOCA)). Стандартите за моделиране на OMG (UML, MOF, MDA) също така позволяват свързването на бизнес модела на разпределената система с техническите средства на CORBA и с нейния компонентен модел. Стандартът CORBA е първата разработена архитектура за корпоративни системи и Интернет приложения.

CORBA е механизъм на програмното осигуряване за осъществяване на интеграция на изолирани системи, който дава възможност на програмите, написани на различни езици за програмиране, работещи в различни възли на мрежата, да взаимодействат една с друга почти толкова лесно, както ако те се намират в адресното пространство на един процес.

CORBA означава Common Object Request Broker Architecture (обща архитектура на брокер за обектни заявки). Задачата на CORBA е да се направи възможна интеграцията на изолираните системи. Клиентите, използващи CORBA, получават предимство в три направления: прозрачност на извикването (invocation transparency), прозрачност на реализацията (implementation transparency) и прозрачност на локализацията (location transparency).

Прозрачността е основната цел на CORBA. За решаването на тази задача CORBA използва всички най-съществени постижения на обектната технология: инкапсулация, наследяване, полиморфизъм, динамично свързване.

Прозрачността на извикването определя гледната точка на клиента, изпращащ съобщение към сървъра. Езикът, използван при реализацията на клиента, определя как извикваният обект получава съобщението, как параметрите се предават в стека, как се реализира обръщението към методите на дадения обект, как се предават връщаните стойности. Ако сървърът е създаден в съответствие с правилата на CORBA, тогава клиентът може да извиква методите на този, съвместим с CORBA обект, така както и за всеки друг обект, реализиран с използвания език за програмиране. CORBA поддържа редица езици като C, C++, Java, COBOL, Ada, Lisp, Smalltalk, PL/1, Python, IDLscript.

Прозрачността на реализацията това е инкапсулация, приложена към разпределените системи. Клиентът за извикването на метода знае три неща: името на метода, параметрите на метода (ако има такива) и типа на връщаната стойност. Клиентът може да предостави на сървъра информация относно своите ограничения до тогава, когато се извикат методите. Клиентът извиква метода със средствата на езика, използван при разработването на клиента, а кодът, извикван за изпълнението на метода, се реализира на езика, който се поддържа от преобразуващите езици в OMG.

Прозрачността на локализацията позволява на клиента да извиква CORBA съвместим код, който може да се изпълнява на всеки възел. Прозрачността на локализацията се базира на свойството прозрачност на реализацията – клиентите като че ли правят локални извиквания, но на практика те извикват код, написан на различни езици и изпълняван зад пределите на адресното пространство на процеса на клиента. Всичко това е възможно благодарение на концепциите на обектната технология (инкапсулация, наследяване и полиморфизъм), които се използват по независим от реализацията начин.

CORBA е продукт на OMG, консорциум от повече от 800 фирми с цел дефинирането и популяризирането на индустриален стандарт за обектни технологии. CORBA представлява спецификация за създаване и използване на разпределени обекти. Прилагането на спецификацията представлява ORB (Object Request Broker). ORB представлява разпределена услуга, която прилага заявките към отдалечените обекти. Брокерът локализира отдалечените обекти в мрежата, предава заявката на обекта, изчаква резултата и го предава обратно към клиента. CORBA използва специален език наречен Interface Definition Language (IDL) за да дефинира интерфейсите между различните брокери. Езикът по синтаксис прилича на C++, но не притежава пълната функционалност на програмен език. За достъп до софтуерните обекти всеки ORB осигурява преобразуване (mapping) от IDL до съответния език.

Важна част от стандарта CORBA е дефинирането на множество от разпределени услуги за поддръжка на интегрирането и на взаимодействието между разпределените обекти. Услугите, известни като CORBA Sevices или COS, са дефинирани като стандартни CORBA обекти с IDL интерфейси и понякога се наричат Object Services.

Технологията CORBA значително е разширила понятието обектно-ориентирани системи. Най-важно е било понятието за прозрачност на местоположението, с което се отделя обекта не само от физическото му месторазположение в мрежата, но и от самата мрежа. С помощта на CORBA обектите могат да се извикват така както ако се намират на същата машина, на която се намира и извикващите ги клиентски приложения. Каскадните извиквания позволяват построяване на приложения от компоненти, като от строителни блокчета. С помощта на реализацията (instantiations) могат да се създават и разрушават обекти според необходимостта. Интерфейсите, описани на разработеният

от OMG език за описание на интерфейси (IDL), не зависят от езиците за програмиране на програмни приложения, които могат да бъдат и не обектно ориентирани.

Значителното достоинство на CORBA в съвременните мрежови среди се състои в това, че детайлите на мрежовото взаимодействие почти напълно са скрити от програмиста.

Спецификацията CORBA дефинира понятието Object Adapter (OA). Той представлява рамка (framework) за създаването на CORBA обекти и предлага API, използвано от обектите за множество услуги на ниско ниво. Съгласно спецификацията OA е отговорен за следните дейности:

- Генериране и интерпретиране на стъбовете;
- Извикването на методите;
- Сигурността на взаимодействието;
- Активирането и деактивирането на отдалечените обекти;
- Връзката между стъбовете и отдалечените обекти;
- Регистрирането на отдалечените обекти.

Обектен модел на CORBA

Обектният модел на CORBA определя взаимодействието между клиентите и обектите на сървъра. Интерфейсите на обектите описват множество от операции (методи), които клиентите могат да извикват за този обект. Реализацията на обекта се осигурява от приложенията, реално изпълняващи действията, заявени от клиента при извикването на операцията (метода).

Базови архитектурни компоненти на CORBA

1. Брокер на обектните заявки (Object Request Broker, ORB);
2. Език за определяне на интерфейсите - Interface Definition Language (OMG IDL);
3. Хранилище на интерфейсите (Interface Repository);
4. Преобразуващи езици (Language mappings);
5. Стъбове и скелетони (Stubs and skeletons);
6. Средства за динамично извикване и за диспетчеризация (Dynamic Invocation Interface и Dynamic Skeleton Interface);
7. Обектни адаптери (Object Adapter - OA);
8. Протоколи за взаимодействие на ORB (Inter-ORB Protocols).

Свойства на дадената технология, осигуряващи приспособяването ѝ за работа в неоднородна среда:

1. IDL позволява описването на интерфейсите на обектите независимо от езика, на който самите обекти ще бъдат реализирани;
2. Мрежовите протоколи, използвани от CORBA, скриват от разработчиците ниските нива на детайлизация на мрежовото взаимодействие;
3. На разработчика не се налага да разполага с информация за физическото разположение на сървъра и механизма за неговата активизация;
4. Разработването на клиентските приложения не зависи от операционната система и от апаратната платформа на сървъра.

Брокер на обектните заявки (ORB)

ORB определя механизма на прозрачно предаване на клиентските заявки за реализация на целевите обекти. Той изолира клиентите от детайлите по извикването на

методите, като така за клиента заявките изглеждат като локални извиквания на процедури.

Когато клиентът извиква метод, ORB отговаря за търсенето на реализацията на обекта, прозрачно за клиента е неговата активизация, предаването на данните на заявката на обекта и връщането на резултатите обратно на клиента. ORB гарантира, че интерфейсът, чрез който клиентът може да извиква методите на обекта, не зависи от разположението на обекта и от езика, на който той е написан.

За да направи заявка, клиентът първо получава указател към обекта (object reference) от ORB.

ORB е логическа същност, която може да бъде реализирана с различни механизми (с един или няколко процеси или набор от библиотеки). За да се отдели приложението от детайлите на реализацията, стандартът CORBA определя абстрактен интерфейс за ORB. Този интерфейс предоставя различни спомогателни функции, като преобразуване на указателите на обектите в низ и др.

OMG IDL

Това е стандартно средство за описание на интерфейси. Синтаксисът прилича на C++, но включва само декларативна часта (несъвместим е с COM IDL). Обработва се от компилатора IDL с генерация на стъбове и скелетони.

Хранилище на интерфейса (Interface Repository)

Хранилището на интерфейса е предназначено за приложения, на които е необходимо динамично да получават информация за регистрираните в системата обекти. В него се съдържат метаданни за достъпните интерфейси като набор от обекти с описание на техните операции, параметрите и изключенията.

Хранилището на интерфейса дава възможност за построяване на приложения в които данните, параметрите и функциите могат да се променят по време на изпълнение на програмата (CASE-средства, браузъри и т.н.).

Преобразуващи езици (Language mappings)

Преобразуващите езици задават как конструкциите на IDL да се изобразяват върху конструкциите на конкретния език за програмиране. Те задават изобразяването на интерфейсите:

1. представянето на системните обекти, такива като самият ORB;
2. описанието на това, как се реализират CORBA обектите на дадения език.

Стъб и скелетон IDL

Стъбът е механизъм, който ефективно създава и изпълнява заявки от името на клиента, а скелетонът доставя заявката до реализацията на сървърния обект. Кодът на стъба и на скелетона директно се генерира от IDL компилатора от кода на IDL и се компилира заедно с приложението-клиент и сървърния обект, затова извикването посредством стъб се нарича още статично извикване. При тях предварително е налична цялата информация за интерфейсите на обекта.

Dynamic Invocation Interface (DII) и Dynamic Skeleton Interface (DSI)

DII позволява на клиентите да формулират произволни заявки към обектите, чиито интерфейси за клиента на етапа на компилацията са неизвестни, т.е. цялата информация е необходимо да се получи по време на изпълнението. Това е аналог на IDispatch в COM. DSI е аналог на DII за сървърната част и той позволява

разработването на сървърните обекти без компилацията с тях на кода на скелетон. Те се използват и реализират значително по-рядко.

Обектни адаптери

Основната функция на обектния адаптер е да се свърже реализацията на обекта с ORB. За тази цел той трябва да осигурява:

1. регистрация на обектите – трябва да поддържа операции, които позволяват на компонентите от един или друг език за програмиране да бъдат регистрирани като реализация на CORBA обекти;
2. генерация на обектните указатели за CORBA обектите;
3. активизация на сървърните процеси, в които могат да бъдат активирани обектите;
4. активизация и деактивизация на обектите;
5. извикване на необходимите методи на обектите.

Има два вида обектни адаптери: остарелият базов обектен адаптер (BOA) и съвременната реализация - преносим обектен адаптер (POA).

Протоколи за свързване между ORB (GIOP, IIOP)

Основната задача на спецификацията CORBA е осигуряването на взаимна връзка между обектите, които са разпределени в нееднородна мрежа и се намират под управлението на различни реализации на ORB. За тази цел е създаден специален стандарт за разработване на протоколи за обмен на данни между брокерите - General Inter-ORB Protocol (GIOP).

GIOP е общ стандарт. Конкретната му реализация, основана на предаването на данни над протоколния стек TCP/IP, се нарича IIOP (Internet Inter-ORB Protocol). IIOP е конкретен протокол, като реализацията му позволява осигуряване на предаването на данните. Всичко, което му е необходимо е поддръжката на TCP/IP в конкретната мрежа.

Ако два ORB на различни производители с различна архитектура поддържат IIOP, то това означава, че те могат да си обменят информация и че клиентите, реализирани с използването на единия от тях, могат да се обръщат към методите на сървърните обекти, изпълнявани под управлението на другия при условие, че стъбовете на клиента и скелетоните на сървъра са генерирани от един и същ код на IDL (възможно е от два различни компилатора за тяхното преобразуване към езика).

Услуги на CORBA (CORBA Services)

Услугите на CORBA са набор от универсални интерфейси, които се използват от много разпределени обектни приложения. Като пример може да се вземе услуга за търсене на други достъпни услуги, в частност:

1. Услуга за имена (Naming Service), която позволява на клиентите да откриват обектите на база на техните имена;
2. Бизнес услуга (Trading Service), която позволява откриването на обектите на база на зададени свойства.

Други важни услуги са: услугите за управление на жизнения цикъл на обектите, услуга за безопасност, услуга за транзакциите, за оповестяване за събития и т.н.

Конкурентите на CORBA са Java и .NET. Родителите на конкурентите също са били членове на OMG. Sun Microsystems Inc. е един от основателите на CORBA OMG, а Microsoft се е присъединил към OMG през 1997 година.

Към настоящия момент CORBA активно си взаимодейства с другите технологии, като на първо място са RMI и Enterprise JavaBeans.

Тема 12. JAVA EE

Най-развитите платформи за разпределени приложения са Microsoft .NET и Java Enterprise Edition (JEE). Последната се базира на отворените стандарти и не зависи от апаратната платформа и от операционната среда. Основен производител на средствата на технологията .NET е Microsoft. По отношение на средствата на технологията JEE, поради отвореността на нейните стандарти, тук спектърът от производители е широк. Лидерите са Oracle, IBM, Sun, Red Hat, като много от средствата на тази технология се разпространяват свободно или имат отворен код.

Java Platform, Enterprise Edition

Технологията Java EE е формално обявена през декември на 1999 година. Тя представлява първият стандарт за създаване на корпоративни разпределени многоелементни приложения. Въпреки първенството си, JEE се е получила изключително хармонична, удобна и полезна. Приела е всички основни стандарти на CORBA и XML. JEE позволява съществено да се опрости труда на системните архитектори, проектанти и разработчици, като предлага ясна и гъвкава архитектура и набор от взаимно свързани стандарти за използване на най-важните системни услуги. JEE обединява такива стандарти, като Enterprise JavaBeans компонентния модел, стандартите на уеб приложенията за формиране на динамични отговори на потребителските действия - Java Servlets и JavaServer Pages, стандарта за достъп до бази от данни JDBC.

Всички компоненти на JEE имат едно общо свойство: трябва да са написани на езика Java. Изключение са само клиентските приложения. Стандартът разрешава на CORBA клиентите да взаимодействат със сървърните компоненти на JEE.

JAVA EE е набор от спецификации и съответна документация за езика Java, описващи архитектурата на сървърната платформа, решаваща задачи на средни и крупни организации. JAVA EE е основно ориентирана за използване чрез уеб, както в Интернет, така и в локални мрежи.

Платформата JAVA EE предлага компонентен подход за проектиране, разработване и внедряване на корпоративни приложения. Основен елемент на JAVA EE е компонентният модел Enterprise JavaBeans. Enterprise Java Beans (EJB) е сървърен компонент, съдържащ бизнес логиката на приложението. Програмните клиенти реализират бизнес логиката с извикване на EJB методите. Този подход освобождава разработчика на клиентското програмно осигуряване от операциите със системата на нивото на приложенията и позволява на разработчика на JavaBeans да се концентрира върху логиката на приложението.

Архитектурата на JAVA EE съдържа следните компоненти: JAVA EE сървър, EJB контейнер и уеб контейнер. Те реализират средния слой в многослойните Java приложения. JAVA EE сървърът предоставя EJB и уеб контейнери. При разработване на уеб приложения се използва уеб контейнер, а при разработване на стандартно клиентско приложение се използва EJB контейнер. Клиентът комуникира с бизнес слоя на JAVA EE или директно при стандартни приложения или посредством JSP или сървлети при уеб приложения.

EJB бизнес компонентите са сървърни компоненти, които се изпълняват на JAVA EE сървър. Те са компоненти на средния слой, съдържащи методи, които реализират бизнес правила. Чрез тях се реализира бизнес логиката на програмното приложение. Бизнес компонентите управляват преобразуването на данните и информационния поток между клиента (или аплети) и JAVA EE сървъра или между сървъра и базата от данни. EJB контейнерът управлява бизнес компонентите, които се

използват в JAVA EE приложенията. В тези компоненти се съдържа само бизнес логиката на приложението. Не е необходимо да се пише код за системни услуги, например за управление на транзакции или за сигурност на приложението. EJB компонентите предоставят такива услуги и могат да се настройват по време на инсталиране на сървъра. В рамките на приложенията, създадени с JEE технологията, връзката с базата от данни и бизнес логиката, която е скрита от потребителя, е прието да се реализира с помощта на Enterprise JavaBeans компонентите.

Концепцията на JavaBeans

JavaBeans са класове на езика Java, написани по определени правила. Те се използват за обединяване на няколко обекта в един (bean) за удобно предаване на данните. JavaBeans осигурява основата за многократно използваните вграждани и модулни компоненти на програмното осигуряване.

Спецификацията на Sun Microsystems определя JavaBeans, като «универсални програмни компоненти, които могат да се управляват с помощта на графичния интерфейс».

Компонентите на JavaBeans могат да приемат различни форми, но най-широко те се използват в елементите на графичния потребителски интерфейс. Една от целите на създаването на JavaBeans е взаимодействието с подобни компонентни структури. Например Windows програма, при наличие на обект-обвивка или съответстващ мост, може да използва JavaBeans компонент така, както ако той се явява COM компонент или ActiveX компонент.

Правила за описание на JavaBean

За да може класът да работи като bean, той трябва да съответства на определени споразумения за имената на методите, конструктора и поведението. Тези споразумения дават възможност за създаване на инструменти, които да могат да използват, заместват и съединяват JavaBeans.

Правилата за описание са следните:

- Класът трябва да има public конструктор без параметри. Такъв конструктор позволява на инструментите да създават обект без допълнителни сложности с параметрите.
- Свойствата на класа трябва да са достъпни чрез get, set, is и други методи за достъп, които се подчиняват на стандартното споразумение за имената. Това лесно позволява на инструментите автоматично да определят и обновяват съдържанието на bean. Много инструменти имат специализирани редактори за различните типове свойства.
- Класът трябва да е сериализуем. Това позволява надеждно съхраняване, запазване и възстановяване на състоянието на bean по независим от платформата и от виртуалната машина начин.
- Класът не е длъжен да съдържа методи за обработване на събитията.

Enterprise JavaBeans

Enterprise JavaBeans е повече от само инфраструктура. Нейното използване предполага още и технология за създаване на разпределени приложения, задава определена архитектура на приложението, а също така и определя стандартните роли на участниците в разработката.

Прилагането на дадените техники осигурява решаването на следните стандартни проблеми на мащабируемите и ефективни сървъри за приложения с използването на езика Java:

- организация на отдалечени извиквания между обектите, работещи под управлението на различни Java виртуални машини;
- управление на потоците на станата на сървъра;
- управление на жизнения цикъл на сървърните обекти (създаване, взаимодействие с потребителя, унищожаване);
- оптимизация на използването на ресурсите (процесорно време, памет, мрежови ресурси);
- създаване на схеми за взаимодействие на контейнерите и операционните среди;
- създаване на схеми за взаимодействие на контейнерите и клиентите, в това число и на универсални средства за разработването на компонентите и включването им в състава на контейнерите;
- създаване на средства за администриране и осигуряване на взаимодействието им със съществуващите системи;
- създаване на универсална система за търсене от клиента на необходимите сървърни компоненти;
- осигуряване на универсална схема за управление на транзакциите;
- осигуряване на необходимите права за достъп до сървърните компоненти;
- осигуряване на универсално взаимодействие със СУБД.

Enterprise JavaBeans технологията определя набор от универсални и предназначени за многократно използване компоненти, които се наричат Enterprise beans (EJB компоненти). При създаването на разпределена система, нейната бизнес-логика се реализира от такива компоненти.

Всеки EJB компонент се състои от:

- отдалечен интерфейса (remote-интерфейс), който определя бизнес-методите, които може да извиква EJB клиента;
- собствен интерфейса (home-интерфейс), който предоставя методите:
  - create за създаването на нови екземпляри на EJB компоненти;
  - търсене (finder) за намиране на екземпляри на EJB компоненти;
  - remove за премахване на екземпляри на EJB.
- реализации на EJB компонента, които определят бизнес-методите, обявени в отдалечения интерфейс, и методите за създаване, премахване и търсене на собствен интерфейс.

EJB контейнерите се изпълняват под управлението на EJB сървъра, който представлява свързващо звено между контейнерите и използваната операционна среда. EJB сървърът осигурява достъп на EJB контейнерите до системните услуги, такива като управление на достъпа до базите от данни или мониторинг на транзакциите. Всички екземпляри на EJB компонентите се изпълняват под управлението на EJB контейнера, който им предоставя разположените в него компоненти и управлява техния жизнен цикъл.

След приключването на разработването, наборите от EJB компоненти се разполагат в специални файлове (архиви, jar), за един или повече компоненти, заедно със специални deployment параметри. След което те се установяват в специална операционна среда, в която се зарежда EJB контейнера.

Клиентът търси компоненти в контейнера с помощта на home-интерфейса на съответстващия компонент.

След като компонентът е създаден и/или намерен, клиентът изпълнява обръщение към неговите методи с помощта на remote-интерфейса.

В общия случай контейнерът е предназначен за решаването на следните задачи:

- *Осигуряване на безопасност*. Дескрипторът за разгръщанията (deployment descriptor) определя правата за достъп на клиентите до бизнес-методите на компонентите. Защитата на данните се осигурява чрез предоставяне на достъп само на оторизирани клиенти и само до разрешени методи.
- *Осигуряване на отдалечени извиквания*. Контейнерът поема всички въпроси от ниско ниво за осигуряване на взаимодействието и организацията на отдалечените извиквания, като скрива напълно всички детайли на процесите, както от разработчика на компонентите, така и от клиентите. Това позволява да се разработват компонентите точно така, както ако системата ще работи в локална конфигурация, т.е. без използване на отдалечени извиквания.
- *Управление на жизнения цикъл*. Клиентът създава и унищожава екземпляри на компонентите, обаче контейнерът с цел оптимизация на ресурсите и повишаване на производителността на системата може самостоятелно да изпълнява различни действия, като например активация и деактивация на тези компоненти, създаване на пулове и т.н.
- *Управление на транзакциите*. Всички параметри, необходими за управлението на транзакциите, се разполагат в дескриптор на доставката. Всички въпроси, свързани с осигуряването на управлението на разпределените транзакции в хетерогенни среди и взаимодействието с няколко бази от данни поема EJB контейнера. Контейнерът осигурява защита на данните и гарантира успешното потвърждаване на внесените изменения, в противен случай транзакцията не се изпълнява.

Типове EJB компоненти

Съществуват три типа EJB компоненти:

- сесийни (Session Beans);
- същностни (Entity Beans);
- управляеми от съобщения (Message Driven Beans).

Сесийният компонент представлява обект, създаден за обслужване на заявките на един клиент. В отговор на отдалечената заявка на клиента контейнерът създава екземпляр на такъв компонент. Сесийният компонент винаги е съпоставен с един клиент и той може да се разглежда като «представител» на клиента на страната на EJB сървъра.

Сесийните компоненти са временни обекти. Обикновено сесийният компонент съществува, докато създалият го клиент поддържа с него сеанс за връзка. След приключването на връзката с клиента компонентът вече няма как да се съпостави с него.

Сесийните компоненти са от три типа:

- stateless (без състояние);
- stateful (с поддръжка на текущо състояние на сесията);
- singleton (един обект за всички приложения).

Същностните компоненти представляват обектно представяне на данните от базите от данни. Основната разлика на същностния компонент от сесийния е тази, че няколко клиента могат едновременно да се обръщат към един екземпляр на същностния компонент. Същностните компоненти изменят състоянието на съпоставените с тях бази от данни в контекста на транзакциите.

Състоянието на компонентите-същности в общия случай е необходимо да се съхранява, и те живеят толкова, колкото съществуват в базата от данни тези данни,

които те представят, а не толкова, колкото съществува клиентският или сървърният процес. Спирането или крахът на EJB контейнера не води до унищожаването на съдържащите се в него същностни компоненти.

Управляемите от съобщения компоненти се характеризират с това, че тяхната логика представлява реакция на събитията в системата.

Съставни части на EJB компонент

EJB компонентът физически се състои от няколко части, включително и самия компонент, реализацията на някои интерфейси и информационен файл.

Всичко това се събира заедно в специален jar-файл – модул за разгръщане.

- Enterprise Bean представлява Java-клас, разработен от доставчика на Enterprise Bean. Той реализира Enterprise Bean интерфейса и осигурява реализацията на бизнес-методите, които изпълнява компонента. Класът не реализира никакви методи за оторизация, многопоточности или поддръжка на транзакции.
- Домашен интерфейс. Всеки създаващ се Enterprise Bean трябва да има асоцииран домашен интерфейс. Домашният интерфейс се използва като фабрика за EJB компонента. Клиентът използва домашния интерфейс за намиране на екземпляра на EJB компонента или за създаване на нов екземпляр на EJB компонента.
- Отдалеченият интерфейс е Java-интерфейс, който отразява чрез рефлексия тези методи на Enterprise Bean, които е необходимо да се показват на външния свят. Отдалеченият интерфейс играе същата роля, като IDL-интерфейса в CORBA, и осигурява възможност за обръщение на клиента към компонента.
- Описателят на разгръщането представлява XML файл, който съдържа информация относно EJB компонента. Използването на XML позволява лесно да се променят атрибутите на компонента. Конфигурационните атрибути, определени в описателя на разгръщането, включват:

  - имената на домашния и отдалечения интерфейс;
  - Java Naming and Directory Interface (JNDI име за публикуване на домашния интерфейс на компонента);
  - транзакционни атрибути за всеки метод на компонента;
  - контролен списък за достъп за оторизация.
- EJB-Jar файл – това е обикновен java-jar файл, който съдържа EJB компонент (компоненти), домашния и отдалечения интерфейси, а също така и описател на разгръщането.

Enterprise JavaBean инфраструктура

EJB инфраструктурата осигурява отдалечено взаимодействие на обектите, управление на транзакциите и безопасност на приложението. EJB спецификацията определя изискванията към елементите на инфраструктурата и определя Java API, като тя не засяга въпросите за избора на платформата, протоколите и други аспекти, свързани с реализацията.

В общия случай е необходимо да се гарантира съхраняването на състоянията на компонентите в контейнерите. EJB инфраструктурата е длъжна да предостави възможности за интеграция на приложението със съществуващите системи и приложения. Всички аспекти на взаимодействието на клиентите със сървърните компоненти трябва да са реализирани в контекста на транзакциите, управлението на които се възлага на EJB инфраструктурата.

Enterprise JavaBeans спецификацията е съществена крачка към стандартизацията на модела на разпределените обекти в Java.

Тема 13. Microsoft .NET Framework

.NET Framework е програмна платформа, създадена от компанията Microsoft през 2002 година. Счита се, че платформата .NET Framework представлява отговорът на Microsoft на получилата по това време огромна популярност платформа Java на компанията Sun Microsystems (сега принадлежаща на Oracle).

За операционните системи на MSWindows с разработването на платформата се занимава нейният създател, компанията Microsoft. Съществуват също така независими реализации, като Mono и Portable.NET, позволяващи използването на .Net програми от други операционни системи, като например GNU Linux.

Технология .NET е официално призната, което е отразено в съответните стандарти ECMA (European Computer Manufacturers Association).

Средата .NET е предназначена за по-широко използване от това на платформата JEE. Но нейната функционалност в частта, предназначена за разработването на разпределени уеб приложения, много прилича на JEE.

Универсалният интерфейс на .NET Framework осигурява интегрирано проектиране и реализация на компонентите на приложенията, които са разработени с използването на различни подходи в програмирането.

В рамките на .NET има аналози на основните видове компоненти на JEE. На уеб компонентите съответстват компонентите, построени по технологията ASP.NET, а на EJB компонентите, свързващи приложението със СУБД, - компонентите ADO.NET. Компонентната среда на .NET обикновено се разглежда като еднородна. Но съществуващите неголеми разлики в правилата, управляващи създаването и работата с компонентите на ASP.NET и с останалите, които позволяват да се предположи, че в рамките на .NET присъства аналог на уеб контейнера, отделна компонентна среда за ASP.NET, и отделна - за останалите видове компоненти. Тези среди са тясно свързани и подобно на контейнерите на JEE, позволяват взаимодействието на компонентите, разположени в различни среди. Компонентната среда за ASP.NET, за разлика от уеб контейнера в JEE, поддържа автоматични разпределени транзакции.

Версиите на ASP.NET са тясно свързани с версиите на .NET Framework.

Всички аспекти на идеологията на .NET са реализирани в посока на гъвкава интеграция с програмно-апаратните ресурси, безопасност и удобство при използването на кода, а също така и намаляване на разходите за създаване на програмното осигуряване.

Microsoft предлага новаторски компонентно ориентиран подход за програмиране, който представлява развитие на обектно ориентираното направление. Според този подход интеграцията на обектите се реализира на база на интерфейсите, представящи тези обекти като независими компоненти. Такъв подход значително опростява имплементацията на взаимодействието н програмните „молекули” - компоненти в хетерогенна среда за проектиране и реализация. Стандартизира се съхраняването и повторното използване на компонентите на програмния проект в условията на разпределена мрежова среда за изчисления.

Съществено предимство на конструктивните решения на .NET е компонентно ориентирания подход при проектирането и реализацията на програмното осигуряване. Същността на подхода се състои във възможността за създаване на независими модули на програмното осигуряване с унифицирана интерфейсна част за многократно повторно и разпределено използване. При това продуктивността на решението е обусловена от многоезиковостта на интегрируемите програмни проекти. Концепцията .NET потенциално поддържа произволен език за програмиране, като най-известните езици са C#, Visual Basic, C++ и др. Една от основните идеи на .Net е съвместимостта на

различните части на приложението, които могат да бъдат разработени на различни езици.

Microsoft .NET Framework е платформа, създадена от Microsoft, която предоставя програмен модел, библиотека от класове (FCL, Framework Class Library) и среда за изпълнение на написания специално за нея програмен код (CRL, Common Language Runtime). Виртуалната машина CLR също така сама се грижи за основната безопасност, управлението на паметта и за системата с изключенията, като подпомага разработчика да не поема тези моменти от работата.

CLR може да изпълнява както обикновени програми, така и сървърни приложения. FCL съдържа множество компоненти за работа с бази от данни, мрежови функции, вход/изход, файлове, потребителски интерфейс и т.н. Това позволява на разработчика да не се занимава с програмиране на ниско ниво, а да използва готови класове.

По-важните части от библиотеката с класове са:

- Windows Forms, която отговаря за разработването на графичния интерфейс.
- ADO.NET - предоставя достъп до данните. Основно се използва за работа с бази от данни.
- ASP.NET е технология за разработване на уеб сайтове, уеб приложения и уеб услуги.
- Language Integrated Query (LINQ) е реализация на езика за заявки, приличаща по синтаксис на SQL в програмите на .Net.
- Windows Presentation Foundation (WPF) е система за създаване на графични интерфейси, използващи XAML.
- Windows Communication Foundation (WCF) е система за обмен на данни между .Net приложенията. Тя се използва за създаване на разпределени приложения.

В платформата Microsoft .NET Framework работи стандартна система за типове Common Type System (CTS), която напълно описва всички типове от данни, поддържани от изпълнителната среда, определя взаимодействието на типовете от данни и тяхното представяне във формата на метаданните на .NET. Строгата йерархичност на организацията на пространството от типове, класовете и имената на същностите на програмата позволява стандартизирането и унифицирането на реализацията. Системата за типизация на Microsoft .NET представлява частично подредено множество, което може да се разглежда като ISA йерархия (“is a”, което означава „явява се един от“).

Две големи групи типове са типове указатели и типове стойности, които отразяват особеностите при извикването на процедури: по име или по стойност (call-by-name, CBN) и по указател (call-by-reference, CBR). Системата за типизация на Microsoft .NET освен развитата йерархия на предопределените типове позволява на потребителя да създава собствени типове (както типове указатели, така и типове стойности) на база на вече съществуващите.

Наборът от правила, определящи подмножеството на обобщени типове от данни, по отношение на които се гарантира, че те са безопасни при използване от всички .NET езици, се описва в рамките на спецификацията Common Language Specification (CLS). За да могат класовете, разработени на различни езици да се използват съвместно в рамките на едно приложение, те трябва да удовлетворяват определени ограничения, задавани от CLS. Класът, удовлетворяващ CLS, се нарича CLS съвместим. Той е достъпен за използване от други езици, класовете на които могат да бъдат клиенти или наследници на съвместимия клас. Платформата .NET предоставя на програмиста библиотека от базови класове, достъпна за всеки един от .NET езиците за

програмиране. С цел структуризация функционално близките класове се обединяват в групи, наричани пространство от имена (Namespace).

Основното пространство от имена в FCL библиотеката е пространството System, съдържащо както класове, така и други вложени пространства от имена. За разработването на приложения с Visual Studio.NET се използват проекти с предопределен тип. Проектът се състои от класове, събрани в едно или няколко пространства от имена. Пространството от имена (Namespaces) позволява да се структурират проектите, съдържащи голям брой класове, като се обединяват в една група близките класове. Няколко проекта могат да се обединят в решение (Solution), което също така може да включва и ресурси, необходими за тези проекти.

Програмата за .NET Framework първо се компилира в единен за .NET междинен байт код Common Intermediate Language (CIL) (преди наричан Microsoft Intermediate Language, MSIL). В термините на .NET се получава assembly. След което кодът или се изпълнява на виртуалната машина Common Language Runtime (CLR), или се транслира с NGen.exe в изпълним код за конкретния целеви процесор. За предпочитане е използването на виртуална машина, понеже така се избягва необходимостта от отчитане на особеностите на апаратната част. В случая с използването на виртуална машина CLR вграденият в нея JIT компилатор (just in time) преобразува междинния байт код в машинен код за конкретния процесор. Съвременната технология за динамична компилация позволява да се достигне високо ниво на бързодействие. Виртуалната машина CLR също така сама се грижи за базовата безопасност, за управлението на паметта и системата на изключенията, като освобождава разработчика от тази част от работата.

Microsoft .NET Framework заедно с Internet Information Server, които са интегрирани компоненти на Windows Server, представляват пълен набор от функционалности EAS (Enterprise Application Server).

Продуктите на семейството корпоративни сървъри Microsoft Server System са предназначени, както и другите сървъри за приложения, за създаване и разгръщане на интегрирани корпоративни решения.

За тези сървъри са характерни:

- относително невисока стойност;
- поддръжка на XML;
- поддръжка на стандартите на Интернет;
- поддръжка на клъстерна архитектура;
- висока степен на взаимна интеграция.

По функционалност семейството сървъри Microsoft Server System, като цяло запълва практически всички съвременни направления за използване на сървъри за приложения.

Всички сървъри от семейството Microsoft Server System поддържат управление COM-, COM+- и .NET-компоненти и са достъпни за операционните системи на семейството MS Windows.

Component Object Model

COM (Component Object Model) е технологичен стандарт на компанията Microsoft, предназначен за създаване на програмно осигуряване на основата на взаимодействащи компоненти, всеки от които може да се използва едновременно от множество програми. Стандартът въплъщава в себе се идеите на полиморфизма и инкапсулацията на ООП.

Стандартът COM е разработен през 1993 г от корпорацията Microsoft в качеството на основа за развитие на технологията OLE (Object Linking and Embedding). Съществуващата към този момент технология OLE 1.0 вече е позволявала да се създават така наречените «съставни документи» (compound documents) (например в пакета Microsoft Office, да се включват диаграми от Microsoft Excel в документи на Microsoft Word).

Стандартът COM е могъл да бъде универсален и платформено-независим, но се е закрепил основно на ОС на семейството Microsoft Windows.

На базата на COM са се реализирали технологиите: Microsoft OLE Automation, ActiveX, DCOM, COM+, DirectX, а също така и XPCOM.

Основното понятие, с което оперира стандартът COM, е COM-компонента.

Програмите, построение по COM стандарта, фактически не се явяват автономни програми, а представляват набор от съвместно взаимодействащи COM-компоненти.

Всеки компонент има уникален идентификатор (GUID - Globally Unique Identifier) и може едновременно да се използва от много програми.

Компонентът взаимодейства с други програми чрез COM-интерфейси — набори от абстрактни функции и свойства. Всеки COM-компонент е длъжен, като минимум, да поддържа стандартния интерфейс «IUnknown», който предоставя базовите средства за работа с компонента и включва в себе си следните три метода:

- QueryInterface служи за получаване на указателя на интерфейса на COM-компонента;
- AddRef увеличива броя на указателите към дадения COM-компонент с единица;
- Release намалява броя на указателите към дадения COM-компонент с единица. Ако при поредното извикване на Release броят на указателите се е оказал равен на нула, тогава COM-компонентът е длъжен да се унищожи и трябва да се освободят всички ресурси.

Заедно функциите AddRef и Release служат за управление на времето на живот на COM- компонента, експортиращ интерфейсите.

Принципи на работа на COM

Базовите функции, позволяващи използването на COM-компоненти, се предоставят от Windows API. Windows API е проектиран за използване на езика C за създаването на приложни програми, предназначени за работа под управлението на операционата система MS Windows, и представлява множество от функции, структури от данни и числови константи, следващи правилата на езика C. Всички езици за програмиране, способни да извикват такива функции и да оперират с такива типове от данни в програмите, изпълнявани в средата на Windows, могат да използват този API. В частност, това са езиците C++, Pascal, Visual Basic и много други.

Съществуват библиотеки и среди за програмиране, предоставящи една или друга част на Windows API в по-удобен вид, като например, разработените от Microsoft: MFC, ATL/WTL, .Net/WinForms/WPF.

Microsoft Foundation Classes (MFC) е библиотека на езика C++, призвана да опрости разработването на GUI-приложенията за Microsoft Windows чрез използването на богат набор от библиотечни класове. MFC създава фреймуърк за приложенията — «рамкова» програма, автоматично създавана по зададен макет на интерфейса и самостоятелно изпълняваща рутинните действия, свързани с обслужването на приложението (отработка на прозоречни събития, прехвърляне на данни между буферите на различните вътрешни елементи и променливите и т.н.). Като програмистът

след генерирането на фреймуърка за приложенията е необходимо само да впише кода на мястото, където се изискват необходимите по-специални действия.

Active Template Library (ATL) е набор от шаблонни класове на езика C++, разработени за опростено написване на COM-компоненти. Тази библиотека позволява на разработчиците да създават различни COM обекти, сървъри за автоматизацията на OLE и ActiveX управляващи елементи. Средата за разработване Visual Studio включва мастери и помощници за ATL, позволяващи създаването на първична обектна структура практически без програмиране на ръка. ATL е в някаква степен опростена алтернатива на MFC в качеството на средства за управление на COM.

Windows Template Library (WTL) е свободно разпространявана библиотека от шаблони (шаблонни классове), написана на C++. Тя е предназначена за създаването на стандартни GUI приложения на MS Windows, представляващи разширение на ATL библиотеката. WTL представлява надстройка над интерфейса Win32 API, като тя първоначално се е разрабатвала в качеството на опростена алтернатива на MFC библиотеката. WTL поддържа работата с прозорци и диалози, стандартните диалози на MS Windows, GDI (Graphics Device Interface - интерфейс на MS Windows за представяне на графични обекти и предаването им на устройства за визуализация (като например монитори, принтери)), стандартни контроли, ActiveX и др. В библиотеката са представени основните елементи за управление: меню, панели с инструменти, бутони, полета за въвеждане, списъци и т. н.

OLE (Object Linking and Embedding) е технология за свързване и внедряване на обекти в други документи и обекти. OLE позволява да се предава част от работата от една програма за редактиране на друга и връщане назад на резултатите.

През 1996 г е направен опит за преименуване на OLE в ActiveX, но това става само частично и довежда до някакво объркване в термините.

Технологии, основани на стандарта COM

С термина ActiveX днес се свързва всичко, което има отношение към OLE, плюс някои нови въведения, обаче пълно съгласие по въпроса за точното определение на ActiveX сред експертите по DCOM (даже в Microsoft) не съществува.

COM+ (преди наричан Microsoft Transaction Server, MTS) е предназначена за поддръжка на системите за обработка на транзакциите. Технологията COM+ се базира на възможностите на COM и осигурява поддръжка на разпределените приложения на компонентна основа.

Обектите на транзакциите COM+ притежават основните свойства на COM обектите.

Освен това, обектите на транзакциите реализират специфични възможности за:

- управление на транзакциите;
- безопасност;
- пулинг на ресурсите (Resource pooling);
- пулинг на обектите (Object pooling).

Особености на COM+ обектите:

- реализация в състава на сървъра като динамична библиотека;
- наличие на указатели към COM+ библиотеката на типовете;
- използване само на стандартен механизъм за COM маршалинг;
- имплементация на IobjectControl интерфейса.

Възможни типове обекти:

- statefull;
- stateless.

COM+ е съвкупност от програмни средства, осигуряващи разработването, разпространяването и функционирането на разпределените приложения за мрежите Интернет и интранет.

В този състав влизат:

- програмното осигуряване от междинен слой, осигуряващо функционирането на обектите на транзакциите по време на изпълнението;
- MTS Explorer, позволяващ управляването на обектите на транзакциите;
- Интерфейсите на приложното програмиране;
- Средствата за управление на ресурсите.

Стандартният програмен модел на приложенията, използващи COM+, представлява трислойна архитектура на разпределените приложения. При което бизнес-логиката на приложението е концентрирана върху обектите на транзакциите, а програмното осигуряване на междинния слой, управляващ тези обекти, е построено с използването на компонентния модел.

Разработчиците, използващи COM+ за приложенията си, създават обекти от бизнес-логиката, удовлетворяващи изискванията към COM+ обектите. След което ги компилират и ги зареждат в COM+ средата с помощта на пакетите. Пакетът COM+ представлява контейнер, осигуряващ групирането на обектите с цел защита на данните, подобрява управлението на ресурсите и увеличава производителността. Управлението на пакетите се реализира с помощта на MTS Explorer.

DCOM (Distributed COM) е разширение на COM за поддръжка на връзките между разпределените (отдалечените) обекти.

Обща архитектура на DCOM

За създаването на обект на отдалечената машина, COM библиотеката извиква мениджър за управление на услугите (SCM) на локалния компютър, който се свързва с SCM сървъра и му предава заявката за създаване на обекта. Името на сървъра може да се задава при извикването на функцията за създаване на обекта или се съхранява в регистъра.

За извикването на отдалечен обект параметрите трябва да се извлекат от стека (или от регистрите на процесора), да се заредят в буфера и да се предадат по мрежата. Процесът на извличането на параметрите и разполагането им в буфера се нарича маршалинг. Този процес е нетривиален, понеже параметрите могат да съдържат указатели към масиви и структури, които, на свой ред, могат да съдържат указатели към други структури. На сървъра се реализира обратния процес на възстановяване на стека (демаршалинг), след което се извиква необходимия обект. След приключване на извикването се прави маршалинг на връщаната стойност и изходящите параметри и се изпращат обратно на клиента.

За изпълнението на маршалинг и демаршалинг е необходимо да има точно описание на метода, включително и на всички типове от данни и размерите на масивите. За описанието се използва език за описание на интерфейсите (IDL), който е част от стандарта на DCE RPC. Получените файлове на описанието се компилират от специален IDL компилатор в изходен код на езика C, реализиращ маршалинг и демаршалинг за зададените интерфейси. Кодът, зареждан на страната на клиента, се нарича «прокси», а на страната на обекта – «стъб» и се зарежда от COM библиотеката при необходимост.

Тема 14. SOA

Основните причини за появяването на SOA (service-oriented architecture) са високата динамика на съвременния бизнес и постоянно растящите изисквания за постоянна адаптация на информационните системи по отношение на тази динамика. Вече не е достатъчно информационната система да осигурява проста автоматизация на информационните и изчислителните задачи на бизнеса. Необходим е стремеж към това, променящите се условия на бизнеса, възникнали като следствие на жестока конкуренция, да намират пълно отражение в информационната система, т.е. корпоративната информационна система трябва да се променя толково бързо, колкото бързо се променят изискванията на бизнеса и бизнес процесите в компанията.

Първите пилотни проекти стартират през 2004 г.

SOA не е технология, а *философия, концепция, парадигма, подход* за построяване на корпоративни информационни системи, интеграция на бизнеса и на информационните технологии. По своята същност SOA не съдържа нови революционни идеи, а представлява обобщение на най-добрите практики за създаване на програмно-информационни системи.

За целите на бизнеса основните изисквания към компютърните мрежи са безопасността и скоростта за предаване на информацията. Изпълнението на тези изисквания се реализира на нивото на транспортните протоколи. Бързо развиващите се бизнес отношения са изисквали мрежовите приложения, използващи различни протоколи, да могат да си обменят данни. Така е възникнала задачата за интеграция на приложенията. Създадени са били няколко технологии за взаимодействие на разпределените приложения, позволяващи реализирането на обмен на данни между приложенията (сред получилите най-голямо развитие са RPC, Distributed COM (DCOM), RMI и CORBA, всяка от тях е сложна за реализация и не притежава необходимата универсалност, т. е. все пак е зависила от избора например на една и съща операционна система за всички участници в обмена) и друго, което не е добре е, че тези технологии особено трудно са се съвместявали. Всъщност се е реализирало окрупняване на групите невзаимодействащи по между си приложения, което не е устройвало както бизнеса, така и IT специалистите.

Тогава е бил избран друг - противоположен подход: погледът е бил насочен към базовите уеб технологии или това, което е основата на Интернет. Това са:

- TCP/IP;
- HTML;
- XML.

Следва да се отбележи универсалността на всяка от тези технологии, понеже тази универсалност е основата за разбирането на уеб услугите. Те са основани само на общоприети, отворени и формално независими технологии. Благодарение на това се постига основното предимство на уеб услугите като концепция за построяване на разпределени информационни системи. Универсалността се изразява във възможността за използване при различни операционни системи, езици за програмиране, сървъри за приложения и т.н. Така уеб услугите решават първоначалната задача - за интеграцията на приложенията с различна природа и построяването на разпределени информационни системи. В това се изразява основната принципна разлика на уеб услугите от предшествениците им.

SOA е извикване на уеб услуги с помощта на средства и езици за управление на бизнес процеси. SOA е термин, който се е появил за описание на изпълними

компоненти, такива като уеб услуги, които могат да се извикват от други програми, в качеството си на клиенти или от потребителите на тези услуги. Тези услуги могат да бъдат напълно съвременни (или даже остарели) приложни програми, които могат да се активират като черна кутия. От разработчика не се изисква да знае, как работи програмата, необходимо е само разбирането на това, какви входни и изходни данни са нужни и как се извикват тези програми за изпълнение.

В най-общ вид SOA предпола наличието на трима основни участници: доставчик на услугата, потребител на услугата и регистър на услугите. Взаимодействието на участниците изглежда просто: доставчикът на услугата регистрира своите услуги в регистъра, а потребителят се обръща към регистъра със заявка.

За използването на услугата е необходимо да се следва споразумението на интерфейса за обръщение към услугата, като интерфейсът е длъжен да не зависи от платформата. SOA реализира мащабируемост на услугите, т.е. възможност за добавяне на услуги, също така и тяхната модернизация. Доставчикът на услугата и неговият потребител общуват с помощта на съобщения. Понеже интерфейсът е длъжен да не зависи от платформа, тогава и технологията, използвана за определянето на съобщенията също така е длъжна да не зависи от платформата. Затова като правило съобщенията представляват XML документи.

В SOA каква е ролята и мястото на уеб услугите?

Отворените стандарти, описващи XML и уеб услугите, позволяват използването на SOA от всички технологии и приложения, работещи в компанията. Уеб услугите се базират на широко разпространени и отворени протоколи: HTTP, XML, UDDI, WSDL и SOAP. Именно тези стандарти реализират основните изисквания на SOA. Първо, услугата е длъжна динамично да се открива и извиква (UDDI, WSDL и SOAP). Второ, следва да се използва независещ от платформата интерфейс (XML). И накрая, HTTP осигурява функционална съвместимост.

Днес SOA се разглежда като ефективен инструмент за интеграция и за взаимодействие на процеси, реализирани от различни компании.

Тактически предимства на SOA:

- По-лесно разработване и внедряване на приложенията.
- Използване на текущите инвестиции.
- Намаляване на риска, свързан с внедряването на проекти в областта на автоматизацията на услугите и процесите.
- Възможност за непрекъснато подобряване на предоставяната услуга.
- Съкращаване на броя на обръщенията към техническа поддръжка.
- Повишаване на показателя за възврат на инвестициите (ROI).

Услугите като компоненти на информационната система

Услуга се нарича независим програмен компонент, изпълняващ определена задача, като например “проверка на кредитна карта”, не изискваща използването от клиентите на някаква конкретно определена програмна технология.

Използването на отворени стандарти представлява важна характерна особеност на SOA. Това значително намалява времето на включване на нова бизнес услуга към съществуващата система, също така при внедряването на SOA няма необходимост от пренаписване или просто отказване от проверени с годините и работещи решения.

Изборът на разпределената технология играе съществена роля.

Услугите в SOA реализират повтарящи се бизнес функции, които са необходими за организация на съгласуваната работа на сложни, съставени от голям брой различни

компоненти, приложения. За успешното внедряване и последващо функциониране на базираща се на SOA система при разработването на първо място следва да е направен анализ и описание на бизнес процесите на компанията.

Модулният подход при разработването на програмно осигуряване за SOA включва:
1. Анализ на бизнес процесите;
2. Разделяне на областите;
3. Проекция на услугите;
4. Реализация на услугите.

Фундаментални, твърдо зададени и написани в стандарта "принципи на SOA" не съществуват. Съществуват редица практики, към които следва да се придържаме при разработването на SOA системи.

1. Услугата следва да допуска повторно използване:
   - SOA системите са длъжни да поддържат повторно използване на всички услуги, независимо от моментните изисквания към техните функционални особености;
   - Последното ще позволи да се опрости разширяването и развитието на системите, отказване от "обвивки" над услугите за тяхната пренастройка за решаване на нови задачи;
   - Така всяка операция на услугата е длъжна да поддържа повторно използване.
   - Услугата може да се използва от няколко абоната.

2. Услугите са длъжни да осигуряват формален договор за ползване.
   Договорът за услугата предоставя следната информация:
   - Адрес на крайната точка;
   - Всички операции, предоставяни от услугата;
   - Всички съобщения, поддържани от всяка операция;
   - Правила и характеристики на услугата и нейните операции.

3. Услугите са длъжни да са слабо свързани.
   - Необходимо е осигуряване на цялостност на системата в рамките на развитието на системата, независимо от вариантите на развитие;
   - Системата от услуги е слабо свързана, ако услугата може да получава знания за друга услуга, оставайки независима от вътрешната реализация на логиката на дадената услуга.

4. Услугите са длъжни да се абстрахират от вътрешната логика:
   - Всяка услуга е длъжна да работи като черна кутия;
   - Това е едно от изискванията, осигуряващи слабата свързаност на услугите.

5. Услугите задължително са съвместими:
   - Услугата може самостоятелно да реализира логиката, а също и да използва други услуги за своята реализация;
   - Услугите трябва така да са проектирани, че да се поддържа възможността за използването им в качеството на елементи на друга услуга;
   - Този процес се нарича оркестрация на услугите.

6. Услугите трябва да са автономни:

- Областта на бизнес логиката и ресурсите, използвани от услугата, следва да са ограничени в зададени граници;
- Вопросът за автономността е най-важният аргумент при разпределянето на бизнес логиката на отделните услуги.

Съществуват два типа автономност:
- *Автономност на нивото на услугата*: границите на отговорност на услугите са разделени, но те могат да използват общи ресурси;
- *Чиста автономност:* бизнес логиката и ресурсите се намират под пълен контрол от услугата.

7. Услугите не са длъжни да използват информация за състоянието:
- "Чистите" услуги са длъжни значително да ограничават обема и времето за съхраняване на информацията (в идеалния случай - само по време на изчисленията);
- Независимостта от състоянието позволява да се увеличи възможността за мащабиране и повторно използване на услугите;
- Това ограничение не винаги е възможно да бъде удовлетворено.

8. Услугите трябва да бъдат откриваеми:
- Откриването на услугите позволява да се избегне случайното създаване на излишни услуги, осигуряващи логика в излишък;
- Всички методи на услугата трябва да съдържат метаданни, описващи възможностите им в системата за търсене;
- Всяка услуга трябва да предоставя колкото може повече информация за своите възможности.

Уеб услугите не бива да се разглеждат като универсално решение за всички бизнес проблеми, налични или възможни. Макар че уеб услугите представляват логично и напълно зряло продължение на предшестващите ги технологии за изграждане на разпределени информационни системи, те са технология, която както и останалите, има своите положителни и отрицателни страни, и като следствие и рамки за приложение. Неотчитането на тези ограничения в реалните проекти може да доведе до негативни последствия. По-подробно ще разгледаме плюсовете и минусите.

Към плюсовете на уеб услугите могат да се отнесат следните особености:
- Уеб услугите позволяват на компанията да осъществява интеграция на собствените бизнес процеси с бизнес процесите на бизнес партньорите и клиентите при по-ниска стойност от тази, при използването на други интеграционни технологии;
- Понеже уеб услугите се организират в публични регистри, достъпни на заинтересованите лица по целия свят, така се намалява прага за излизане на компаниите на нови пазари, но се увеличават възможностите за клиентска база;
- Уеб услугите осигуряват приемственост по отношение на наличните в компанията информационни системи, т.е. уеб услугите са надстройка над съществуващите информационни системи, но не функционират вместо тях. Така се осигурява поддръжка на направените инвестиции в IT инфраструктурата и не се налага използването на нови, понеже няма потребност от радикални промени;
- Построяването на нови корпоративни решения с използването на уеб услуги се реализира по-бързо и по-евтино, понеже основното внимание се съсредоточава върху създаването на бизнес логиката на решението, програмирането на самите

уеб услуги, което ефективно прилага повторно използване на кода и на адаптираните средства за разработка.

Минусите са следните:

- Стандартите за интеграция на бизнес процеси, това са въпроси на управлението на транзакциите и изработването на единни бизнес и IT политики на взаимодействащите посредством уеб услугите на компанията. Те се определят на стадия на разработване. Без тяхната ясна формализация и публикуване построяването на информационна система на базата на уеб услуги може да е с променлив успех;
- Динамичното използване на информацията от бизнес регистрите на уеб услугите изисква решаването на въпросите на доверителност на отношенията между различните бизнес регистри. Също така има трудности със съвместното използване на бизнес регистри от различен формат;
- Добавянето към функциите на сървъра на приложенията функционалност на доставчик на уеб услуги може да представлява определена сложност;
- Въпросите за безопасността на функционирането на информационната система на базата на уеб услуги все още не са напълно решени.

Анализирайки плюсовете и минусите може да се отбележи, че плюсовете са бизнес предимства от стратегически характер, а минусите са от технологичен характер и отчитайки темповете на развитие на IT, решението на тези проблеми е въпрос на време.

Формално определение на уеб услуга

Уеб услуга са нарича програмна система, идентифицируема с URI ред, чиито публични интерфейси и свързвания са определени и описани посредством XML. Описанието на тази програмна система може да се намери от други програмни системи, които могат да взаимодействат с нея според това описание посредством съобщения, основани на XML и предавани с помощта на Интернет протоколите.

Услугата е ресурс, реализиращ бизнес функция, притежаваща следните свойства: възможност за повторно използване; определена е от един или няколко явни технологически независими интерфейса; слабо е свързана с други подобни ресурси и може да бъде извикана чрез комуникационен протокол, осигуряващ възможност за взаимодействие на ресурсите по между си.

От функционална гледна точка бизнес приложенито е съвкупност от взаимодействащи услуги. Тази съвкупност от взаимодействащи услуги може да се отъждестви с SOA:

Компонентен модел, състоящ се от отделни функционални модули на приложения, наричани услуги, притежаващи определени според някакво общо правило интерфейси и механизми за взаимодействие помежду си, се нарича SOA.

Архитектура от приложения, в рамките на която всички функции на приложението са независими услуги с определени интерфейси, които могат да се извикват в необходимия ред с цел формиране на бизнес процеси, се нарича SOA.

В SOA присъстват елементи на обектния подход при построяване на информационни системи: декомпозиция (приложения за различните функции) и инкапсулация (услуги като черни кутиии). Обаче терминът "обектно-ориентиран" по отношение на SOA не е коректен, понеже в SOA отсъстват всички необходими елементи на обектно-ориентираната идеология. По-правилно е да се нарича SOA концепция, използваща обектния подход.

Тема 15. GRID технологии. Облачни изчисления

Терминът «грид» е въведен в обръщение от Ian Foster в началото на 1998 година в книгата «Грид. Нова инфраструктура на изчисленията»:

Грид това е система, която координира разпределени ресурси посредством стандартни, отворени, универсални протоколи и интерфейси за осигуряване на нетривиално качество на обслужване (QoS – Quality of Service).

Въпреки, че през годините базовата идея на грид не е претърпяла съществени изменения, всеобхватно определение за грид не съществува и до днес.

Архитектура на Грид

Основната идея, заложена в концепцията за грид изчисленията, представлява централизирано отдалечено предоставяне на ресурси, необходими за решаването на разнородни изчислителни задачи. В някакъв смисъл концепцията на грид изчисленията идейно се съгласува с концепцията за електрическата мрежа (Power Grid): за нас не е важно откъде идва електричеството в контакта. Независимо от това можем да си включим в електрическата мрежа ютията, компютъра или пералнята. Също така и в идеологията на грид: ние можем да зададем всякаква задача за решаване на всеки компютър или мобилно устройство, като ресурсите за това изчисление трябва автоматично да се предоставят от отдалечени високо-производителни сървъри, независимо от типа на нашата задача.

Главната задача, лежаща в основата на концепцията на грид, това е съгласувано разпределение на ресурсите и решаване на задачите в условията на динамични, многопрофилни виртуални организации. Разпределението на ресурсите, в което са заинтересовани разработчиците на грид, това не е обмен на файлове, а директен достъп до компютри, програмно осигуряване, данни и други ресурси, които са необходими за съвместното решаване на задачите и за стратегията за управление на ресурсите, възникваща в промишлеността, науката и техниката. Виртуална организация се нарича редица от отделни хора или учреждения, обединени от единни правила за колективен достъп до разпределени изчислителни ресурси. За организацията на работата в рамките на такива виртуални организации възниква необходимост от следните елементи:

1. гъвкави механизми за разделяне на ресурсите, от клиент-сървърни до такива, с равноправен достъп;
2. развита система за контрол на използването на ресурсите, включително контрол на методите за достъп и използването на локални и глобални подходи;
3. разпределен достъп до различни ресурси, като програми, файлове и данни, а също така и компютри, сензори и мрежи;
4. различни модели за използване на ресурсите (от еднопотребителски до многопотребителски, от високопроизводителни до икономични) и, следователно, включващи регулиране на качеството на предоставяното обслужване, планиране, преразпределение и водене на отчетност на ресурсите.

Анализът на алтернативните технологии за изграждане на разпределени изчислителни системи е показал, че тяхното използване не позволява в пълна степен да се достигнат всички изисквания, посочени по-горе. В съответствие с това е била предложена алтернативна архитектура на грид. Изследванията и разработките в съобществото на грид довеждат до създаването на протоколи, услуги и инструментариум, насочен към решаването на проблемите, които възникват при опит за създаване на мащабируеми виртуални организации. Тези технологии включват:

1. свързани с безопасността решения, поддържащи управление на сертифицирането и политиките за безопасност, когато изчисленията се осъществяват от няколко организации;

2. протоколи за управлението на ресурсите и услугите, поддържащи безопасен отдалечен достъп до изчислителните ресурси и ресурсите с данни, а също така и преразпределение на различните ресурси;

3. протоколи за заявки на информация и услуги, осигуряващи настройка и мониторинг на състоянието на ресурсите, организациите и услугите;

4. услуги за обработване на данни, осигуряващи търсене и предаване на набори от данни между системите за съхраняване на данни и приложенията.

В архитектурата на грид са представени следните слоеве (нива):

1. Базов слой (Fabric) – съдържа различни ресурси, като компютри, устройства за съхраняване, мрежи, сензори и др.

2. Свързващ слой (Connectivity) – определя комуникационните протоколи и протоколите за автентикация.

3. Ресурсен слой (Resource) – реализира протоколите за взаимодействие с ресурсите на разпределените изчислителни системи и тяхното управление.

4. Колективен слой (Collective) – управление на каталозите с ресурсите, диагностика, мониторинг;

5. Приложен слой (Applications) – инструментариум за работа с грид и потребителските приложения.

На базовия слой се определят службите, осигуряващи непосредствен достъп до ресурсите, използването на които е разпределено посредством Грид протоколите.

1. Изчислителните ресурси предоставят на потребителя на Грид системата (с други думи, за задачата на потребителя) процесорни мощности. Изчислителни ресурси могат да бъдат както клъстери, така и отделни работни станции. При цялото разнообразие от архитектури всяка изчислителна система може да се разглежда като потенциален изчислителен ресурс на Грид-система.

2. Ресурсите за памет представляват пространство за съхраняване на данни. За достъп до ресурсите за паметта също така се използва програмното осигуряване на междинния слой, реализиращо унифициран интерфейс за управление и предаване на данни.

3. Информационните ресурси и каталозите представляват особен вид ресурс за памет. Те служат за съхраняване и предоставяне на метаданни и информация за други ресурси на Грид системите.

4. Мрежовият ресурс представлява свързващо звено между разпределените ресурси на Грид системата. Основната характеристика на мрежовия ресурс е скоростта за предаване на данните.

Свързващият слой определя комуникационните протоколи и протоколите за автентикации, осигурява предаването на данните между ресурсите на базовия слой. Свързващият слой на грид се базира на TCP/IP протоколния стек:

1. Интернет протоколи (IP, ICMP);
2. Транспортни протоколи (TCP, UDP);
3. Приложни протоколи (DNS, OSRF…).

Ресурсният слой реализира протоколите, осигуряващи изпълнението на следните функции:

- съгласуване на политиките за безопасност на използвания ресурс;
- процедура за инициализация на ресурса;
- мониторинг на състоянието на ресурса;
- контрол над ресурса;
- отчитане на използването на ресурса.

Също така има два типа протоколи на ресурсния слой:

1. Информационни протоколи – използват се за получаване на информация за структурата и състоянието на ресурса.
2. Управляващи протоколи – използват се за съгласуване на достъпа до разделяемите ресурси, като определят изискванията и допустимите действия по отношение на ресурса (например поддръжка на резервиране, възможност за създаване на процеси, достъп до данни).

Колективният слой отговаря за глобалната интеграция на различните набори ресурси и може да включва служби с каталози; служби за съвместно предоставяне, планиране и разпределение на ресурсите; служби за мониторинг и диагностика на ресурсите; служби за репликация на данните.

На приложния слой се разполагат потребителските приложения, изпълнявани в средата на виртуалната организация. Те могат да използват ресурси, които се намират на по-долу лежащите слоеве на Грид архитектурата.

Облачни изчисления

Облачните изчисления привличат силно вниманието през последните години. В средствата за масова информация, в Интернет, даже по телевизията могат да се видят възторжени материали, описващи всички „прекрасни“ възможности, които може да ни предостави тази технология.

Въпреки, че метафората «облак» отдавна се използва от специалистите в областта на мрежовите технологии за представяне на мрежови диаграми със сложна изчислителна инфраструктура (или даже Интернет като такъв), скриващи вътрешната си организация зад определени интерфейси, терминът «Облачни изчисления» се е появил сравнително неотдавна. Според резултатите от анализа на търсещата система Google, терминът «Облачни изчисления» («Cloud Computing») е започнал да се налага в края на 2007 – началото на 2008 година, като постепенно измества популярното в това време словосъчетание «Грид изчисления» («Grid Computing»). Съдейки по заглавията на новините от онова време, една от първите компании, дала на света този термин, е компанията IBM, разгърнала в началото на 2008 година проекта «Blue Cloud» и спонсорирала Европейския проект «Joint Research Initiative for Cloud Computing».

Днес вече може да се твърди, че облачните изчисления стабилно са влезли в ежедневния живот на всеки потребител на Интернет (въпреки, че се намират и такива, които за последното не подозират). Според някои експертни оценки, технологията за облачни изчисления може от три до пет пъти да съкрати стойността на бизнес приложенията и повече от пет пъти - стойността на приложенията за крайните потребители, но, въпреки общото позитивно отношение към облачните технологи, и до сега ясно разграничаване на това, какво е «Облачни изчисления» и по какъв начин те се съотнасят с парадигмата на «Грид изчисленията». За да се изясни последното ще разгледаме няколко съществуващи определения за облачни изчисления, ще изясним

основните характеристики на облаците и ще видим, какви общи аспекти може да се определят в архитектурата на облачните решения.

Определение за облачни изчисления

От една страна, терминът «Облачни изчисления» няма наложило се стандартно определение. От друга страна, множество различни корпорации, учени и аналитици дават собствени определения на този термин.

Определението за облачни изчисления е предизвикало дебати и в научното съобщество. За разлика от определенията, които могат да се намерят в комерсиалните издания, научните определения се ориентират не само върху това, което ще се предостави на потребителя, но и върху архитектурните особености на предлаганата технология.

Например в Калифорнийския университет в Бъркли са дали следното определение за облачни изчисления:

«Облачните изчисления – това са не само приложения, предоставяни като услуги от Интернет, но и апаратни средства и програмни системи в центровете за обработване на данни, които осигуряват предоставяне на тези услуги».

Услугите сами за себе си отдавна се наричат «предоставяне на програмно осигуряване като услуги» (Software-as-a-Service или SaaS). Облак се нарича апаратното и програмното осигуряване на центъра за обработване на данни. Обществеият облак предоставя ресурсите на облака на широк кръг потребители на принципа «заплащане според използването» (pay-as-you-go – принцип за предоставяне на услуги, при който потребителят заплаща само за тези ресурси, които са изразходвани при решаването на поставената задача). Частният облак – това са вътрешни центрове за обработване на данни, в комерсиална или друга организация, които не са достъпни на широк кръг потребители. Така облачните изчисления представляват обединанеи от SaaS и «комунални изчисления» (Utility Computing – модел за изчислителни системи, в който предоставянето на данни и процесорна мощ се организира на принципите за ползване на комунални услуги). Хората могат да бъдат потребители или доставчици на SaaS, или потребители или доставчици на комунални изчисления».

Даденото сложно и обширно определение показва другата страна на облачните изчисления: от гледна точка на доставчика, най-важната съставляваща на облака е центърът за обработване на данните. Центърът за обработване на данните съдържа изчислителни ресурси и хранилища на информация, които съвместно с програмното осигуряване се предоставят на потребителя на принципа «заплащане според използването».

Ian Foster определя облачните изчисления така: «Това е парадигма за мащабни разпределени изчисления, основана на ефекта на мащаба, в рамките на която пул от абстрактни, виртуализирани, динамично мащабируеми изчислителни ресурси, ресурси за съхранение, платформи и услуги се предоставят по заявка на външни потребители от Интернет».

Даденото определение добавя двата най-важни аспекта в определението на облачните изчисления: виртуализацията и мащабируемостта. Облачните изчисления се абстрахират от базовата апаратна и програмна инфраструктура посредством виртуализацията. Виртуализираните ресурси се предоставят посредством определени абстрактни интерфейси (API програмни интерфейси или услуги). Такава архитектура осигурява мащабируемост и гъвкавост на физическия слой на облака без последствия за интерфейса на крайния потребител.

След обработването на повече от 20 определения за облачни изчисления от Luis M. Vaquero от Hewlett-Packard Labs е предложено следващото определение за облачни изчисления (по-всеобхватно и цялостно от първоначалните):

«Облакът – това е голям пул от лесни за използване и лесно достъпни виртуализирани ресурси (такива като апаратни комплекси, услуги и др.). Тези ресурси могат динамично да се преразпределят (да се мащабират) за настройване към динамично променящо се натоварване, осигуряващо оптимално използване на ресурсите. Този пул от ресурси обикновено се предоставя на принципа «заплащане според консумацията». При което собственикът на облака гарантира качество на обслужването на база на определени споразумения с потребителя».

Всички определения илюстрират следната простичка мисъл: феноменът на облачните изчисления обединява няколко различни концепции на информационните технологии и представлява нова парадигма за предоставяне на информационни ресурси (апаратни и програмни комплекси). От страната на собственика на изчислителните ресурси облачните изчисления са ориентирани да предоставят информационни ресурси на външни потребители. За страната на потребителя, облачните изчисления – това е получаването на информационни ресурси във вид на услуги от външен доставчик, заплащането на които става в зависимост от обема на използваните ресурси според установена тарифа. Ключовите характеристики на облачните изчисления са мащабируемостта и виртуализацията.

Мащабируемостта представлява възможност за динамична настройка на информационните ресурси за променящото се натоварване, например при увеличаване или намаляване на броя на потребителите, промяна на необходимия капацитет на хранилищата за данни или на изчислителната мощност.

Виртуализацията, която също така се разглежда като най-важната технология за всички облачни системи, основно се използва за осигуряване на абстракция и инкапсулация.

Абстракцията позволява да се унифицират «суровите» изчислителни, комуникациони ресурси и хранилищата на информацията във вид на пул от ресурси и да се построи унифициран слой от ресурси, който съдържа същите ресурси, но в абстрактен вид. Те се предоставят на потребителите и на горните слоеве на облачните системи като виртуални сървъри, сървърни клъстери, файлови системи и СУБД. Инкапсулацията на приложенията повишава безопасността, управляемостта и изолираността. Друга също така важна особеност на облачните платформи е интеграцията на апаратните ресурси и системното програмно осигуряване с приложенията, които се предоставят на крайния потребител във вид на услуги.

В съответствие с изложеното по-горе, могат да се определят следните основни моменти за облачните изчисления:

− облачните изчисления представляват нова парадигма за предоставяне на изчислителни ресурси;

− базовите инфраструктурни ресурси (апаратни ресурси, системи за съхраняване на данни, системно програмно осигуряване) и приложения се предоставят във вид на услуги. Дадените услуги могат да се предоставят от независим доставчик за външни потребители според принципа «заплащане според потреблението»;

− основните особености на облачните изчисления са виртуализацията и динамичната мащабируемост;

− облачните услуги могат да се предоставят за крайния потребител посредством уеб браузър или посредством определен програмен интерфейс.

Днес облачните изчисления са множество от платформи, предоставяни от компаниите Google, Microsoft, Amazon и др.

Тема 16. Архитектури на информационните системи

В архитектурата на всяка конкретна информационна система (ИС) често пъти може да се открие влиянието на няколко архитектурни решения. Областта на информационните системи се развива изключително бързо. Всички подходи за организация на информационни системи за работа в мрежата се базират на общата архитектура клиент-сървър.

Функции на ИС:

- Потребителски интерфейс (презентационни функции) - функции, реализиращи взаимодействието на потребителите с ИС,
- Бизнес логика - функции по обработка на информацията в съответствие със специфичните правила за бизнес процеса, характерни за съответната приложна област (бизнес модел - търговия, транспорт и т.н.),
- *Функции по управление, съхранение и извличане на данни от бази данни или отделни файлове;*
- Бизнес модел - отговаря на съответната приложна област (търговия, транспорт и т.н.). Състои се от определен брой бизнес процеси;
- Бизнес процес - работен цикъл (повтаряща се последователност от действия, която може да бъде документирана, като доставка, продажба и др.),
- Бизнес логика - операциите по обработка на информацията в съответствие с правилата на бизнес процеса (ДДС, търговска отстъпка, % печалба, разходни норми и т.н.);

Бизнес процесът притежава: *Цел* - решава определени задачи; *Вход; Изход*; *Използва ресурси*, в т.ч. и информационни; *Влияе хоризонтално на фирмата*, т.е може да засяга повече от една организационна единица (отдел доставки, склад и др.); *Определени брой дейности, изпълнявани последователно* (бизнес логика);

В зависимост от разположението и реализацията на трите функции, съществуват: Еднослойна архитектура, Двуслойна архитектура, Трислойна архитектура, Четирислойна архитектура, Сложна (N слойна) архитектура;

1. Еднослойна архитектура - приложението се изпълнява върху един компютър, като трите вида функции са свързани,
2. Двуслойна архитектура - обособяване на два слоя:
    - Клиентски слой - обхваща потребителския интерфейс и бизнес логиката,
    - Сървърен слой - реализира всички функции по работа с базите данни;
3. Трислойната архитектура - обособяване на трите вида функции в три самостоятелни слоя:
    - Слой на потребителския интерфейс,
    - Слой на бизнес логиката,
    - Слой на управление на данните;

- с “тънък” клиент - на клиентската работна станция е инсталиран потребителския интерфейс, а на сървъра - бизнес логиката и управлението на данните чрез СУБД,
- с “дебел” клиент - на клиентската работна станция са реализирани потребителския интерфейс и бизнес логиката, на сървъра - само слоя за управление на данните, чрез определена СУБД,

- Клиентската работна станция с “тънък” клиент притежава:
  - Ограничени технически възможности - процесор и памет,
  - Мрежова карта и входно- изходни устройства,

  Предимства: По-ниски разходи за реализиране, Не е необходимо локално администриране на данните,
- Клиентската работна станция с “дебел” клиент притежава:
  - Високи технически възможности - процесор и памет, оперативна и дискова,
  - Мрежова карта и входно- изходни устройства,

  Предимства: Не е необходимо локално администриране на данните (backup);

  Недостатъци: По-високи разходи за реализиране.

СУБД основана на файл-сървърна архитектура

Организацията на информационните системи на базата на използването на файл-сървъри все още е сред най-разпространените поради наличието на голям брой персонални компютри (PC) и сравнително евтиното свързване на PC-та в локални мрежи. При файл-сървърната архитектура се запазва автономността на приложното (и на по-голямата част на системното) програмно осигуряване, работещо на всяко PC в мрежата. Компонентите на информационната система, изпълнявани на различни PC-та, взаимодействат само поради наличието на общо хранилище на файловете, което стои на файл-сървъра. В класическия случай на всяко PC се дублират не само приложните програми, но и средствата за управление на базите от данни. Файл-сървърът представлява разделяем от всички PC-та комплекс за разширение на дисковата памет.

Основното достоинство на тази архитектура е лесната организация. Обаче за много от детайлите тази простота е привидна. Първо, на информационната система ѝ предстои да работи с база от данни. Следователно тази база от данни трябва да се проектира. Колкото по-качествено е проектирана, толкова са по-големи шансовете информационната система ефективно да се използва впоследствие. Сложността на проектирането на базата от данни се определя от обективната сложност на моделираната предметна област. Второ, необходимо изискване към базата от данни на информационната система е поддържане на цялостност на състоянието ѝ и гарантирана надеждност на съхраняваната информация. Минималните условия, при спазването на които могат да бъдат удовлетворени тези изисквания, са: наличие на транзакционно управление; съхраняване на данни в излишък (например с прилагането на методите за журналиране); възможност за формулиране на ограничения за цялостност и проверка за спазването на тези ограничения.

Работещото с неголеми обеми информация и разчетено за използване в еднопотребителски режим, файл-сървърното приложение може да се проектира, разработва и дебъгва много лесно. Често пъти за неголяма компания е достатъчно да има такава изолирана система, работеща на отделно PC. Обаче при по-сложните случаи файл-сървърните архитектури са недостатъчни.

Сървъри за БД

При *сървър за бази от данни* програмната система, свързана със съответната база от данни, съществува независимо от съществуването на потребителски (клиентски) процеси и се изпълнявана (не задължително) на отделна апаратура.

Тези архитектури са със съществено по-сложна организация от предишните (файл-сървъри), осигуряват по-детайлно и ефективно управление на базите от данни.

Интерфейсът на развитите сървъри за бази от данни е основан на използването на езика от високо ниво за бази данни SQL, което позволява използването на мрежовия трафик между клиента и сървъра при базите от данни само за полезни цели (от клиента към сървъра основно се изпращат оператори на езика SQL, от сървъра към клиента-резултати от изпълнението на операторите). При файл-сървърната организация клиентът работи с отдалечени файлове, което представлява сериозно натоварване на трафика. Като цяло във файл-сървърната архитектура има "дебел" клиент и много "тънък" сървър в смисъл, че почти цялата работа се изпълнява на страната на клиента, а от сървъра се изисква само достатъчен капацитет за дискова памет. Интерфейсът между клиентската част на приложението и сървъра за бази от данни е основан на използването на езика SQL. Затова такива функции, като например предварителна обработка на форми, предназначени за заявки към базата от данни или формирането на резултатни отчети се изпълняват в кода на приложението. При клиент-сървърната организация клиентите могат да са доста "тънки", а сървърът е длъжен да е "дебел" дотолкова, че да е в състояние да удовлетворява потребностите на всички клиенти.

Разработчиците и потребителите на информационни системи, базирани на архитектурата клиент-сървър, често пъти са неудовлетворени от потребността за обръщение от клиента към сървъра за всяка поредна заявка. На практика е разпространена ситуацията, когато за ефективната работа на отделната клиентска съставляваща на информационната система в действителност се изисква само неголяма част от общата база данни. Това води до идеята за поддръжка на локален кеш на общата база данни на страната на всеки клиент. Архитектурата клиент-сървър на пръв поглед изглежда много по-скъпа от архитектурата файл-сървър. Необходима е по-мощна апаратура (поне за сървъра) и съществено по-развити средства за управление на базите данни. Но това е само частично така. Голямото предимство на клиент-сървърната архитектура е нейната мащабируемост и способност за развитие.

Intranet приложения

Като мрежово архитектурно решение и*нформационната Intranet система* е корпоративна система, в която се използват методите и средствата на Internet. Такава система може да бъде локална, изолирана от осталия свят на Internet или да се опира на виртуална корпоративна подмрежа на Internet. В последния случай са особено важни средствата за защита на информацията от несанкциониран достъп.

Въпреки предимствата си, тази схема притежава ограничения. Това, което може потребителят е само преглед на информацията, поддържана от уеб сървъра. Хипертекстовите структури трудно се модифицират. Не винаги е достатъчно търсенето на информация в стил преглед на хипертекст. На практика тези трудности се разрешават с по-развитите механизми на уеб технологиите.

Intranet е удобно и мощно средство за разработване и използване на информационни системи. Единствен относителен недостатък на този подход е постоянното изменение на механизмите и отсътствието на стандарти. От друга страна, ако създадената информационна система използва текущото ниво на технологиите и се окаже, че удовлетворява потребностите на корпорацията, тогава няма защо нещо да се променя в системата поради причина, като появяването на по-съвършени механизми.

Трислойна архитектура при уеб базирани приложения

1. Визуализирането на потребителския интерфейс се реализира чрез програма браузър,
2. Включва уеб сървър за управление на потребителския интерфейс и слой на бизнес логиката,

3. Включва уеб сървър за данни,

Управлението на потребителския интерфейс не се разграничава строго от бизнес логиката. Реализира се на някой от езиците, поддържани от уеб сървърите - ASP, JSP, PHP и т.н.

4 слойна архитектура при уеб базирани приложения

1. На клиентския компютър се визуализира потребителския интерфейс,
2. На уеб сървър се реализира управлението на потребителския интерфейс,
3. Бизнес логиката се реализира като самостоятелен слой на компютър-сървър на приложенията,
4. На уеб сървър за данни се извършва управлението им;

Предимства на приложенията с N слойна архитектура

- Модифицируемост - дава възможност да се променя функционалността на един от модулите, без това да влияе на другите модули,
- Разширяемост - възможност на добавяне на нови функции чрез включване на нови модули с бизнес логика,
- Многократна използваемост - възможност за изграждане на нови приложения чрез интегриране на съществуващи бизнес компоненти;

До тук са разгледани начините и възможните архитектури на информационни системи, предназначени за оперативна обработка на данни, т.е. за получаване на текуща информация, позволяваща решаването на ежедневните проблеми на корпорацията. Обаче аналитичните звена на корпорацията и висшият управленски състав имат и други задачи, като например анализиране поведението на корпорацията на пазара с отчитането на съпътстващи външни фактори и прогнозиране поне за най-близко бъдеще, избор на тактика и на стратегия на корпорацията. За решаването на тези задачи са необходими данни и приложни програми по-различни, използвани за оперативни информационни системи. През последните години те стават все по-популярни.

Склад за данни (DataWarehousing) и системи за оперативна аналитична обработка на данни

DataWarehousing е голяма база данни, която може да има достъп до цялата информация на дадена компания или организация. Тази огромна база данни съхранява информация като хранилище за данни, но отива с една крачка по-далеч: дава на потребителите си достъп до данни за извършване на изследователски анализ.

През 1993 г. основоположникът на релационния подход за организация на базите данни Edgar Codd, отчитайки потребностите на системите за динамична аналитична обработка на данни, е формулирал 12 основни изисквания към системите, поддържащи аналитични бази данни. Тези изисквания представят гледната точка при проектирането и разработването на системите за аналитична обработка на данните.

1. *Многомерно концептуално представяне на данните.* Това изискване възниква поради причината, че бизнес потребителят разбира се си представя историята и дейността на корпорацията като многомерни (например едното измерение е времето, другото - клиентите, третото - произведената продукция и т.н.). OLAP моделите са длъжни да поддържат това представяне и то трябва да се опира на възможностите на аналитичната база данни.
2. *Прозрачност.* За бизнес потребителя не бива да е съществено къде конкретно са разположени средствата за динамичен анализ на данните. При разработването на OLAP системи следва да се придържа към подхода на отворените системи, което

ще позволи разместването на средствата за анализ на кой да е възел в корпоративната мрежа.

3. *Достъпност*. Логическата схема, по която работи OLAP системата, трябва да се отразява в схемите на разнородни физически складове за данни. При достъп до данните трябва да се поддържа тяхното единно и съгласуванно представяне.
4. *Съгласувана ефективност при производството на отчетите.* Тази ефективност не бива да деградира при увеличаването на броя на измеренията.
5. *Архитектура клиент-сървър.* Сървърният компонент на OLAP системата трябва да бъде достатъчно развит за да могат разнородни клиенти да се свързват с него с минимални усилия и загуби за допълнително "интегриращо" програмиране.
6. *Родова многомерност.* Структурните и операционните възможности за работа с всяко измерване на данните трябва да са еквивалентни. За всички измервания трябва да съществува само една логическа структура. Всяка функция, приложима към едно измерване, трябва да е приложима и към всяко друго измерване.
7. *Управление на динамично разредени матрици.* Сървърът на OLAP системата трябва да може ефективно да съхранява и обработва разредени матрици. Физическите методи за достъп трябва да са разнообразни, да включват прякото изчисление, B-дървета, хеширане или комбинации на тези методи.
8. *Поддръжка на многопотребителски режим.* OLAP системата е длъжна да поддържа многопотребителски достъп до данните, осигурявайки цялостност и безопасност на данните.
9. *Неограничени операции между измерванията.* При изпълнението на многомерен анализ на данните всички измервания се създават и обрабатват еднообразно. OLAP системата трябва да може да изпълнява съответните изчисления между измерванията.
10. *Интуитивно манипулиране с данните.* Манипулациите, подобно на промяната на пътя на анализа или нивото на детализация, трябва да се изпълнява с помощта на прякото въздействие на елементите на OLAP модела без потребност от използване на меню или други спомогателни средства.
11. *Гъбкава отчетна система.* Бизнес потребителят трябва да има възможност да манипулира с данните, да анализира и/ или синтезира, а също така и да ги преглежда по начин, какъвто му харесва.
12. *Неограничен брой измервания и нива на агрегация.* OLAP сървърът е длъжен да поддържа не по-малко от 15 измервания за всеки аналитичен модел. За всяко измерване трябва да се допуска неограничен брой определяни от потребителите агрегати.

Интегрирани разпределени приложения

Обикновено няма проблеми, ако от самото начало информационното приложение е проектирано и разрабатено с използването на подхода на отворените системи: всички компоненти са мобилни и има оперативна съвместимост (interoperability), общото функциониране на системата не зависи от конкретното местоположение на компонентите, системата притежава добри възможности за съпровождане и развитие. За съжаление на практика този идеал е трудно постижим. Поради най-различни причини възникава потребността от интеграция при независими и организирани по различен начин информационно-изчислителни ресурси. Става ясно, че тук се налага прилагането на някаква технология за интеграция. Съществува предложен подход от *OMG (Object Management Group).*

Фактори, стимулиращи използването на методите за интеграция на разнородни информационни ресурси:

1. *Неоднородност, разпределеност и автономност на информационните ресурси на системата.* Неоднородността на ресурсите може да бъде синтактична (например при представянето им се използват различни модели от данни) и/ или семантична (използват се различни видове семантични правила, детайлизират се и/ или се агрегират различни аспекти на предметната област). Възможна е реализирана неоднородност на информационните ресурси, обусловена от използването на различни компютърни платформи, операционни системи, системи за управление на бази данни, системи за програмиране и т.н.
2. *Потребност от интеграционно съвместяване на компонентите на информационната система.* Очевидно, че най-естественият механизъм за организация на сложна информационна система е нейното йерархично вложено построяване. По-сложните функционално-ориентирани компоненти се изграждат от по-прости компоненти, които могат да се проектират и да се разработят независимо (което поражда неоднородността).
3. *Реинженеринг на системата.* След създаването на началния вариант на информационната система неизбежно следва процеса на нейното непрекъснато преправяне (реинженеринг), обусловен от развитието и измененията на съответстващите бизнес процеси на корпорацията. Реконструкцията на системата не бива да е кардинална. Всички компоненти, не засегнати он процеса на реинженеринг, трябва да запазват своята работоспособност.
4. *Решаването на проблема за наследяването (legacy) на системите.* Всяка компютърна система (счита се, че това не е валидно за отворените системи в настоящето им възприятие) с времето стават бреме за корпорацията. Постоянно (и колкото по-рано, толкова по-добре) се налага да се решава задачата за вграждането на остарели информационни компоненти в системата, вече основана на нова технология. Необходимо е тази задача да бъде разрешима, т.е. компонентите на наследените системи да запазват своята оперативна съвместимост.
5. *Повторно използвани (reusable) ресурси.* Технологията за разработването на информационните системи трябва да способства за използването на вече съществуващите компоненти, което в крайна сметка трябва да доведе от екстензивен ръчен програмистки труд към интензивни методи за конструиране, ориентирани към конкретна област за приложение на информационната система.
6. *Удължаване на жизнения цикъл на информационната система.* Колкото по-дълго живее и носи ползи информационната система, толкова това е по-изгодно на корпорацията. Естествено, че за тази цел трябва да съществува възможност за добавяне на компоненти в нея, проектирани и разработени от друга технология.

Като правило при разпределените изчислителни мрежи се налага да се интегрират неоднородни БД. Това в значителна степен усложнява реализацията. Освен проблемите на интеграцията се налага да се решават всички проблеми, присъщи на разпределените СУБД: управление на глобалните транзакции, мрежова оптимизация на заявките и т.н.

Най-много за създаването на съответстваща технология е допринесъл международният консорциум OMG, публикувал редица документи, в които се специфицира архитектурата и инструменталните средства за поддръжка на разпределени информационни системи, интегрирани на базата на общ обектно-ориентиран подход.