Objektum-orientált megvalósítások

Előadásvázlat „Szolgáltatásbiztonságra tervezés” tárgyból

Majzik István
BME Méréstechnika és Információs Rendszerek Tanszék

Tartalomjegyzék:

1 Az alapfeladatok áttekintése 2

2 Mikrokernel megoldások 3

3 Köztesréteg megoldások 3
3.1 Elosztott OO köztesréteg konfigurációk 4

4 Öröklésen alapuló megoldások 5

5 Reflektív programozáson alapuló megoldások 6

6 Aspektus-orientált programozáson alapuló megoldások 7
6.1 Bevezető 7
6.2 Az aspektus-orientált programozás (aspect oriented programming, AOP) 7
6.3 AspectJ megvalósítás: A Java kiterjesztése 8

7 Hibatűrő OO rendszerek tipikus architektúrája 10
7.1 Alapelvek 10
7.2 Alkalmazáshoz kötődő objektumok 10
7.3 A hibatűrő infrastruktúra (Fault Tolerance Infrastructure) 11
7.4 Mechanizmusok összefoglalása 13

1 Az alapfeladatok áttekintése

Hibamodell: A környezet által könnyen detektálható hibamódok (pl. ”I am alive” üzenetekkel), amik nagy hibafedésű lokális hibadetektálást feltételeznek:

- *Fail silent* működés: Hibás üzenet vagy információ nem terjed tovább a hiba (detektálása) után
- *Omission* (elnyelés): Kimaradhatnak üzenetek (de a vett üzenet mindig hibamentes)

Hibatűrés módszerei:

- Fizikai hibák ellen: Objektumok replikálása különböző szerver (host) gépeken.
- Szoftver (tervezési) hibák ellen: Az objektum replikákban eltérő tervezés (*design diversity*):
  - Metódus szinten: Eltérő módon megvalósított metódusok
  - Objektum szinten: Objektum csoport némiképpen eltérő belső állapottal (*data diversity*, amennyiben a hibák várhatóan erősen adatfüggők, viszont kis változások a kimenetben elfogadhatók)
  - Osztály szinten: Függetlenül tervezett és implementált osztályok (teljesen OO kompatibilis),
    pl. egy absztrakt osztály eltérő leszármazottai, vagy teljesen eltérő osztályok
  - Rendszer szinten: Ld. N-verziós programozás a teljes rendszerre.

Objektum életciklus és a hibatűréshez szükséges többletfeladatok kapcsolata:

Objektum életciklus	Többletfeladatok hibatűréshez
Inicializálás (konstruktor)	Replikák létrehozása
	Replika csoport kezelése
Metódushívás	Kérések sorrendezése
	Többszörös kérések szűrése
Hívott metódus indulása	Kérés sorszám ellenőrzés
	Kérés tárolása (üzenettár)
Hívott metódus befejeződése	Objektum állapot mentése
	Szinkronizáció replikákkal
Törlés (destruktor)	Kilépés a replika csoportból
Elsődleges objektum példány crash hiba	Helyreállítás
	Újrakonfigurálás

Alapvető szolgáltatások a replika alapú hibatűréshez (objektum szintű redundanciához):

- *Inter-Replica Protocol* (IRP): Állapot-szinkronizáció az elsődleges és a tartalékok között, az állapotmentés és újraindítás kiváltása
- *Group Communication Service* (GCS): Csoportkommunikáció az elsődleges(ek) és a tartalékok között; teljes (és oksági) üzenet sorrendezés megvalósítása.

Ehhez szükséges alacsonyabb szintű (tipikusan köztesréteg) szolgáltatások:

- Set: objektumok egy halmazának elérése egy referenciával
- Multicast: a halmazreferencia alapján az elérés a halmaz minden tagjához eljut
- Ordering: az elérések sorrendezése minden objektumhoz

- Reliable messaging: minden elérhető (nem kiesett) objektumhoz eljut a kérés
- Atomic: minden kérés vagy eljut minden objektumhoz, vagy semmi hatása nem lesz
- Membership: tagsági kép at elérhető illetve kiesett objektumokról

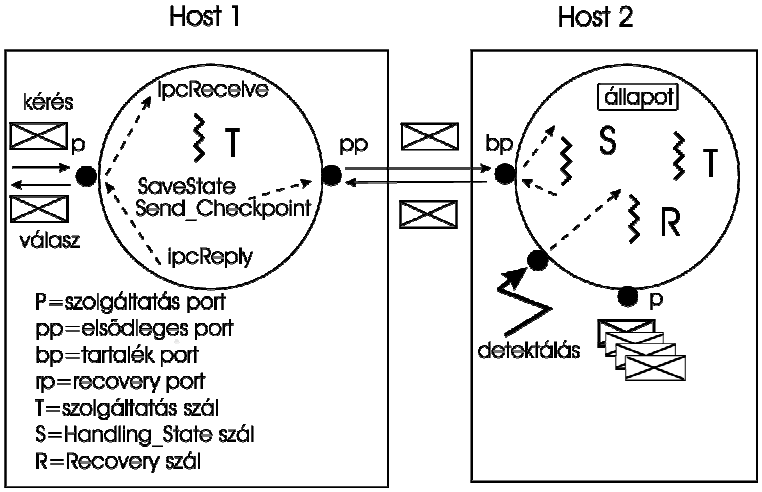
2 Mikrokernel megoldások

Alapelvek (pl. Chorus kernel):

- A mikrokernel biztosítja az alap OS funkciókat (memóriakezelés, processz kezelés, szinkronizáció, kommunikáció, ...).
- Az objektumok állapotot és konkurens végrehajtási szálakat tartalmaznak.
- Portok kezelése az üzenetek fogadásához/küldéséhez.

Hibatűrést támogató funkciók

- Portok dinamikus hozzárendelése objektumokhoz (így váltani lehet elsődleges-tartalék között)
- Globális objektum azonosítás, automatikus lokalizáció
- Szinkronizáció a szálak befejeződésekor lehetséges (amikor nincs belső konkurens működés)



Mikrokernelre épülő redundáns objektumok

Működés aktív és tartalék objektumokkal:

- Aktív objektum normál működése esetén:
  - portján a kérést fogadja
  - a számítás után állapotát a tartalékkal szinkronizálja (SaveState metódus menti, SendCheckpoint metódus átküldi, HandlingState szál átveszi)
- Az aktív objektum detektált hibája esetén:
  - a tartalék objektum recovery portján a hibadetektor jelez, ekkor Recovery szál indul
  - szerep átvétele (aktív mód), új replika klónozása
  - elsődleges port átvétele (átnevezés), kérések fogadása

3 Köztesréteg megoldások

Alapfeladatok: Objektum replikáció támogatása köztesréteg segítséggel

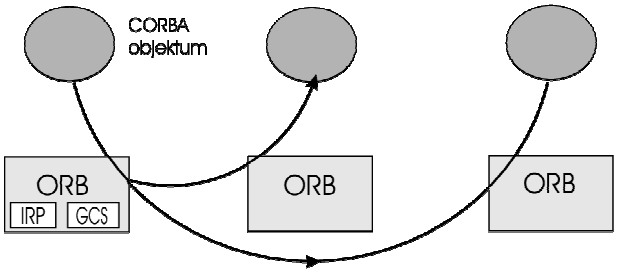
- Inter-Replica Protocol (IRP): Állapot-szinkronizáció az elsődleges és a tartalékok között, az állapotmentés és újraindítás kiváltása

- Group Communication Service (GCS): Csoportkommunikáció az elsődleges(ek) és a tartalékok között; teljes (és oksági) üzenet sorrendezés megvalósítása.

3.1 Elosztott OO köztesréteg konfigurációk

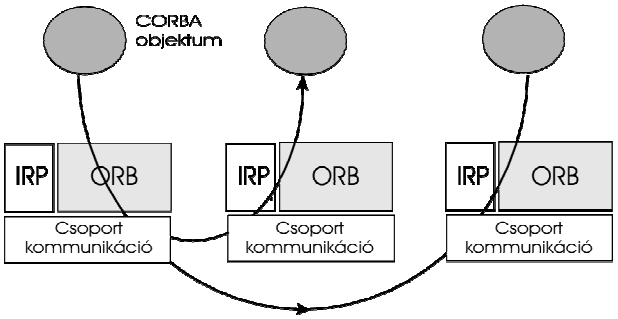
Példaként a CORBA ORB (Object Request Broker) példáján mutatjuk be a köztesrétegen alapuló hibatűrő (FT) architektúrák kialakításának lehetőségeit:

- **Integráció** (*integration*): az ORB módosítása a replika kezelés érdekében (pl. Orbix+Isis). Csoportkommunikáció és FT stratégiák az ORB-be ágyazva.



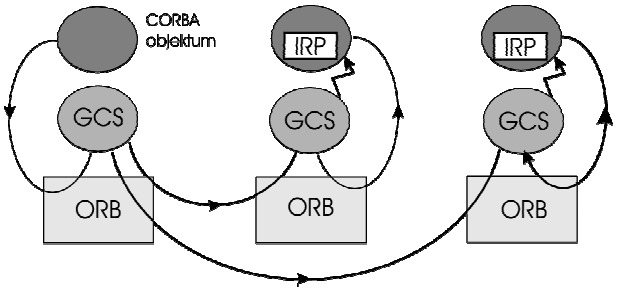
Integráció a köztesrétegbe

- **Beavatkozás** (*interception*): a CORBA üzenetek eltérítése (pl. Eternal). Csoportkezelő réteg (alrendszer) veszi át az üzeneteket és kezeli a replikákat.



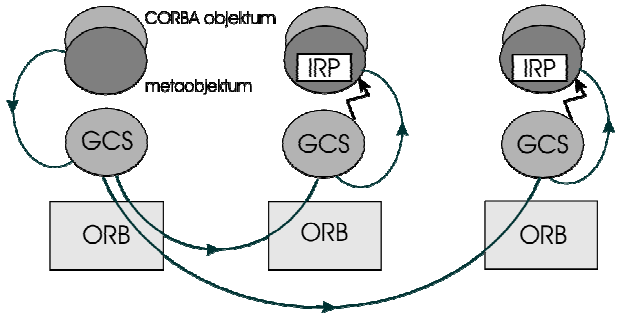
Beavatkozás: Az üzenetek eltérítése

- **Szolgáltatás** (*service*): CORBA szolgáltatást alakítanak ki a replikáció támogatására, ami az alkalmazás szintjéről érhető el (pl. OpenDreams).
  A CORBA üzeneteket egy csoportkommunikáció szolgáltatáshoz (GCS) kell irányítani, ennek feladata a replika kezelés (létrehozás, csatlakozás, leválás, multicast).



Szolgáltatás a csoportkommunikáció megvalósításához

- **Reflektivitás** (*reflective*): az FT-specifikus funkciókat az úgynevezett metaobjektumok szintjén kell elvégezni (ld. később).



Reflektivitáson alapuló megvalósítás

4 Öröklésen alapuló megoldások

Kialakítás:

- (Előredefiniált) öröklési struktúra, amely az alapvető funkciókat biztosítja:
  - állapotkezelés: mentés, visszaállítás
  - replika kezelés: állapotmentés elküldése, értesítés, szinkronizáció
  - konfiguráció kezelés: replika létrehozása, törlése, klónozása
- Objektumok a szükséges funkciókat az öröklési struktúrán keresztül veszik át
- Jól illeszkedik az OO világhoz (tervezési módszerek, nyelvek)
- A hibatűréshez szükséges alapszolgáltatásokat könyvtári osztályokban lehet megvalósítani pl. állapotkezelés, csoportkezelés (kommunikáció), tokenkezelés (szinkronizáció), tranzakciókezelés (atomiság)
  - nem transzparens, hatékonysága is az alkalmazótól függ
  - testreszabás/optimalizálás egyszerűbb

Példa: Az Arjuna rendszer (University of Newcastle-upon-Tyne, UK) osztályhierarchia részlete:

Osztály	Alosztály	Alosztály	Metódusok
State_Mgr			Activate, Deactivate, SaveState, RestoreState
	AtomicAction		Begin, End, Abort (Prepare, Commit)
	Lock_Mgr		Setlock, ReleaseLock
	AbstractRecord		
		ServerGroupRecord	Szerverek listája egy AtomicAction-ben
		LockRecord	Helyreállítás zár objektumokon
		RecoveryRecord	Helyreállítás objektumokon

5 Reflektív programozáson alapuló megoldások

Megközelítés, rendszerstruktúra:

- Reflektivitás: az a folyamat, amelynek során a rendszer magáról következtet, magát befolyásolja; eszközöket ad arra, hogy a rendszer belső mechanizmusait (struktúra és viselkedés) megfigyelni és befolyásolni tudjuk.
- Két szint, amelyek között az úgynevezett Metaobject Protocol teremt kapcsolatot:
  - Alapszint, ahol az alkalmazói objektumok vannak
  - Metaszint, ahol az alkalmazói objektumok viselkedését befolyásoló metaobjektumok vannak (minden alapszintű objektum egy metaobjektumhoz kötött)
- Metaobjektumok szerepe: viselkedés meghatározása az egyes mechanizmusok metaszintre való ”átirányítása” révén; innen lehetséges az alapfunkciókhoz való visszahívás. A metaobjektum hívott metódusai között tipikusan szerepelnek a következők:
  - Objektum létrehozáskor (konstruktor): Meta_StartUp()
  - Metódushíváskor: Meta_Call(), Meta_HandleMethodCall()
  - Attribútum hozzáféréskor: Meta_Read(), Meta_Assign(), Meta_HandleRead(), Meta_HandleAssign()
  - Objektum törléskor (destruktor): Meta_CleanUp()
- Hibatűrés: metaobjektumok határozzák meg az objektumok egyes funkcióinak viselkedését: átveszik az objektumoknak szóló üzeneteket, és a metaszintről valósítják meg a replikációs stratégiát, ami így transzparens lehet, jól elválasztva a funkcionális és nem-funkcionális követelményeket.

Nyelvi támogatás: a MetaObject Protocol nyelvi implementációja

- objektum - metaobjektum kapcsolat fordítási vagy futási időben
  - a metaobjektum magába foglalja az objektumot, statikus kötés van közöttük, egy egységként élnek
  - csak egy csonk köti össze az objektumot és a metaobjektumot, dinamikus köztük a kötés, két futásidejű egység létezik

Példa: Open C++

- előfeldolgozó létezik a reflektív programozás megvalósítására
- metódusonként illetve attribútumonként állítható a reflektív megvalósítás (//MOP reflect):

```
class MyClass {
  public: f();
  //MOP reflect:
  g();
  protected: int i;
  //MOP reflect:
  float x;
}
```

- a reflektivitást megvalósító metaobjektumok szintén örökléssel származnak, itt a technika fordítási időben meghatározott és nem teljesen átlátszó:

  ```
  //MOP reflect class MyClass : MyMetaClass;
  ```

  jelentése: `MyClass` osztály minden objektumát a `MyMetaClass` osztály egy-egy objektuma fogja meghatározni

- reflektivitás korlátozott: struktúra, öröklés nem változtatható a metaszinten

Példa: Passzív replikáció megoldása, az elosztott objektumok kezelése alapszinten

- felépítés: kliens, elsődleges és tartalék szerverek
- metaszint feladatai: szerver hívásait átvenni
    - elsődleges szerver: kliens kérés feldolgozása után állapotadatok (checkpoint) átküldése a tartaléknak
      tartalék figyelése (crash detektálás, pl. ”I am alive” üzenetekkel)
      tartalék hibája esetén új tartalék létrehozása és ehhez való csatlakozás
    - tartalék szerver: az elsődlegestől érkező állapotadatok fogadása és beállítása
      elsődleges szerver figyelése (crash detektálás, pl. ”I am alive” üzenetekkel)
      elsődleges szerver hibája esetén szerepét átvenni, új tartalék létrehozása és csatlakozás hozzá

6 Aspektus-orientált programozáson alapuló megoldások

6.1 Bevezető

Szoftverfejlesztés néhány általános problémája:

- *Szempontok* szétválasztása (separation of *concerns*):
    - produkciós: pl. üzleti logika, perzisztencia, biztonság
    - fejlesztési: pl. tesztelhetőség, adaptálhatóság, naplózás
    - hibakezelés, QoS: pl. hibadetektálás, redundanciakezelés, időbeliség
- Alapmegfigyelés: Szempontok és moduláris egységek határai nem esnek egybe
    - code tangling: egy modul több vonatkozásnak kíván eleget tenni
    - code scattering: egy vonatkozás sok modulban valósul meg
- Ebből eredő hatékonyságromlás:
    - követelmények követhetősége rossz
    - újrafelhasználhatóság korlátozott az átszövő megvalósítás miatt
    - nehezebb továbbfejlesztés
    - meglévő technikák csak részben oldják meg (ld. Visitor, Template tervezési minta)
- Célkitűzés: Adott szempontot megvalósító részek elkülönítése, kiemelése.

Megoldások:

- Statikus osztálystruktúra + öröklés: redundancia sémák megvalósítására használható (nem igazán átszövő)
- Reflektív architektúrák: alapszintű objektumok kiterjesztése metaszintű objektumokkal
- *Aspektus–orientált programozás*: nyelvi támogatás a problématér eltérő szempontú elemeinek moduláris egységekbe (aspektus konstrukciókba) szervezésére; majd ezek egybeszerkesztése egy aspektus-szövő (aspect weaver) segítségével.

6.2 Az aspektus-orientált programozás (aspect oriented programming, AOP)

AOP tervezés lépései:

- aspektusok dekompozíciója (átszövő aspektusok azonosítása)

- aspektusok implementációja (elkülönítve lehetséges, külön nyelveken)
- aspektusok összeállítása (AOP nyelvi szabályok alapján)

AOP elemei:

- alap (üzleti logika): komponens nyelv → komponens program
- járulékos szempontok: aspektus nyelv → aspektus program
- aspektus-szövő (aspect weaver) → teljes program (fordítható)
  futásidejű (run-time) vagy fordítási idejű (compile-time) lehet

AOP nyelvi elemek szerepe:

- mit kell tenni a beavatkozáshoz: *tanács* vagy *beavatkozás* (advice)
- hol kell beavatkozni: *vágási pont* vagy *eseményleíró* kifejezés (pointcut)
- hol történik a tényleges beavatkozás: *kapcsolódási pont* (join point):
  - egy-egy eseményleíró több kapcsolódási pontot is meghatározhat
  - a kapcsolódási pontot nem kell felkészíteni a beavatkozáshoz (meglévő kódhoz illeszthető).

6.3 AspectJ megvalósítás: A Java kiterjesztése

6.3.1 Kapcsolódási pont (joint point)

Aspektus interakciójának a helye

- statikus: programszöveghez kapcsolódik
- dinamikus: programfutáshoz kapcsolódik
  - metódushívás és végrehajtás
  - konstruktor hívás és végrehajtás
  - attribútum írás/olvasás
  - kivételkezelés

6.3.2 Vágási pont vagy eseményleíró kifejezés (pointcut)

Kapcsolódási pontok kijelölése és kontextus összegyűjtése

- Kapcsolódási pontok kijelölése: dinamikus kapcsolódási pontok halmazai, ezekből logikai műveletekkel képzett halmazok;
  - név alapú (name-based): explicit felsorolás
  - tulajdonság alapú (property-based): wildcards: metódusnév, paraméterek, visszatérési érték stb.
- Kontextus kijelölése: paraméterek (this, target, args)
- Absztrakt vágási pont: leszármazott aspektus definiálja
- Konkrét kulcsszavak a kapcsolódási pont megadására:
  - call(method), execution(method),
  - get(field), set(field),
  - handler(exception),
  - staticinitialization(class),
  - within(class), withincode(method),
  - cflow(pointcut), cflowbelow(pointcut),
  - this(object), target(object), args(object)
  - if(BooleanExpression)

- ||, &&, és ! operátorok vágási pontokon
- kontextus megadás: target(), this(), and args() pointcuts
- reflektív tulajdonság: speciális objektum, thisJoinPoint, tartalmaz információt a kapcsolódási pontról

6.3.3 Tanács vagy beavatkozás (advice)

Eseményleíró kifejezéshez rendelt, paraméterezhető, esemény előtt, után, közben végrehajtott metódus-szerű műveletek:

- before() advice: a kapcsolódási pont előtt fut le
- after() advice: a kapcsolódási pont után fut le, normál és kivételkezelés esetén is
- around() advice: a kapcsolódási pont helyett fut le
- A vágási pont és tanács együtt az *aspect weaver* egy szabályát határozza meg

Bevezetés (introduction): Új adattagok és metódusok definiálása az osztályokban (a csatlakozási pontot itt az osztály jelenti)

- osztály metódusának és attribútumának bevezetése
- osztály öröklődési hierarchiájának megváltoztatása
- kivételkezelés megváltoztatása

6.3.4 Aspektus (aspect)

Moduláris egység (osztályhoz hasonlóan: metódusok, konstruktor, inicializáló, eseményleíró és tanács alkotja);

- Fenti elemek valamint bármely osztályon belüli tag tetszőleges kombinációja
- Aspektusok kiterjeszthetik egymást, interfészeket valósíthatnak meg
- Aspektusok asszociációi (separált aspektus példány az objektumban, a cél objektumban, a vezérlési folyamban, vagy a vezérlési folyamban a vágási pontot kivéve)
- Egy-egy belépési ponthoz kapcsolódó aspektusok között precedencia relációk állnak fenn
- Aspektusok között öröklődési relációk állnak fenn
- Aspektusok csak azokkal az osztályokkal együtt értelmezhetőek, amelyeknek az átszövő kódját tartalmazzák (önállóan nem fordíthatók); mintegy változtatásokként értelmezhetők
- Specialitások:
    - wildcard használata az eseményleíró kifejezésben
    - absztrakt aspektusok: absztrakt eseményleíró deklarációk helyezhetők el, melyek definícióját (az adott osztályhoz igazítva) az aspektus egy leszármazottjában adjuk meg

6.3.5 Példák

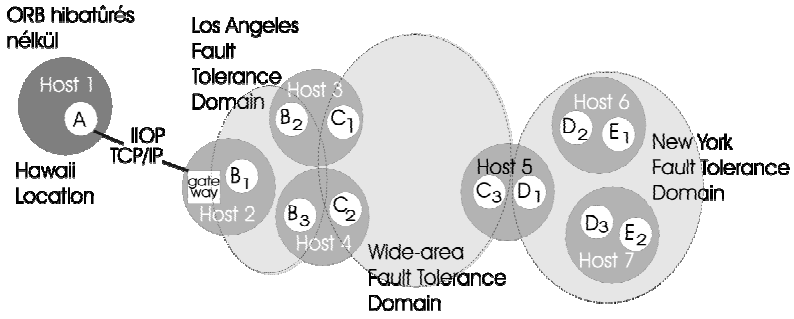
- Üzenettárolás és állapot-szinkronizáció aspektus-orientált megvalósítása
- Aktív redundáns szerverek aspektus-orientált megvalósítása

7 Hibatűrő OO rendszerek tipikus architektúrája

Mintapéldaként a hibatűrő CORBA (FT-CORBA) szabványos architektúráját tekintjük át.

7.1 Alapelvek

- Hibamodell: objektum crash;
  Nincs kezelve: kommunikációs hálózat particionálása, nem fail-silent működés (hibás válaszok, bizánci hibák), korrelált hibák
- Hibatűrés: Ne legyen egyszeres hibapont (single point of failure); objektum redundancia:
  - térbeli redundancia: objektum *replikáció* (host elhelyezkedéstől független);
  - időbeli redundancia: *többszörös hívások* a kliens által (ugyanahhoz a szerver objektumhoz, *at most once* szemantikával)
- Objektum replikák egy *objektumcsoportot* képeznek; ezek menedzselhetők (pl. passzív vagy aktív replikáció)
  - Referencia: IOGR (interoperable object group reference), ezt teszi közzé a replikált aktív objektum, a továbbiakban a normál objektumhoz hasonlóan működik (kérések fogadása és kiszolgálása ez alapján)
  - Rejtett belső koordináció: replikakezelés (aktív, passzív) illetve hibakezelés
- Hibatűrő tartomány (fault tolerance domain, FTD): több host, több objektumcsoport összefogása
  - Egy FTD-n belül minden objektumcsoportot egy replikáció menedzser (replication manager) kezel
  - Hibatűrés tulajdonságai objektumcsoportra vagy FTD-re is állíthatók, ezek dinamikusan változhatnak



A CORBA hibatűrő tartomány

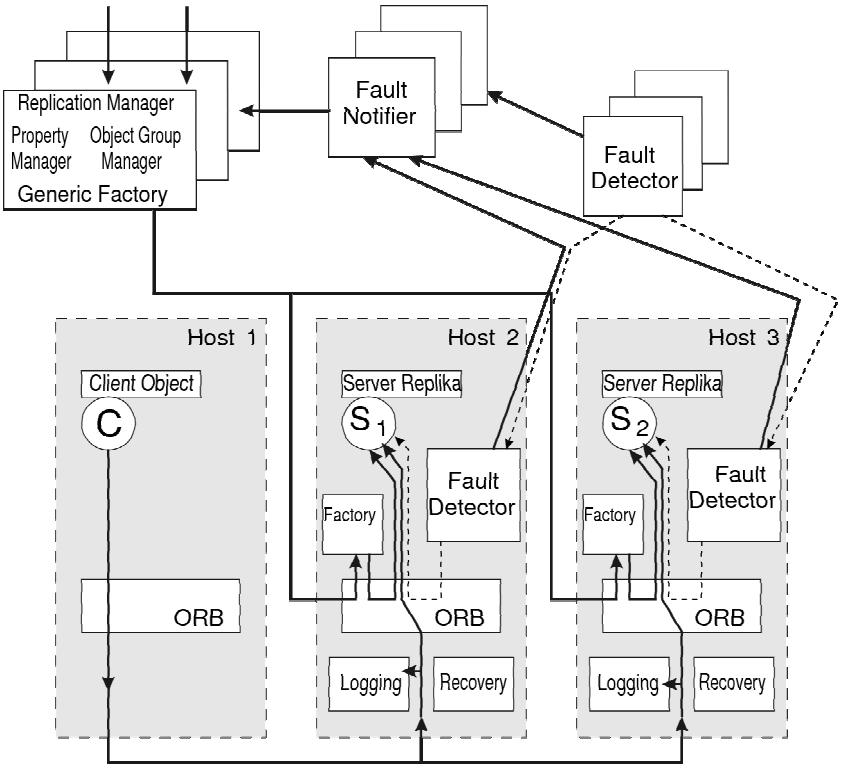
7.2 Alkalmazáshoz kötődő objektumok

- *Factory* objektum minden hoston (nem többszörözött): objektumok létrehozására
- *Fault Detector* objektumok minden hoston (nem többszörözött): lokális hibadetektálás; ehhez az alkalmazás objektumok megvalósítják a PullMonitorable interfészt, ami tartalmazza az is_alive() metódust; ezt hívja a Fault Detector
- ORB-hez csatolt *mechanizmusok* az ORB és az OS között: Logging, Recovery és Message Handling Mechanism

- Logging Mechanism: minden üzenetet automatikusan tárolni tud;
  periodikusan hívja a get_state() metódusát az alkalmazás objektumnak (ez a Checkpointable interfész része, az objektumnak implementálnia kell), így az állapotot is menteni tudja
- Recovery Mechanism: üzeneteket vissza tudja játszani
  helyreállításkor a set_state() metódus hívásával tölti vissza az elmentett állapotot (ez az Updateable interfész része)
- Message Handling Mechanism: többszörös kérések és válaszok kiszűrése, csoportkommunikáció

7.3 A hibatűrő infrastruktúra (Fault Tolerance Infrastructure)

Többszörözhető infrastruktura objektumok minden FTD-ben: Replication Manager, Fault Notifier, Fault Detector



A hibatűrő CORBA architektúra

7.3.1 Replication Manager: Objektumcsoport menedzselése

PropertyManager interface: az objektumcsoport hibatűrő tulajdonságainak beállítása

- MembershipStyle: objektumcsoport tagság kezelése
  - infrastruktúra vezérelt: a Replication Manager hozza létre az objektumcsoportot, fenntartja a minimális számú replikát
  - alkalmazásvezérelt: az alkalmazás hozza létre a replikákat, kéri a Redundancy Managert, hogy adja hozzá ezeket az objektumcsoporthoz; a minimális számú replika fenntartását az alkalmazás végzi

- ConsistencyStyle: replika objektumok állapotának kezelése
    - infrastruktúra által vezérelt: szigorú replika konzisztencia (az objektumcsoport viselkedése megfelel egy egyszerű objektum viselkedésének, hiba esetén is: aktív replikáció esetén minden hívás után konzisztens állapotba kel kerüljenek a replika objektumok; passzív replikáció esetén pedig állapot-transzfer után) – megköveteli a Checkpointable interfész implementációját az alkalmazás objektumban (állapotmentés, üzenettárolás és helyreállítás)
    - alkalmazásvezérelt: nem garantált a szigorú replika konzisztencia, alkalmazásfüggő mentés és helyreállítás valósulhat meg
- ReplicationStyle: többszörözés típusa
    - STATELESS: a szerver objektum működését a hívások sorrendje nem befolyásolja;
    - COLD_PASSIVE: csak az aktív (elsődleges) objektum hajtja végre a kéréseket; ennek állapotát és az üzeneteket periodikusan menteni kell; hiba esetén egy tartalék veszi át az aktív szerepét; ennek állapota a mentés és az üzenet naplózása alapján állítható helyre;
    - WARM_PASSIVE: csak az aktív (elsődleges) objektum hajtja végre a kéréseket; ennek állapotát és az üzeneteket periodikusan menteni kell; a tartalék objektum(ok) rendszeresen átveszi(k) az aktív objektum állapotát;
    - ACTIVE: valamennyi objektum (azonos sorrendben) végrehajtja a kéréseket; a többszörözött válaszokat és kéréseket a Message Handling Mechanism szűri ki; ha egy objektum meghibásodik, a tartalékok még működnek;
    - ACTIVE_WITH_VOTING (kiterjesztés): az aktívan többszörözött objektumok szavaznak a kérésekről/válaszokról; csak a többség egyetértése esetén van kérés/válasz (adathibák esetén is alkalmazható a hibatűrés)
- FaultMonitoringStyle: hibamonitorozás (detektálás) módja
    - PULL: lekérdezés, a Fault Detektor hívja az objektumot, hogy ellenőrizze az elérhetőséget (az objektum implementálja a PullMonitorable interfészt)
    - PUSH (kiterjesztés): jelentkezés, az alkalmazásobjektum periodikusan jelenti a Fault Detektornak, hogy elérhető (alkalmazásfüggő)
- FaultMonitoringGranularity:
    - MEMB: az objektumcsoport minden tagját monitorozni kell
    - LOC: hostonként csak egy tagját kell monitorozni az objektumcsoportnak
    - LOC_AND_TYPE: egy típus esetén hostonként csak egy objektumot kell monitorozni
- Factories: Factory objektumok adatai
- InitialNumberReplicas: a replika objektumok kezdeti száma (létrehozáskor)
- MinimumNumberReplicas: a replika objektumok minimális száma (működés közben)
- FaultMonitoringInterval: objektum monitorozás periódusideje
- CheckpointInterval: objektum állapotmentés periódusideje

GenericFactory interface: Objektumcsoport létrehozása

- ezen keresztül az alkalmazás create_object() üzenettel kéri egy objektumcsoport létrehozását (mintha csak egy objektum létrehozását kérné, a különbség annyi, hogy a Factory helyett a ReplicationManager-hez megy a kérés);
- ennek hatására a ReplicationManager create_object() üzenetet küld a különböző hostok Factory objektumainak, amelyek létrehozzák a replika objektumokat.

ObjectGroupManager interface: Objektumcsoport tagság

- alkalmazásvezérelt MembershipStyle esetben az alkalmazás menedzselheti az objektumcsoportot a create_member(), add_member(), remove_member() üzenetekkel

7.3.2 Fault Detector: globális hibadetektálás

- A lokális Fault Detector objektumokat monitorozza (így egy host kiesése is detektálható)

7.3.3 Fault Notifier: hibajelzések

- Ennek jelentik a hibákat a lokális és globális Fault Detector objektumok, valamint egyéb hibadetektáló objektumok
- A hibát a Replication Manager felé terjeszti, valamint minden objektum felé is, amely a hibajelzésre regisztrálta magát
- Hibaanalízis (és -előrejelzés) lehetséges alkalmazás-specifikus regisztrált objektumokkal (Fault Analyzer objektumokkal)

7.4 Mechanizmusok összefoglalása

- Objektumcsoport létrehozás: alkalmazás hívja a ReplicationManager GenericFactory interfész create_object() metódusát,
  ez a hostok lokális Factory objektumának create_object() metódusát hívja a beállított paraméterek (ld. PropertyManager interface) szerint
- Objektumcsoport hívása: a Redundancy Manager által visszaadott csoportreferencia alapján hívható az objektumcsoport, mintha egy egyszerű objektum volna
- Hibadetektálás: lokális Fault Detector monitorozza az objektumokat azok is_alive() metódusának hívásával, ezeket a globális Fault Detectorok monitorozzák (szintén is_alive() metódushívással)
- Hibajelzés: a Fault Detector objektumok a Fault Notifier felé jelzik a hibát; a Fault Notifier-hez speciális hibaanalízis objektum is csatlakozhat.
- Hibakezelés: Redundancy Manager által,
  - a ReplicationStyle függvényében újrakonfigurálás,
  - a ConsistencyStyle függvényében helyreállítás.
- Újrakonfigurálás:
  - Aktív replikáció esetén csak a hibás objektum helyreállítása szükséges;
  - Passzív replikáció esetén csak egy aktív objektum lehet;
    ha a Fault Detector ennek hibáját jelzi, akkor a Redundancy Manager ezt újraindítja, ha ez sikertelen, akkor egy másik objektumot jelöl ki aktívnak a tartalékok közül.

- Helyreállítás:
    - ha alkalmazásvezérelt a ConsistencyStyle, akkor az új aktív objektum feladata a saját állapot helyreállítása
    - ha infrastruktúravezérelt a ConsistencyStyle, akkor a helyreállítás automatikus: az alkalmazás objektumokban felhasználható a Logging Mechanism (üzenetek tárolása, állapotlekérdezés és -tárolás a Checkpointable interfész get_state() metódusa segítségével), és Recovery Mechanism (üzenetek újrajátszása, állapot betöltése a Checkpointable interfész set_state() metódusa segítségével).