Dr. Nehéz Károly Róbert

Szoftverintegrációs stratégiák

bevezetés

Mi a szoftverintegráció?

Definíció

A *szoftver integráció* olyan fejlesztési folyamat, amely során különálló szoftverrendszereket – alkalmazásokat, komponenseket - kapcsolnak össze annak érdekében, hogy azok együttműködjenek és új, egységes rendszert alkossanak.

Fázisai

1. Igényfelmérés és tervezés
   - **igényfelmérés**: felhasználói igények és üzleti követelmények meghatározása
   - **tervezés**: az integrációs stratégia kidolgozása, rendszer architektúra és integrációs pontok megtervezése
2. Követelmények elemzése és specifikáció
   - **követelmények elemzése**: A szükséges funkciók és a meglévő rendszerek képességeinek részletes elemzése
   - **specifikáció**: az interfészek és adatcsere formátumok meghatározása
3. Fejlesztés és implementáció
   - **fejlesztés**: integrációs kód fejlesztése (a meglévő rendszerekhez)
   - **implementáció**: az új komponensek bevezetése és a meglévő rendszerek módosítása
4. Tesztelés, validálás
5. Karbantartás és támogatás

Legacy systems – Örökölt rendszerek

Definíció

A "*legacy system*" kifejezés olyan informatikai rendszerekre utal, amelyek régebbi technológiát használnak (akár elavultak), de még mindig aktívan működnek és fontos szerepet játszanak egy szervezet mindennapi működésében..

Miért használnak legacy rendszereket?

Hosszú élettartam és stabilitás: Sok *legacy* rendszer nagyon hosszú ideig működött megbízhatóan. Ha valami jól működik, és a szervezet számára kritikus fontosságú, akkor nincs nyomós ok arra, hogy lecseréljék.

- **Költség**: egy teljes rendszer lecserélése nagyon drága lehet.
- **Komplexitás**: a *legacy* rendszerek gyakran mélyen integráltak a szervezet más folyamataiba és rendszereibe.
- **Kockázatvállalás**: nem vállalják fel a lecserélés kockázatát

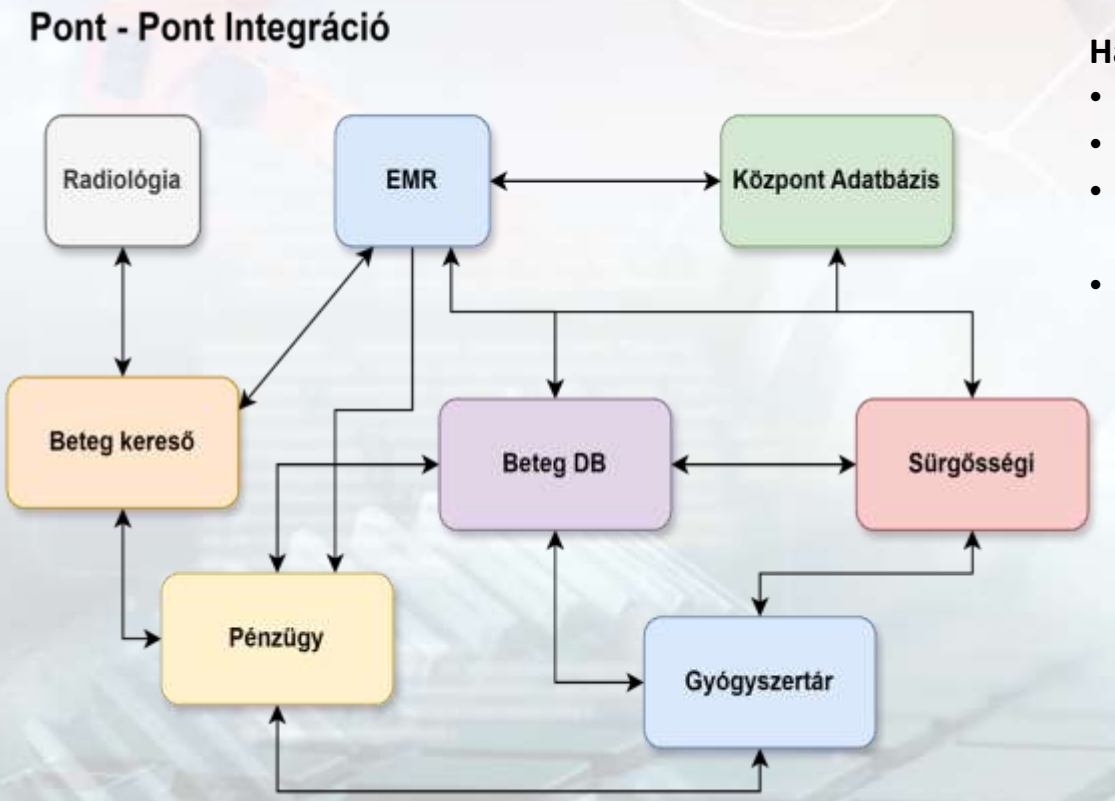
Miért nem cserélik őket?

- Költség és források hiánya: sokszor nem gazdaságos a csere – nem tudják becsülni a kockázatot (szigorú security szabályok)
- Üzleti folyamatok megzavarása: Egy új rendszer bevezetése jelentős zavarokat okozhat a napi működésben. A szervezetek sokszor inkább a stabilitást helyezik előtérbe. (pl. bank)

Megoldás

- **Fokozatos migráció**: A teljes rendszer lecserélése helyett fokozatosan migrálunk új rendszerekre, egy-egy részegység vagy funkciót cserélve.
- **Outsourcing**: a legacy rendszerek karbantartását és üzemeltetését külső szolgáltatóknak adják, így csökken a belső költség és a szakértői igény.

Integrációs stratégiák áttekintése



Pont – Pont kapcsolat

- A komponensek közvetlenül kapcsolódnak egymáshoz, általában fájlátvitel vagy közvetlen adatbázis eléréssel
- Nincs köztes réteg, ezért a kommunikáció gyors
- Kezdetben könnyű bevezetni

Hátrányai - kihívásai

- Nehezen skálázható
- A későbbi bővítés bonyolult lehet
- A kapcsolatok száma exponenciálisan növekszik -> n(n-1)/2 kapcsolat
- Törékeny, mert a rendszer hibákat nehéz monitorozni

Integrációs stratégiák áttekintése

Middleware alapú Integráció

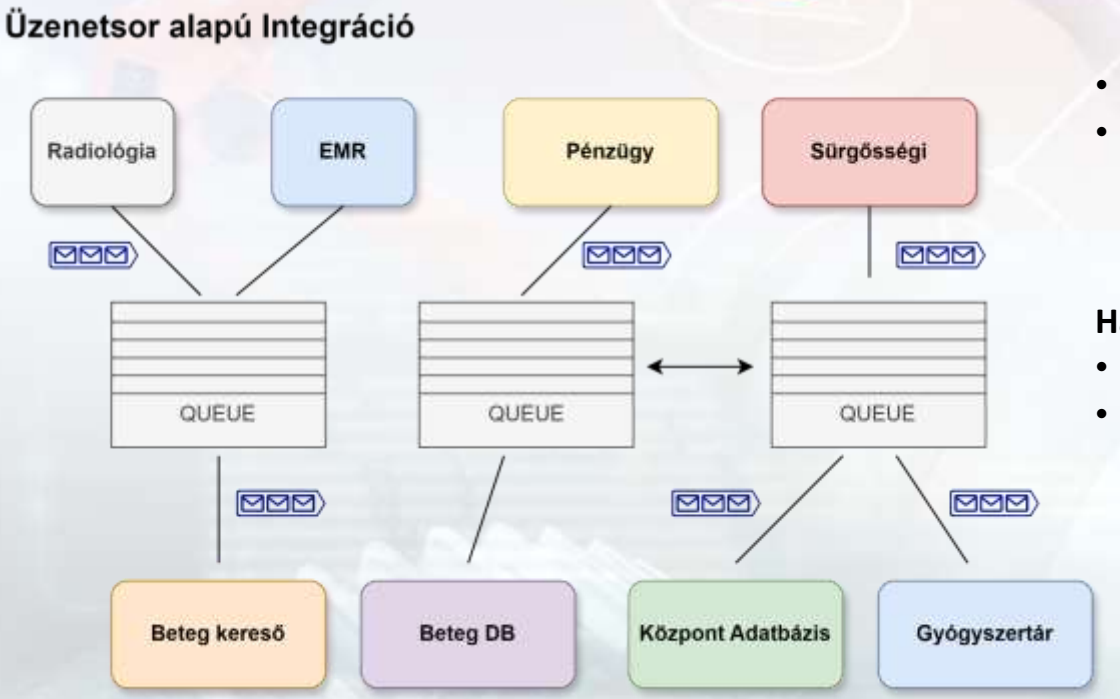
Klasszikus Middleware

- A komponensek nem közvetlenül kapcsolódnak egymáshoz, hanem egy központi közvetítőn keresztül (API gateway, Alkalmazás szerver, ESB)
- A köztes réteg kezeli a különböző protokollokat
- Monitoring lehetőségek: kommunikáció, központi felügyelet
- Jobban skálázható
- Központi funkciók: jogosultságok, tranzakciók kezelése

Hátrányai - kihívásai

- Komplexitás: a rendszerfejlesztés költsége magas
- Monolitikus rendszer – egy központi szerver – sok kliens

Integrációs stratégiák áttekintése



Üzenetsor alapú integráció

- A komponensek nem közvetlenül kapcsolódnak egymáshoz, hanem üzenetsorokon keresztül
- Az üzenetek aszinkron módon kerülnek feldolgozásra
- Monitoring lehetőségek: speciális üzenetsorok használata. DLQ
- Legjobban skálázható
- Felhőrendszerekben „természetes módon” használható

Hátrányai - kihívásai

- Komplexitás: a rendszerfejlesztés költsége magas
- Nagy szakértelmet, tervezést igényel

Mi a szoftverintegráció?

Adatmegosztás
Az integráció egyszerű megközelítése az adatmegosztás

Az adatmegosztás alapú integráció célja az adatok átadása és megosztása a rendszerek között. Ez lehetővé teszi, hogy az egyes rendszerek hozzáférjenek és felhasználják a más rendszerekben tárolt adatokat. Az adatmegosztásnak több formája létezik:

1.) Adatmigráció
Az adatok átvitele egyik rendszerből a másikba, általában egyszeri alkalommal, például egy rendszerfrissítés vagy rendszercsere során.

2.) Adatszinkronizáció
Az adatok folyamatos frissítése és szinkronizálása a különböző rendszerek között, hogy az adatok minden rendszerben naprakészek legyenek.

Adatmegosztó szolgáltatások (Data sharing services)
Olyan szolgáltatások, amelyek lehetővé teszik az adatok valós idejű megosztását és hozzáférését különböző rendszerek között

Fájl alapú adatmegosztás

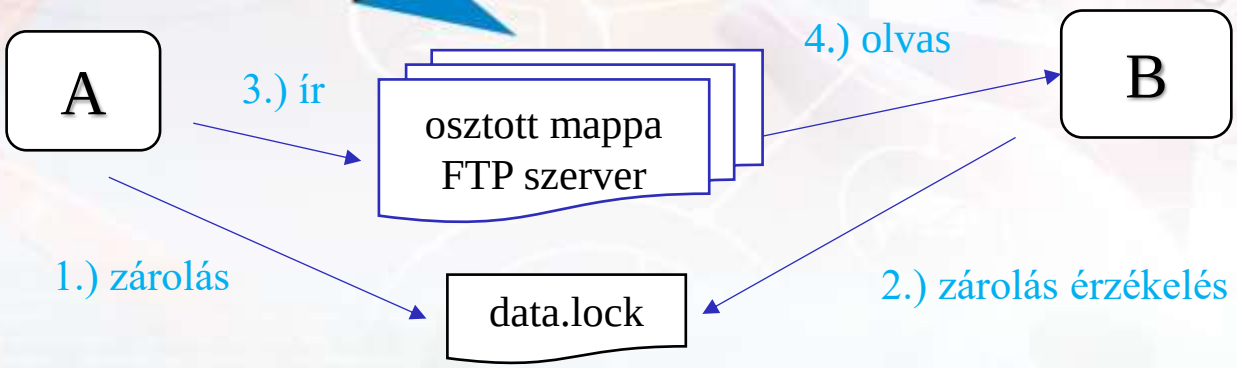
Az adatok megosztásának legalapvetőbb módszere

- Az egyik alkalmazás adatokat ír, a másik pedig adatokat olvas ugyanabból az állományból.
  - az adatállományok központi tárolón - osztott mappában (pl. NFS) vagy (S)FTP szerveren – helyezkednek el
  - az információ folyam egyirányú. A -> B

- Adatok kódolása: a legtöbb file alapú megosztás szöveges állományokat használ
  - legelterjedtebb a sima szöveges és XML (újabban JSON) alapú kommunikáció
  - nyers szöveg: fix hosszúságú rekordok és változó hosszúságú rekordok (számlázó, pénzügyi rendszerek)
  - a változó hosszúságúaknál kell valamilyen elválasztójel az adatok elválasztásához. A legismertebb ilyen módszer a CSV (comma separated values)

Fájl alapú adatmegosztás

Példa: adott egy ‚A' rendszer, ami adatokat olvas egy fájlból és a ‚B' rendszerbe továbbítja, létrehozhat egy állapotfájlt, amivel nyomon követi, hogy mely adatok lettek már feldolgozva.



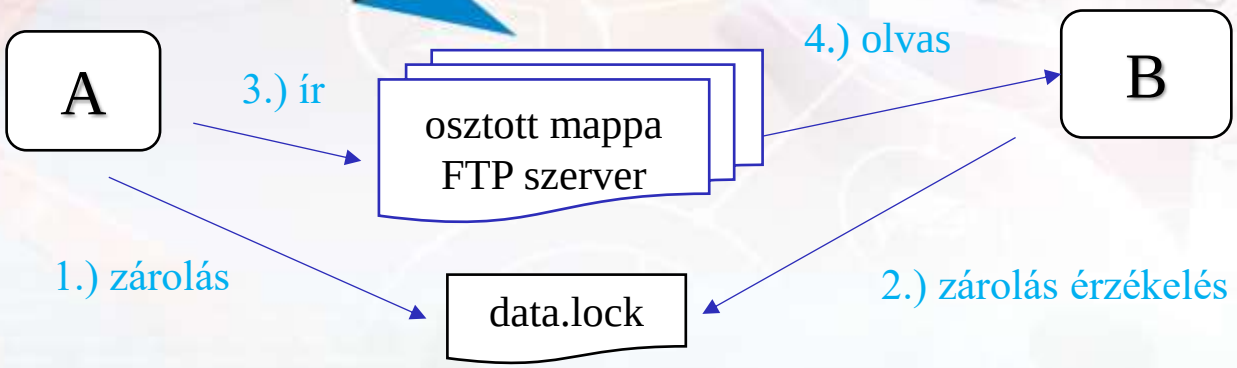
Állapotfájlok: állapotfájlokat lehet használni az adatok feldolgozási állapotának nyomon követésére. Ezek a fájlok tartalmazhatják a feldolgozás aktuális állapotát, például „in progress", „completed", „failed" stb.

1.) *Lock* fájl (zárolás) létrehozása: ‚A' elkezd dolgozni egy adatfájlon és létrehoz egy *lock* fájlt, például *data.lock*.
2.) ‚A' létrehozza az adatfájlt, miközben a *data.lock* létezik
‚B' megpróbál hozzáférni az adatfájlhoz, de látja, hogy a .lock fájl létezik, így várakozik.
3.) ‚A' befejezte a feldolgozást, törli a *data.lock* fájlt
4.) ‚B' észleli, hogy a *data.lock* fájl eltűnt (törlődött) és elkezdheti a saját feldolgozását

Ez a módszer biztosítja, hogy az adatfájlt egyszerre csak egy rendszer dolgozza fel, megelőzve az adatütközéseket és inkonzisztenciákat. A *lock* fájlok alkalmazása egyszerű és hatékony módszer a folyamatok szinkronizálására és koordinálására az integráció során.

Fájl alapú adatmegosztás

Példa: adott egy ‚A' rendszer, ami adatokat olvas egy fájlból és a ‚B' rendszerbe továbbítja, létrehozhat egy állapotfájlt, amivel nyomon követi, hogy mely adatok lettek már feldolgozva.

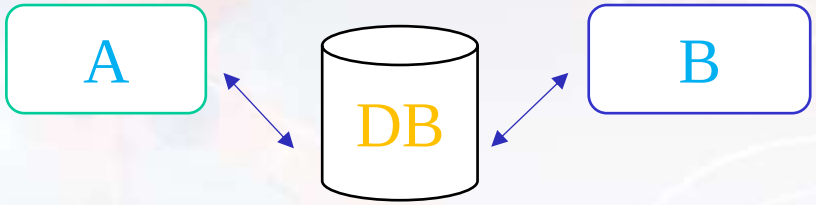


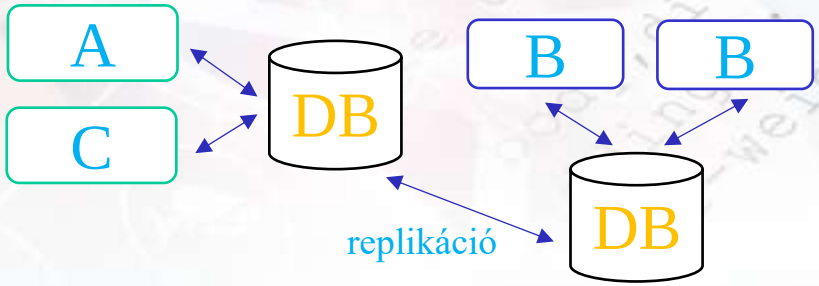
Ez a módszer máig is a legelterjedtebb, és a legtöbb hátránnyal rendelkezik:

- A megosztás nem valós idejű (real-time). Általában napi, heti, havi adatcseréhez érdemes használni. Ha a ciklusok között adatmódosítást végeznek, pl egy ügyfélnek megváltozik a címe, elképzelhető hogy a számlanyomtató alkalmazás még a régi címre küldi a számlát, mert csak később értesül a változásról.
- Megbízhatatlan lehet ha nagyszámú állomány átvitele szükséges. (*rsync* azért segíthet!)
- A sikeres integrációhoz mindkét alkalmazás fejlesztőjének (általában) ismernie kell:
  - az állomány formátumát, az állományok elnevezési konvencióit, az állományok helyét
  - hogyan történik az állományok törlése?
  - lock-olási mechanizmus alkalmazása
  - meg kell egyezni a fájl-átviteli módszerben

Adatbázis alapú adatmegosztás

Példa: E-kereskedelmi Platform és Raktárkezelő Rendszer: az e-kereskedelmi platform közvetlenül integrálható a raktárkezelő rendszer adatbázisával, hogy valós idejű készletinformációk álljanak rendelkezésre.





Az adatbázis alapú integráció egy módszer, amely lehetővé teszi az adatok megosztását és szinkronizálását különböző rendszerek között közvetlenül az adatbázisokon keresztül. Ebben a megközelítésben a különböző alkalmazások és rendszerek egy közös adatbázist vagy adatbázis-replikációt használnak az adatok eléréséhez és kezeléséhez.
Hasonló a fájl alapú megoldáshoz:

- platformfüggetlen kapcsolódási lehetőség: pl. JDBC, ODBC
- azonos komponensekből többet is használhatunk
  - szinkronizációs probléma: ki fogja feldolgozni a soron következő rekordot?
  - adatgyűjtésnél viszont ideális megoldás lehet
- nem valós idejű – ha egy alkalmazás ír az adatbázisba, a másik automatikusan nem kap értesítést
  - DB notification: PostgreSQL – NOTIFY és LISTEN mechanizmusok
  - triggerek, polling mechanizmusok alkalmazhatók
- biztonság: jól specifikált jogosultságok szükségesek (táblaelérések, engedélyezett műveletek). Sok fejlesztő nem „enged betekintést” a táblák struktúrájába.
- nincsenek jól definiált interfészek – ez csak adatkapcsolat

Socket alapú integráció

A socket alapú integrációnál, socketek segítségével történik az adatcsere.

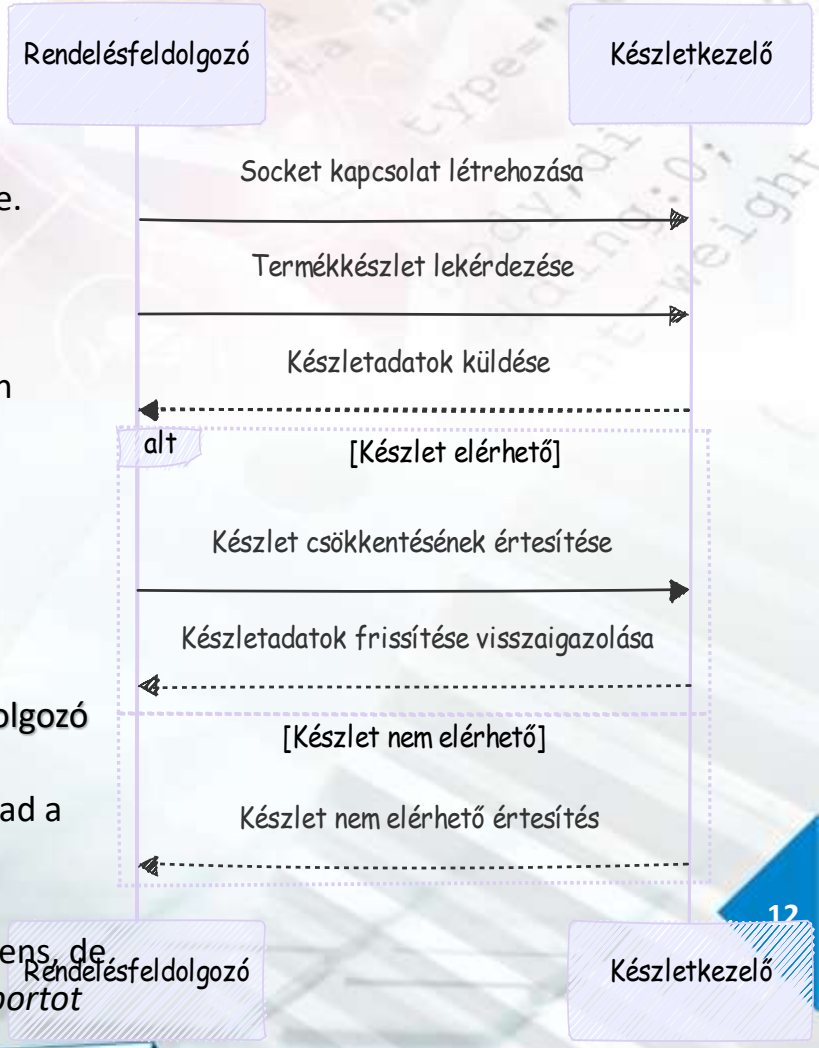
- Emlékeztető: a TCP/UDP socket nem más, mint az IP *cím* és a *port* kombinációja.
  - Például: tcp://193.6.4.187:8080
- Operációrendszer szinten támogatott, minden programozási nyelven elérhető
  - HTML5-ben websocket is van!

- A kommunikáció:
  - valós idejű, alacsony szintű

- A **Készletező** megnyit egy port-ot, pl. 8080 és figyel. A **Rendelésfeldolgozó** becsatlakozik és lekérdezi a termékeket. De nem zárjuk ki egy másik rendelésfeldolgozó csatlakozását sem. A kapcsolat (port) nyitva marad a kommunikáció teljes időtartama alatt.

- A **Készletező** a kiszolgáló szerepében van, a **Rendelésfeldolgozó** a kliens, de ez már *nem merev szerepkör*. A **Rendelésfeldolgozó** is nyithat saját *portot* és kiszolgálóként is viselkedhet.

Példa: a rendelésfeldolgozó lekérdezi a termékkészletet, kiválasztja a megrendelt terméket és frissíti a készletet.



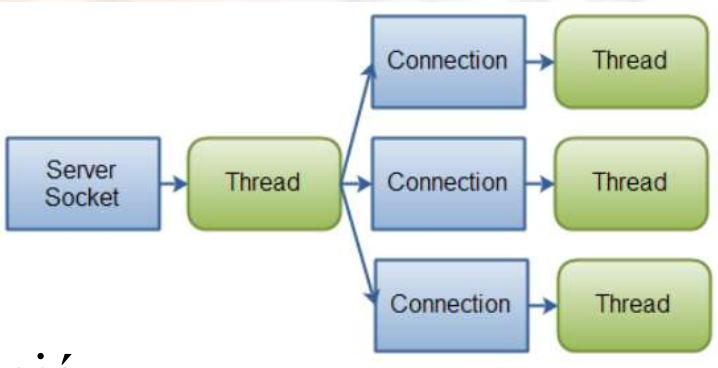
Socket alapú integráció

Java implementáció (elvben)

```java
public class Server {
    ...

    void run() {
        providerSocket = new ServerSocket(8080);

        connection = providerSocket.accept();

        in = new ObjectInputStream(connection.getInputStream());

        message = (String) in.readObject();
        processMessage(message);

        in.close();
        providerSocket.close();
    }

    public static void main(String args[]) {
        Server server = new Server();
        while (true) {
            server.run();
        }
    }
}
```

```java
public class Client {
    ...

    void run() {
        requestSocket = new Socket("service.com", 8080);

        out = new ObjectOutputStream(requestSocket.getOutputStream());
        in = new ObjectInputStream(requestSocket.getInputStream());

        sendMessage("Hello szerver");

        message = (String) in.readObject();

        processMessage(message);

        in.close();
        out.close();
        requestSocket.close();
    }

    void sendMessage(String msg) {
        out.writeObject(msg);
        out.flush();
    }

    public static void main(String args[]) {
        Client client = new Client();
        client.run();
    }
}
```

Socket alapú integráció



Többszálú socket szerver – elvi implementáció

```
1  public class Server implements Runnable {
2
3      private Socket connect;
4
5      public Server(Socket c) {
6          connect = c;
7      }
8
9      public static void main(String[] args) {
10         ServerSocket serverConnect = new ServerSocket(8080);
11
12         while (true) {
13             Server myServer = new Server(serverConnect.accept());
14
15             Thread thread = new Thread(myServer);
16             thread.start();
17         }
18     }
```

Socket alapú integráció

Blokkolt socket – több szálon

- Hagyományos megközelítés, az esetek többségében megfelelő
- Több ezer kliens esetén viszont több ezer szál jön létre!
  - modern hardver esetén sem optimális
  - a kliensek válaszideje nagymértékben lecsökken

- Nem-blokkolt socket alkalmazása szükséges, ahol:
  - minden kliens azonos „porton marad”, a megkülönböztetésük tokenekkel lehetséges
  - a szerver oldal sokkal komplikáltabb
  - *Node-js* alkalmazása érdemes
    - a gyakorlaton lesz példa rá!
    - a gyakorlaton láthatunk egyszerű példát UDP-re is

Socket alapú integráció

Előnyök

- a leggyorsabb adatcserét teszi lehetővé, amely különösen fontos olyan alkalmazásoknál, ahol az azonnali válaszidő kritikus
  - valójában - a háttérben - erre épül a fejlett integrációs megoldások mindegyike
  - kihasználhatjuk az operációs rendszerek finombeállításait

- az operációs rendszer támogatás miatt nem kell egyéb függőségeket használni
  - kis méretű maradhat a kódbázis
  - csökkent a közvetítő rétegek száma

- Példák
  - chat alkalmazások
  - online játékok
  - pénzügyi rendszerek

Socket alapú integráció

Hátrányok

- az adatok megosztásán kívül a funkciók megosztását csak közvetetten támogatja
  - *gyakorlaton*: egyszerű FTP szerver funkciók implementálása
- a *socket* alacsony szintű API támogatása miatt – a programozása nem könnyű feladat
- nincs szabványos interfész a kommunikáció leírására:
  - az integráció fejlesztése során érdemes UML szekvencia diagrammokat rajzolni – e nélkül a kommunikáció nem lesz átlátható
  - a hibakezelés, nyomkövetés komoly kihívást jelent a fejlesztés közben
- a kliens és a szerver nem fejleszthető anélkül, hogy az ellenoldalt nem implementáljuk (*mock* implementációk)
- a fejlesztési idő nagyságrendekkel lassabb, mint a magasabb szintű integrációknál (JAX-WS, JAX-RS)
- összetett adatstruktúrák (vagy numerikus adatok) esetén a szerializációt meg kell valósítani
  - Ha a kliens és a szerver különböző OS-en fut (és a nyelv sem azonos)

Modern pont-pont adatintegráció

Protobuf – protokol buffer

- A Google által fejlesztett strukturált adat szerializációs eljárás. Eredetileg a felhőszolgáltatások egyes komponensei közti kommunikációban volt sikeres (Microsoft Azure is átvette)
  - Alkalmazásai:
    - Nagyvállalti rendszerek
      - **Netflix**: mikroszolgáltatási architektúrájában használja, hogy a különböző szolgáltatások közötti kommunikáció gyors és hatékony legyen
      - **Uber**: a valós idejű helymeghatározási adatok továbbítására és feldolgozására használják
      - **WhatsApp**: az üzenetek és más adatok hatékony szerializációra a szerverek és a mobilalkalmazások között
      - **Spotify**: a zenei adatbázis és a felhasználói adatok kezelésére, hogy a mobilalkalmazások gyorsan és hatékonyan férjenek hozzá az adatokhoz
    - Banki rendszerek, tőzsdei kereskedési rendszerek

Protokol Buffer

A gyors adatcsere követelménye miatt a hagyományos socket kommunikációt bővítették egy *interfész leíróval*

- Interfész leíró (először a régi Corba rendszerben jelent meg)
- Összetett adatszerkezetek platformfüggetlen megadására szolgál
  - Egyedi fordítót (compiler) alkalmaz az interfészek létrehozására
  - Viszonylag könnyű a programozása
  - Verziókövetés: az adatok eltérő verzióit is kezeli

```
1  syntax = "proto3";
2
3  message Book {
4      int32 id = 1;
5      string title = 2;
6      string author = 3;
7      float price = 4;
8  }
9
10 message Books {
11     repeated Book books = 1;
12 }
```

pozíció index

Ez miért kell?

```
1  message Person {
2    optional string name = 1;
3    optional int32 id = 2;
4    optional string email = 3;
5
6    enum PhoneType {
7      HOME = 2;
8      WORK = 3;
9    }
10
11   message PhoneNumber {
12     optional string number = 1;
13     optional PhoneType type = 2 [default = HOME];
14   }
15
16   repeated PhoneNumber phones = 4;
17 }
18
19 message AddressBook {
20   repeated Person people = 1;
21 }
```

Protokol Buffer

Szerver

```python
import socket
import book_pb2
import create_books as c

# protoc/bin/protoc --python_out=./ book.proto
# pip3 install --upgrade protobuf

books = c.create_books()

book_store = book_pb2.Books()
for book in books:
    book_store.books.append(book)

bytes_to_send = book_store.SerializeToString()

#TCP socket server
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((socket.gethostname(), 4100))
s.listen(10)

while True:
    client_socket, address = s.accept()
    print(f"server> Connection from {address} has been established!\n")

    client_socket.send(bytes_to_send)
    print(f"server> Message sent: {bytes_to_send}\n")

    msg = client_socket.recv(1024)
    print(f"client> {msg}\n")
    client_socket.close()

    if msg == b'bye':
        break

s.close()
```

Kliens

```python
import socket
import book_pb2
from google.protobuf.json_format import MessageToJson
import json

#TCP socket client
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((socket.gethostname(), 4100))

msg = s.recv(1024)
print(f"server> {msg}\n")

s.sendall(b'bye')
print(f"client> Message sent: {b'bye'}\n")

s.close()

books = book_pb2.Books()
books.ParseFromString(msg)

json_obj = MessageToJson(books)
```

```python
import book_pb2

def create_books():
    books = []

    books.append(book_pb2.Book())
    books[0].id = 1
    books[0].title = "Solaris"
    books[0].author = "Stanislaw Lem"
    books[0].price = 7.54
```

Összefoglalás

- Alapfogalmak
- Három fő stratégia áttekintése:
  - Pont-pont, Middleware, Üzenetsorok alapú integrációs módszerek
- Adatmegosztási stratégiák
  - File, DB alapú
- TCP/UDP Socket alapú adatmegosztás
  - Blokkolt socket szerver és kliens
  - Nem-blokkolt socket szerver és kliens a gyakorlaton
  - UDP – a gyakorlaton
- Protobuf alapú adatmegosztás

KÖSZÖNÖM A MEGTISZTELŐ FIGYELMET!