Elosztott rendszerek alapszolgáltatásai

Előadásvázlat „Szolgáltatásbiztonságra tervezés” tárgyból

Majzik István

BME Méréstechnika és Információs Rendszerek Tanszék

Tartalomjegyzék:

1 Bevezető

Ebben a fejezetben az elosztott rendszerek szolgáltatásbiztonságához szükséges alapszolgáltatásokat tekintjük át.

Milyen jellegű elosztott rendszerekkel foglalkozunk? Lazán csatolt rendszerekkel:

- Több résztvevő (szerver, processzor), lokális memóriával (nincs megosztott memória).
- Kommunikációs hálózat, üzenetekkel történő kommunikáció a résztvevők között.

Szoftver struktúra:

- Alkalmazói réteg: Elosztott feldolgozás (együttműködő processzek).
- Köztesréteg az elosztott működéshez és szolgáltatásbiztonsághoz, alapszolgáltatásokkal, pl.:
    - Tagsági protokollok: mely résztvevők elérhetők a csoportban és melyek estek ki
    - Globális időkezelés: események sorrendezéséhez lehet szükséges
    - Csoportkommunikáció: résztvevők egy csoportja ugyanazokat az üzeneteket kapja meg, garantált sorrendezéssel
    - Távoli eljáráshívás: eljáráshívás hívó és távoli hívott fél között
    - Atomi egyetértés: összetett műveletek, akciók egységes eredményű végrehajtása
    - Konzisztens helyreállítás: állapotmentés és hiba utáni helyreállítás
- Rendszerszoftver: Szokásos operációs rendszer (OS) szolgáltatások

Programozási paradigmák a hibatűrő elosztott rendszer alkalmazói rétegében:

- Aktív replikáció: Egy szolgáltatást több, párhuzamosan működő résztvevő valósít meg
    - tagsági kép, csoportkommunikáció mindenképpen szükséges
- Passzív replikáció: Elsődleges és tartalék résztvevők
    - tagsági kép, állapotmentés szükséges
- Elosztott helyreállítás: Résztvevők lokális állapotmentésén (és opcionálisan üzenettárolásán) alapuló rendszerszintű helyreállítás
    - tagsági kép, atomi egyetértés, csoportkommunikáció, konzisztens helyreállítás szükséges

Hibamodell: Az egyes résztvevőkre érvényes feltételezés a hibák jellegéről:

- Leállás (fail-stop): hibás egység jól detektálhatóan beszünteti a működését (megáll)
  fail-silent: nincs hibás kimenő üzenet vagy szolgáltatás
- Összeomlás (crash): hibás egység összeomlik, ezért működést beszünteti
  nehezebben detektálható (pl. csatorna nyelőként működik)
- Kihagyás (omission): hibás egység szolgáltatást/választ kihagy
  pl. egy szerver adott kérésre nem válaszol, csatornában eltűnnek bizonyos üzenetek
- Időzítés (timing): nem megfelelő időben nyújtott szolgáltatás/válasz (túl lassú vagy túl gyors)
- Adathiba (corruption): hibás adatérték jelenik meg
- Bizánci (Byzantine): tetszőleges, félrevezető hiba
  pl. egy multicast üzenetküldés esetén nem mindenkihez azonos üzenet jut el

Szinkronitás: Itt a ”szinkron” szó jelentése: a végrehajtási idő véges és ismert határon belül van:

- üzenetküldés (kommunikációs hálózat): az üzenetet korlátos időn belül célhoz ér;
- processz végrehajtás (processzor): korlátos időn belül elvégzi a számítást.

2 Tagsági kép nyilvántartása

Cél: Minden résztvevőnek egységes képe legyen a rendszerben elérhető hibamentes résztvevőkről (a továbbiakban „résztvevő” helyett sokszor „processz” fog szerepelni).

- Elérhetőséget befolyásolja: kiesés (meghibásodás, leválás) illetve csatlakozás (helyreállítás).

Kulcselemek a *változás észlelése* és a *konszenzus* (minden résztvevő egységesen lássa a tagsági képet).

- Változás észlelése: kiesés (meghibásodás, leválás) esetén *hibadetektálás* szükséges, míg a csatlakozás esetén a csatlakozó (helyreállított) résztvevő explicit jelezni tud (bejelentkezés).

2.1 Hibadetektálás

Hibadetektor tervezése:

- legyen megbízhatóbb a megfigyelt rendszernél,
- hibás detektálás következménye ne legyen károsabb, mint a detektálás hiánya.

Hibadetektor típusok:

- Lokális: a megfigyelési csatorna tökéletes (pl. helyi hihetőségvizsgálat, watchdog, MMU alapú ellenőrzés) - a hatékony lokális hibadetektálás a fail silent működés biztosítéka
- Elosztott: másik résztvevő végzi a hibadetektálást; a megfigyelt résztvevő és a megfigyeléshez használt csatorna hibája együtt detektálható.
  Összeomlás (crash) hibák detektálása esetén:
  - Push stílus: ”I am alive” jelek periodikus küldése a hibadetektorhoz; kimaradás észlelhető (aktív időszakban a normál üzenetek adják a ”szívdobbanást”)
  - Pull stílus: Hibadetektor lekérdez, a válasz hiánya vagy hibája észlelhető

Konzisztens hibadetektálás: Egy hibát minden processz ugyanúgy könyveljen

- Szigorú elvárások:
  - Szigorú pontosság: *Minden* hibamentes processzre igaz, hogy nem detektálja hibásnak a többi hibamentes.
  - Szigorú teljesség: *Minden* hibás processzre igaz, hogy végül *minden* hibátlan detektálja.
- Hibadetektorok megvalósítása:
  - *Tökéletes hibadetektor*: A fentieket (szigorú pontosság és szigorú teljesség) teljesíti. Ha a csatornák nem hibáznak, akkor a fenti push és pull megvalósítás is tökéletes hibadetektálást biztosít összeomlás (crash) hibák esetén. (A crash hibák detektálása nélkül megbízható üzenetküldés sem lehetséges: ha a crash hiba nem detektált, akkor végtelen sokszor lehet szükség újraküldésre.)
  - Hibázó csatorna tökéletessé konvertálható, ha a kihagyások (omission) száma korlátos: redundáns (többszörös) üzenet átvitel kell.
  - Nem készíthető tökéletes hibadetektor:

        - hibák száma és típusa a csatornán nem korlátos,
        - aszinkron a csatorna: ez esetben bizonyítható, hogy nem lehetséges tökéletes hibadetektor (Fischer-Lynch-Patterson lehetetlenségi tétel)

- Gyengített elvárások:
    - Gyenge pontosság: *Legalább egy* hibamentes processzre igaz, hogy nem detektálja hibásnak a többi hibamentes.
      Hisznek neki, így pl. koordinátor szerepű lehet a tagsági kép kialakításában.
    - Gyenge teljesség: Minden hibás processzre igaz, hogy végül *legalább egy* hibátlan detektálja.
- Aszinkron rendszerekben jellemző elvárások:
    - Végső soron (eventually) gyenge pontosság: *Bizonyos idő után legalább egy* hibamentes processzre igaz, hogy nem detektálja hibásnak a többi hibamentes.
      Az így elért stabilitási periódusban a hibátlannak elfogadott processz lehet koordinátor szerepű.
    - Gyenge teljesség itt is kritérium.

2.2 Konszenzus

Konszenzus probléma: A bekövetkező hibák ellenére minden processz egyetért egy közös értékben (itt: a tagsági képben).

Konszenzus protokollok tulajdonságai:

- *Egységes* (uniform) konszenzus: Ha két résztvevő döntésre jut, akkor azonos döntésre jutnak; ha közben az egyik crash hibás lesz, a nála lévő döntés is azonos a hibamentesek által hozott döntéssel.
  Következmény: Helyreállítás után a hibás is használhatja a döntést; a konszenzus kialakítása során tekintettel kell lenni azokra a résztvevőkre, amik a kiesésük előtt kap(hat)tak döntést.
- *Nem egységes* konszenzus: Ha két *hibamentes* résztvevő döntésre jut, akkor azonos döntésre jutnak.
  Következmény: A konszenzus protokoll közben kieső résztvevő által a kiesés előtt kapott döntést nem kell figyelembe venni a protokoll lezárásakor. A kieső résztvevőnek helyreállítás után le kell kérdeznie a döntést.

Egy tipikus konszenzus protokoll:

- Hibamentes esetben: Koordinátor választható, ez karbantartja és szétküldi a tagsági képet.
- Koordinátor crash hiba esetén, tökéletes hibadetektorral, szinkron rendszerben: Előfordulhat, hogy csak néhány processz kapja meg a tagsági képet, majd a koordinátor kiesik. Az új koordinátor tagsági képe különbözhet az előzőétől $\rightarrow$ nem lesz egységes konszenzus a rendszerben.
    - Megoldás: Egy érték (itt tagsági kép) csak akkor használható, ha garancia van arra, hogy a többiek is erre döntenek (hiba esetén is). Háromfázisú algoritmussal megoldható:
        - A koordinátor az első fázisban egy *javasolt* értéket küld a résztvevőknek, ezt a résztvevők maguknak beállítják.
        - A résztvevők *nyugtát* küldenek a kapott javasolt értékről a koordinátornak.

- Ha a koordinátor minden nyugtát megkap, akkor jóváhagyást küld a résztvevőknek (a *döntés* ekkor történik meg, a résztvevők a jóváhagyott értéket használják).

Ha az első két fázisban esik ki a koordinátor: Csak javasolt értékek voltak, döntés még nem történt, az új koordinátor új értéket javasolhat.
Ha a harmadik fázisban esik ki a koordinátor: Az új koordinátor az általa már megkapott javasolt értéket fogja használni, így ugyanazt, amit az előző koordinátor is használhatott.

- Ha nem szükséges az egységes konszenzus: Elég, ha az új koordinátor csak lekérdezi az előző döntést. Ha már volt ilyen, akkor ezzel fejezi be a protokollt, ha még nem volt ilyen, akkor ez esetben szabadon javasolhat új értéket (az esetlegesen döntést kapott, de kiesett résztvevőket nem kell figyelembe vennie).

- Aszinkron rendszerben: *Végső soron gyenge pontosságú* hibadetektor szükséges; a résztvevők többségének nyugtája alapján dönthet a koordinátor. Az új koordinátornak *le kell kérdeznie* a többieket (mivel az ő nyugtája nem biztos, hogy részt vett a javasolt értéket jóváhagyó fázisban).

2.3 Jellegzetes problémák

Partíciók problémája:

- Csatornák crash hibája partíciókra bonthatja a rendszert:
  - Több partíció aktív maradhat (párhuzamosan működnek): Csatorna helyreállítása után a partíciók állapotainak szinkronizálása a legtöbb gyakorlati alkalmazásban nem megvalósítható (melyik partíció állapota legyen rákényszerítve a többire?).
  - Csak egy (elsődleges) partíció maradhat aktív: Tipikusan az a partíció marad aktív, ahol a résztvevők többsége (vagy ezek súlyainak többsége) van.
- A csatorna crash hibája time-out alapján detektálható (ezért aszinkron rendszerben nincs partíciódetektálás). Ez ciklikusan partíciókra szakadáshoz és újracsatlakozáshoz vezethet.
  Internet routing protokoll példa: túlterhelés → csatorna time-out → csatorna hibásnak detektálva → partíciókra bomlás → egy aktív partíció marad → csatorna túlterhelése megszűnik → kapcsolat helyreáll → partíciók állapotainak szinkronizálása szükséges → ezáétal újra csatorna túlterhelés következik be és kezdődik elölről …
  Itt a csatorna time-out olyan megválasztása szükséges, hogy a szinkronizálás ne okozzon újabb partíciókra szakadást.

Tagsági kép változás sorrendezése: Fontos a tagsági kép változásáról szóló üzenetek és az alkalmazói üzenetek azonos sorrendben történő feldolgozása minden résztvevő esetén.

Példa: Terheléselosztás: A bejövő kérések tagsági kép alapján történő elosztása; minden résztvevő feldolgozza a rá eső részt.

- Egy szerver kiesik: Detektálás, majd tagsági kép frissítés külön üzenettel a koordinátortól.
- Tagsági kép frissítés üzenet és bejövő kérés átlapolódik: Ha nem ugyanaz a feldolgozási sorrend minden résztvevő esetén, akkor más-más információk alapján osztják el a kéréseket az egyes résztvevők, így egyes kérések feldolgozása kimaradhat.

3 Atomi döntéshozatali (commit) protokollok

A konszenzus egy speciális esete, szabályok (ld. lentebb) által megkötött azonos döntés (vagy a döntés hiánya) szükséges minden résztvevő esetén. Használatára példa: Egy összetett művelet esetén a részműveletek eredményéről (sikeres/sikertelen végrehajtásáról) kell azonos döntésre jutni a résztvevőknek.

- Általában a megoldás a következő:
    - A döntéshozatalt egy *koordinátor* indítja (pl. a művelet kezdeményezője).
    - A résztvevők szavaznak (Yes/No jellegű szavazat a részművelet eredményéről).
    - Egyetértés (agreement) eredményeként Commit/Abort lezárása a műveletnek.
- Az atomi *döntéshozó protokoll* (atomic commitment protocol) egy speciális egységes konszenzus protokoll:
    - minden résztvevő ugyanarra a döntésre jut,
    - résztvevők szavazataikat utólag nem változtathatják meg,
    - a döntés eredménye csak akkor Commit, ha minden résztvevő szavazata Yes,
    - ha minden résztvevő szavazat Yes, akkor a döntés eredménye Commit,
    - hibás résztvevők javítás/helyreállítás után megkaphatják a döntést.

Jellegzetességek:

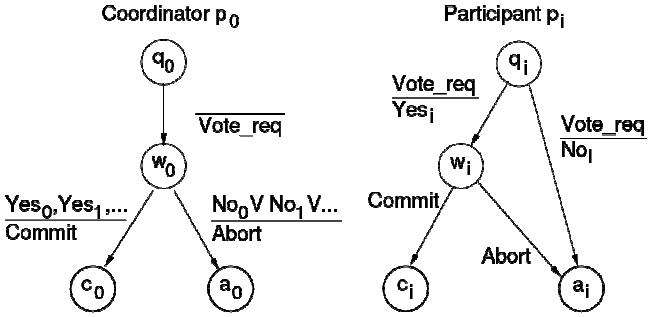
- Hibamodell: Résztvevő crash hibája illetve kommunikáció omission hibája.
- Résztvevő bizonytalan állapota: Szavazatként Yes-t küldött, de döntést még nem kapott.
- Blokkolás: Ez egy speciális rendszerállapot; meghibásodott résztvevők vannak; a hibamentesek bizonytalanok és nem tudnak döntést hozni addig, amíg a hibás résztvevők újra nem indulnak, mert fennáll az inkonzisztens döntéshozatal veszélye.
  Költséges: A művelet lezárása késlekedik, az erőforrások nem szabadíthatók fel.

A konzisztens döntéshozatal itt tárgyalt protokolljai:

- Kétfázisú döntéshozó protokoll (two-phase commit protocol, 2PC)
- Háromfázisú döntéshozó protokoll (three-phase commit protocol, 3PC)

3.1 Kétfázisú döntéshozó protokoll (2PC)

3.1.1 Alapséma



A kétfázisú döntéshozó protokoll

3.1.2 Az időtúllépés (time-out) kezelése

A 2PC során bekövetkező hibát az üzenet time-out detektálja. A time-out kezelése:

- Résztvevő vár Vote_req-ra: Abort a döntés
- Koordinátor vár Yes/No-ra: Abort a döntés
- Egy résztvevő vár Commit/Abort-ra: Bizonytalan állapotban van!
  Többi résztvevőt kell lekérdeznie az eredményről (feltehető, hogy a résztvevők egymást ismerik, pl. a Vote_req-val küldött partnerlista alapján).
  A lekérdezett elküldi a döntést (ha már megkapta) illetve Abort-ot, ha No szavazata volt.

3.1.3 Blokkolás a 2PC során

Blokkolás: Minden elérhető résztvevő bizonytalan állapotban van:

- a koordinátor nem elérhető,
- a résztvevők, amelyek már kaphattak döntést, szintén nem elérhetőek,
- az élő résztevők nem dönthetnek (még Abort-ra sem), mert lehetséges, hogy a nem elérhető résztvevők közül valamelyik már kapott (akár Commit) döntést a koordinátortól, és újraindítás után így fog viselkedni.

3.1.4 Helyreállítás a 2PC során

- Résztvevők állapota elmentve: naplózva a partnerlista és az állapotra utaló bejegyzések:
  - koordinátor: Start_2PC, Commit, Abort
  - egyszerű résztvevő: Commit, Abort, Yes
- Koordinátor: Start_2PC van a naplójában
  - Commit/Abort a naplóban: Megvan a döntés
  - Nincs Commit/Abort a naplóban: Abort-ra dönthet
- Résztvevő: Nincs Start_2PC a naplójában
  - Commit/Abort a naplóban: Megvan a döntés
  - Yes a naplóban: Bizonytalan állapot, a terminálási protokollra van szükség.
  - Nincs szavazat a naplóban: Abort-ra dönthet.

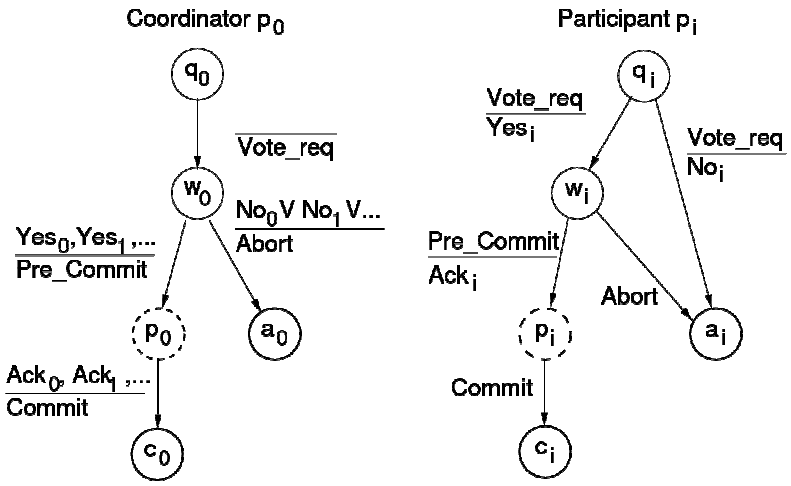
3.1.5 A 2PC értékelése

- Blokkolás: lehetséges akkor is, ha nem hibásodik meg minden résztvevő (azaz nincs totális hiba).
- Idő komplexitás: 3 (+ 2, ha lekérdezés és válasz szükséges)
- Üzenet komplexitás: $3n$ (n résztvevő esetén)

3.2 Háromfázisú döntéshozó protokoll (3PC)

3.2.1 Új elemek a 3PC-ben a blokkolás elkerülésére

- A 2PC-ben a blokkolás oka: Egy időben lehetnek bizonytalan, illetve döntést kapott, de nem elérhető résztvevők.
- A 3PC-ben: Közbenső állapotok és új üzenetek vannak bevezetve annak érdekében, hogy döntést csak akkor kaphasson egy résztvevő, ha már nincs bizonytalan. (A döntést csak akkor küldi el a koordinátor, ha már nincs bizonytalan résztvevő.)
    - *Pre_Commit* üzenet jelentése: Mindenki szavazata Yes volt; a döntés Commit lesz, ha senki sem hibásodik meg.
    - *Ack* üzenet jelentése: A résztvevő lezárta a bizonytalan állapotot.



A háromfázisú döntéshozó protokoll

3.2.2 Time-out kezelése a 3PC során

- Koordinátor Yes/No-ra vár: Abort-ra dönthet
- Koordinátor Ack-ra vár: Commit-ra dönthet
    - minden résztvevő Yes-re szavazott
    - néhány résztvevő meghibásodott, újraindulás után megkapják a döntés eredményét
- Résztvevő Vote_req-ra vár: Abort-ra dönthet
- Résztvevő Pre_Commit/Abort-ra vár: *Terminálási protokoll* indítása szükséges.
  A koordinátor meghibásodott, résztvevőknek kell dönteniük.
- Résztvevő Commit-ra vár: *Terminálási protokoll* indítása szükséges.
  Miért nem Commit-ra dönt? Ugyan már kapott Pre_Commit-ot, de még lehet olyan résztvevő, amely nem kapott (a koordinátor hibája miatt), így bizonytalan állapotban van.

3.2.3 Terminálási protokoll 3PC-ben

1. Új koordinátor választása (*vezetőválasztási protokoll*)

- Tipikus megvalósítás: Résztvevőknek prioritása van; a legnagyobb prioritású lesz a jelölt. Aki detektálja az aktuális koordinátor kiesését, az You_are_elected üzenetet küld a legnagyobb prioritású elérhető résztvevőnek. A protokollt az új koordinátor fejezi be.

2. Az új koordinátor lekérdezi a résztvevők állapotát State_request üzenettel.
   Az állapotok lehetnek:
   - Aborted: Abort-ot kapott, No-val szavazott vagy nem szavazott
   - Uncertain: Yes-szel szavazott, nem kapott Pre_Commit-ot vagy Abort-ot
   - Committable: Pre_Commit-ot kapott, de Commit-ot még nem
   - Committed: megkapta a Commit-ot
3. Döntés az így kapott állapotinformáció alapján, sorban vizsgálva a következőket:
   - van Aborted: Abort,
   - van Committed: Commit,
   - mind Uncertain: Abort (3PC-ben ez lehetséges, mivel nem lehet olyan résztvevő, amely már döntést kapott),
   - van Committable (a többi Uncertain): Pre_Commit a bizonytalanoknak, bevárni az Ack-ot; küldeni a Commit-ot.
4. Time-out a terminálási protokoll során:
   - A vezetőválasztás során: nem elérhetőek nem lesznek figyelembe véve
   - Új koordinátor meghibásodása: új választás kezdeményezése

3.2.4 Helyreállítás a 3PC során

Hasonlóan, mint 2PC esetén, naplózni kell a partnerlistát és az állapotra utaló bejegyzést (Start_3PC, Yes, Commit, Abort bejegyzésekkel).

- Résztvevő (naplójában Yes, Commit, vagy Abort lehet), sorban vizsgálható:
  - Commit/Abort a naplóban: Döntés már megvan
  - Nincs Yes szavazat a naplóban: Abort-ra dönthet
  - Yes a naplóban, de nincs Commit/Abort: Többi résztvevő kérdezése szükséges (nem terminálási protokoll; a többiek már dönthettek!)
  - Ha Pre_Commit lenne a naplóban: Többi résztvevő kérdezése kellene itt is!
    Ld.: ha a koordinátor meghibásodott, a többiek Abort-ra dönthettek (terminálási protokoll során);
    Pre_Commit tehát nincs is a log-ban tárolva, felesleges lenne.
- Koordinátor: hasonló módon jár el (az új koordinátor választása és döntése miatt neki is kérdezgetnie kell).

3.2.5 Kommunikációs hibák és a 3PC

- Kommunikációs hibák: Inkonzisztens döntések lehetségesek a terminálási protokoll során.
  Partíciókra szakadás lehetséges, ezt a terminálási protokoll nem kezeli.
  Pl: két független partíció kialakulása:

1: minden résztvevő Uncertain, így Abort-ra döntenek
2: minden résztvevő Committable, így Commit-ra döntenek

- Javított terminálási protokoll: Az új koordinátor csak akkor küldhet döntést, ha azt a résztvevők többsége megkaphatja (pl. a terminálási protokoll során a lekérdezéssel és az Ack fogadásával erről meggyőződhet).
- Javított koordinátorválasztási protokoll: A jelölt is ellenőrzi, hogy van-e nála nagyobb prioritású.

3.2.6 A 3PC értékelése

- Hibatűrés: Csak a résztvevők crash hibájára van felkészülve.
- Blokkolás: csak akkor, ha minden résztvevő meghibásodik (total site failure)
  - Ha minden élő résztvevő bizonytalan, Abort-ra dönthetnek (nincs résztvevő, amely megkaphatta a döntést).
  - Minden résztvevő meghibásodik: Az újraindulóknak várakozni kell egy olyan (szintén újrainduló) résztvevőre, amely megkapta a döntést, vagy meghatározza a döntést (pl. No-val szavazott). Ld. helyreállítás: ha bizonytalan, kérdeznie kell!
- Idő komplexitás: 5 (+ 6 a terminálási protokoll, ha szükséges)
- Üzenet komplexitás: *5n*

4 Események és üzenetek sorrendezése

Elosztott rendszerekben a kommunikációs késleltetések miatt a különböző résztvevők különböző sorrendben értesülhetnek az egyes eseményekről. Ez nem minden esemény esetén megengedhető. Általában cél az, hogy események (elosztott) feldolgozásának sorrendje feleljen meg az oksági viszonyoknak (hamarabb értesüljenek a résztvevők az okról, mint az okozatról). Az oksági viszonyok megállapítása lehet utólagos cél (pl. egy napló elemzése során), vagy futás közben biztosítandó feltétel (pl. beérkező üzenetek várakoztatása, ha az azonnali feldolgozás nem felelne meg az oksági viszonyoknak).

4.1 Események sorrendezése a valós idő alapján

- Motiváció: Ha egy esemény oka egy másik eseménynek, akkor fizikai idő szerint előbb kellett történnie; ez a valós (fizikai) órák által nyújtott időbélyegek alapján bizonyos korlátok között megállapítható.
- A valós időbélyegek használatához a lokális órákat szinkronizálni kell: Ez történhet külső forráshoz képest, vagy egymáshoz képest (belső).
- A szinkronizáció tökéletesen nem oldható meg a hálózati késleltetések változása miatt. Fizikai órák jellemzői: Monotonitás: $C_i(t+\tau) \geq C_i(t)$ ha $\tau \geq 0$; szinkronizációs pontosság: $|C_i(t) - C_j(t)| < \beta$; intervallum pontosság: $(1-\rho)\tau \leq C_i(t+\tau) - C_i(t) \leq (1+\rho)\tau$, ahol $C_i(t)$ jelenti a p_i processz lokális óráját a t időpillanatban.
- Jellegzetes megoldások a szinkronizálás során a hibatűrő átlagolás (legkisebb n és legnagyobb m értéket elhagyva), illetve a bizánci átlagolás. Hálózati késleltetés ingadozása döntően befolyásolja a szinkronizációs pontosságot.

4.2 Események logikai sorrendezése

Az ismert „előbb történt” (happened before) reláció egy lehetséges oksági viszonyt határoz meg (azaz a valós oksági viszonyok egy megjelenítője, ld. lentebb).

Az „előbb történt” reláció jelölése két esemény között $a \rightarrow b$ (az a esemény előbb történt, mint b). Definíciója a következő:

- Egy processzben az egymás utáni a, b műveletekre $a \rightarrow b$;
- Processzek között az üzenet küldése $send(m)$ és fogadása $receive(m)$ között $send(m) \rightarrow receive(m)$ definiálható;
- Tranzitív reláció.

Ez a reláció az oksági kapcsolat egy megjelenítője; ha *a* oksági kapcsolatban van *b*-vel (az *a* oka *b*-nek) akkor $a \rightarrow b$. Visszafelé már nem igaz! Konkurens eseményekről van szó, ha $a \not\rightarrow b$ és $b \not\rightarrow a$

Természetes számokkal történő sorrendezés (logikai óra) megvalósítás: Minden processz lokális számlálót tart fenn (lokális *C(a)* „időbélyeg” hozzárendelése egy *a* eseményhez).

- Egy processzben egymás után következő lokális $a \rightarrow b$ események esetén (az üzenetküldés is ide tartozik) a számlálót (időbélyeget) egyesével növelik: $C(b) = C(a) + 1$
- Üzenetküldéskor az aktuális időbélyeget csatolják a küldött üzenethez.
- Ha $C_i(send(m))$ egy m üzenet küldésének időbélyege a p_i processzben, az időbélyeg csatolva van az üzenethez, és a p_j processz a $C_j(e)$ időbélyegű lokális esemény után fogadja ezt az üzenetet, akkor a fogadáshoz tartozó időbélyeg:

$$C_j(receive(m)) = \max(C_i(send(m)), C_j(e)) + 1$$

- Tulajdonság: Ha $a \rightarrow b$ akkor $C(a) < C(b)$ fennáll, de fordítva nem tudunk biztosat mondani: Ha $C(a) < C(b)$, akkor még nem biztos, hogy $a \rightarrow b$ (az időbélyegek konkurens események esetén is rendezhetők).

Idővektor alapú időbélyeg megvalósítás: Minden processz egy n elemű vektort tart fenn időbélyegként, ha n processz van a rendszerben.

- Egy p_i processz esetén a vektor i-edik eleme lokális számláló, $j \neq i$ eleme a p_i processztől érkezett üzenet sorszáma; ennek frissítése hasonlóan zajlik, mint a számlálós algoritmus esetén.
- Vektorok összevetése: $V_1 < V_2$ akkor, ha $\forall i : V_1[i] \leq V_2[i]$ valamint $\exists j : V_1[j] < V_2[j]$
- Konkurens események: $V_a \not< V_b$ és $V_b \not< V_a$

4.3 Üzenetek sorrendezése

Az üzenetek feldolgozásának sorrendezése egy speciális feladat, különös jelentősége csoportkommunikáció esetén van, azaz amikor az üzeneteket a résztvevők egy csoportja (multicast üzenetek esetén) illetve a rendszer minden résztvevője (broadcast üzenetek esetén) megkapja. Itt az üzenetek sorrendezése előírásként jelenik meg (előírja az adott sorrendben történő feldolgozást).

4.3.1 Tipikus rendezések

Legyen m és n két üzenet; $s(m)$ az m üzenet küldése, $r(m)$ az m üzenet fogadása. A következő sorrendezési előírások tipikusak:

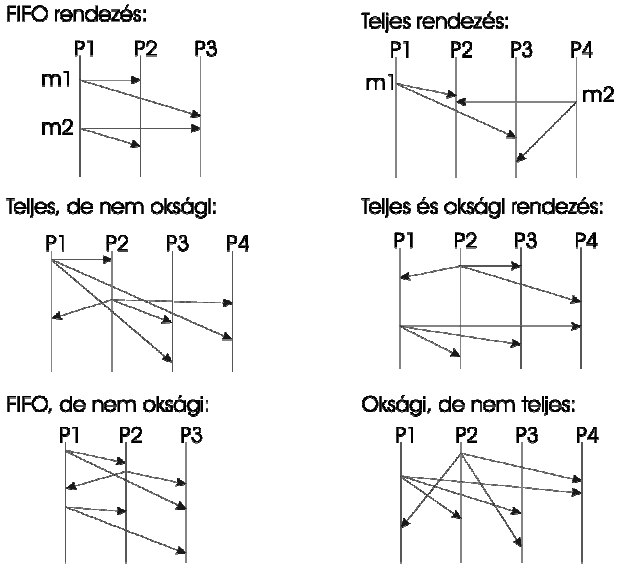
- **FIFO sorrendezés** (FIFO ordering): az azonos processz által küldött bármely két n és m üzenetre:
  ha $m \rightarrow n$ akkor az üzeneteket fogadó minden i processzre $r_i(m) \rightarrow r_i(n)$.

Alkalmazási példa: Szerverek és egymástól független kliensek kommunikációja.
Megvalósításra példa: Üzenetek sorszámozása a küldőnél.

- **Oksági sorrendezés** (causal ordering): bármely két n és m üzenetre (nem csak azonos processztől):
  ha $s(m) \rightarrow s(n)$ akkor az üzeneteket fogadó minden i processzre $r_i(m) \rightarrow r_i(n)$
  Alkalmazási példa: Egymással együttműködő (kooperáló) kliensek és ezeket kiszolgáló szerverek; munkafolyamatok, e-business alkalmazások.
  Megvalósítási példa: CBCAST algoritmus (ld. a következő alfejezetben).
  Az oksági sorrendezés egyben FIFO sorrendezés is, de fordítva ez nem igaz.

- **Teljes sorrendezés** (total ordering): bármely két n és m üzenetre (nem csak azonos processztől):
  az üzeneteket fogadó minden processz azonos sorrendben dolgozza fel ezeket.
  Ez az előírás nem garantálja az oksági sorrendet, csak az azonos feldolgozási sorrendet.
  Alkalmazási példa: Replikált (meleg tartalékolt) szerverek, amelyeknek egymással szinkronban kell működniük, hogy egymás helyébe léphessenek; ez esetben a konkurens üzeneteket is célszerű rendezni.
  Megvalósítási példa: Sorrendező (sequencer) processz alkalmazása (ld. a következő alfejezetben).

- **Teljes és oksági sorrendezés** (total order preserving causality):
  Az üzeneteket fogadó minden processz azonos sorrendben dolgozza fel az üzeneteket, ami egyben megfelel az oksági sorrendnek is (ha van).



Példák üzenetek sorrendezésére

4.3.2 Algoritmusok

4.3.2.1 Oksági sorrendezés broadcast üzenetek esetén: CBCAST

Előfeltételek:

- n számú processz: $p_1, p_2, ..., p_n$, sorrendezik az általuk kiküldött üzeneteket
- minden processz tárol egy vektort az eddig beérkezett üzenetek sorszámairól: $[s_1, s_2, ..., s_n]$, ahol s_i az i-edik processztől vett eddigi legnagyobb sorszám;
- az üzenet küldője a saját vektorát az üzenethez csatolja.

Algoritmus: p_i processz az $[s_1, s_2, ..., s_n]$ vektort tárolja; üzenetet kap a p_j processztől, az üzenethez a $[t_1, t_2, ..., t_n]$ vektor van csatolva.

- p_i elfogadja az üzenetet, ha $t_j = s_j + 1$ és $\forall k, k \neq j : t_k \leq s_k$ (azaz a következő üzenet érkezett a p_j processztől, és p_i tudása elég friss);
- p_i késlelteti az üzenet elfogadását, ha $t_j > s_j + 1$ (kimaradt üzenet a p_j processztől) vagy $t_j = s_j + 1$ de $\exists k, k \neq j : t_k > s_k$ (azaz p_i tudása nem elég friss, p_j kapott már egy üzenetet p_k-tól, amit p_i még nem kapott meg);
- p_i eldobja az üzenetet, ha $t_j < s_j + 1$ (azaz duplikált üzenetről van szó).

4.3.2.2 Teljes sorrendezés: Sorrendező processz

Ha egy processz multicast üzenetet akar küldeni, akkor a sorrendezőhöz fordul, és sorszámot kér; ezt az üzenethez csatolva küldi majd el. Ez határozza meg a feldolgozási sorrendet minden címzettnél.

Minden fogadó nyilvántartja a legnagyobb, eddig beérkezett sorszámot:

- ha ennél kisebb vagy egyenlő sorszámú üzenet érkezik: eldobja (duplikált),
- ha ennél eggyel nagyobb sorszámú üzenet érkezik: elfogadja és feldolgozza,
- egyébként késlelteti.

Ha a sorrendezőtől a sorszám vétele és a hozzá tartozó üzenet küldése nem oszthatatlan művelet (közben pl. fogadhat másik üzenetet a küldő) akkor nem lesz oksági a rendezés.

A sorrendező szűk keresztmetszet (egyszeres hibapont), de dinamikusan váltható.

4.3.2.3 A csoportkommunikációban előforduló bizánci hibák kezelése

Részletes elemzés és algoritmusok tárgyalása nélkül megjegyezzük, hogy legalább $3t + 1$ számú résztvevőből álló elosztott rendszer szükséges t számú bizánci hibás résztvevő kezeléséhez (tehát 1 bizánci hibás résztvevőt 3 hibátlan résztvevő tud detektálni és hibáját kezelni).

5 Elosztott helyreállítás

A célkitűzés: Több, együttműködő processz hiba utáni helyreállítása elosztott rendszerekben. Az egyszerű, egy-egy processz esetén alkalmazható technikák kiegészítésre szorulnak.

A megoldási lehetőségek (amelyeket a következő alfejezetekben tárgyalunk) a következők szerint tekinthetők át:

Visszalépő helyreállítás (rollback recovery):

- *Állapotmentés alapú visszaléptetés* (checkpoint based rollback recovery): A visszaléptetés alapja csak az elmentett állapot, az elosztott rendszer minden processzét visszaállítják, így a processzek közötti hibaterjedés is kezelhető.
  - *Független állapotmentés* (independent checkpointing): a processzek egymástól függetlenül mentik állapotukat, konzisztens rendszerszintű elmentett állapot léte nem garantált.
  - *Koordinált állapotmentés* (coordinated checkpointing): a processzek összehangolják az állapotmentést, így konzisztens rendszerszintű állapotmentés garantált.
  - *Kommunikációfüggő állapotmentés* (communication-induced checkpointing): az állapotmentés processzenként függetlenül, de a kommunikációs szekvenciához illeszkedve történik.
- *Üzenettárolás alapú visszaléptetés* (log-based rollback recovery): A visszaléptetéshez az elmentett állapot mellett a normál működés során eltárolt üzenetek (események) is

használhatók. Csak a meghibásodott processzt léptetik vissza, ezért alkalmazási feltétel a hibaterjedés korlátozása. A tárolt állapotra történő lokális visszaléptetéssel és a tárolt üzenetek megfelelő sorrendben történő visszajátszásával a hibás processz állapota helyreállítható.

- *Optimista üzenettárolás* (optimistic logging): feltételezzük, hogy az üzenetek tárolása sikeres, a stabil tárba mentés a háttérben történik, miközben tovább futhat az alkalmazás.
- *Pesszimista üzenettárolás* (pessimistic logging): az alkalmazás csak akkor futhat tovább, ha megbizonyosodott az üzenetek sikeres tárolásáról (ennek megfelelően lassabb az optimista technikánál, de több garanciát is ad).
- *Oksági üzenettárolás* (causal logging): az üzenetek tárolása elosztott módon történik, azoknál a processzeknél, amelyeket befolyásolna az üzenetek elvesztése (azaz sikertelen tárolása).

Előrelépő helyreállítás: Nem egy korábbi állapotra történő visszalépés történik, hanem a hiba (alkalmazásfüggő) kezelése korrekcióval vagy kompenzáló műveletekkel.

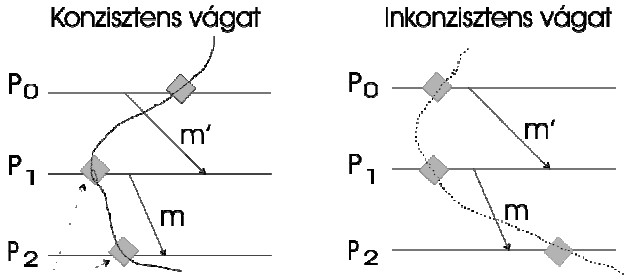
- *Koordinált atomi műveletek* (coordinated atomic actions): A konzisztens hibakezelés megvalósítása kompenzáló akciókkal.

5.1 Állapotmentés alapú visszaléptetés

Minden processz rendszeresen elmenti az állapotát, amelyet rendszerszintű visszaléptetésre lehet használni.

Fogalmak:

- *Vágat* (cut): minden processz esetén egy lokálisan elmentett állapotot ad meg; ezek alapján lehet megpróbálni a visszaléptetést.
- *Függő üzenet* (in-transit message, missing message): hátulról metszi a vágatot.
  Elmentett állapot jellemzője: A küldő már elküldte, fogadó még nem kapta meg (a ”csatornában van”). Ha a rendszer ilyen állapotra lép vissza, akkor itt egy „elveszett üzenet” keletkezik, de nem kommunikációs hiba, hanem a visszalépés miatt. Ennek megfelelően a függő üzenetek kezelése kétféleképpen valósulhat meg:
  - Ha a megbízható üzenetküldésért felelős protokoll réteg a helyreállításért felelős réteg fölött van, akkor a visszalépés miatt „elveszett üzenetet” ez kezelheti (ugyanúgy, mint a csatornában a kommunikációs hibák miatt esetlegesen elveszett üzeneteket).
  - Ha a megbízható üzenetküldésért felelős protokoll réteg a helyreállításért felelős réteg alatt van, akkor a függő üzenetek kezelésésért a helyreállítási réteg lesz a felelős. A függő üzenetek hozzá kell, hogy tartozzanak az elmentett rendszerállapothoz, azaz tárolni kell ezeket és a visszalépés után visszajátszani a fogadónak.
- *Inkonzisztens üzenet* (inconsistent message, orphan message): elölről metszi a vágatot.
  Elmentett állapot jellemzője: A küldő még nem küldte el, de a fogadó már megkapta az üzenetet. Ilyen állapot a normál működés során nem fordulhat elő, sérti az oksági relációt, ezért az sem megengedett, hogy hiba utáni visszalépés során ilyen állapotba kerüljön a rendszer.
  Ha egy vágatban inkonzisztens üzenet található, akkor azt inkonzisztens vágatnak nevezzük.
- *Visszalépési vonal* (recovery line): a legutolsó visszaállítható konzisztens állapot (konzisztens vágat).



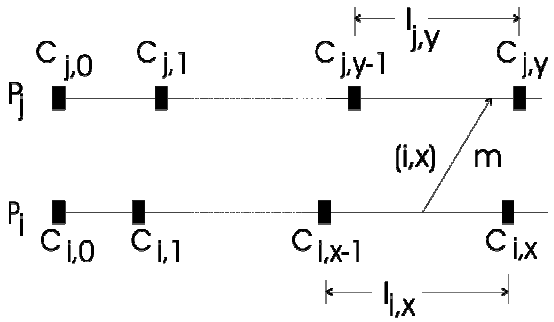
Konzisztens és inkonzisztens vágat

5.1.1 Független állapotmentés

Minden processz maga dönti el, mikor ment állapotot, a mentés lokálisan optimális időpontban történik (pl. egy feldolgozási ciklus végén), ezért kicsi lehet a futásidejű overhead.

A helyreállítás során a konzisztens globális állapot meghatározása kereséssel történhet a bejegyzett függőségek alapján:

- Függőségek nyilvántartása:
  Ha p_i az $I_{i,x}$ intervallumban küldi az (i,x) címkével ellátott m üzenetet a p_j processznek, amit az az $I_{j,y}$ intervallumban vesz, akkor a függőséget $I_{i,x}$ intervallum felől $I_{j,y}$ intervallum felé fel kell jegyezni, amikor a $c_{j,y}$ állapotmentés történik. (A függőség azt jelenti, hogy ha $I_{i,x}$ visszalépés történik, akkor $I_{j,y}$ visszalépés is szükséges, máskülönben inkonzisztens üzenet keletkezik.)

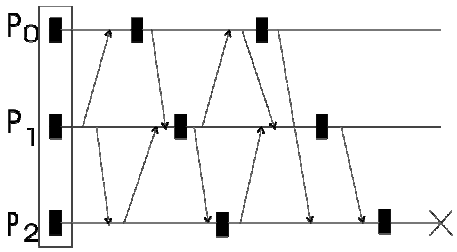


Függőségek a processzek között

- Helyreállítás során a visszalépési vonal (legutolsó konzisztens vágat) keresése a függőségi gráf felépítésével, majd ebben való kereséssel történik.
- Szemétgyűjtés: Az így megtalált visszalépési vonal előtti állapotmentések eldobhatók.

Problémák:

- Dominó-effektus: Előfordulhat, hogy az inkonzisztens üzenetek miatt nem található konzisztens vágat (”sorozatos visszaléptetés” szükséges, az elmentett állapotok használhatatlanok, a számítást újra kell kezdeni). Elkerülés: rendszerszinten összehangolt állapotmentés, amelynek során konzisztens vágat alakítható ki.



A dominó effektus

- Aszinkron visszaléptetés valósul meg, ha hibát detektáló processz visszalép és a többieket a függőségek mentén visszalépésre szólítja fel (majd ez a visszaléptetés a függőségek mentén továbbterjed, amíg függőségektől mentes konzisztens vágat nem alakul ki). Itt megjelenhet a *livelock* (leragadás): Ha kölcsönös a függőség két processz között, hiba esetén egymást rendre visszaléptetik. Elkerülés: Globálisan kell a visszalépési vonalat keresni (mint fentebb), nem pedig egymás aszinkron visszaléptetésével.

5.1.2 Koordinált állapotmentés

A koordinált állapotmentés során a processzek összehangolják lokális állapotmentésüket annak érdekében, hogy konzisztens vágat alakuljon ki.

- Előny: Nem áll fenn a dominó effektus veszélye (mindig van visszalépési vonal, ez az utoljára lezárt állapotmentésekből áll). Ennek megfelelően nem kell több állapotmentést tárolni, így a tárigény csökken.
- Hátrány: A koordináció idő- és üzenetigénye jelentős lehet (de általában elhanyagolható az állapotmentés ideje mellett).

A megvalósítások alapja a következő protokoll:

1. Az állapotmentést kezdeményező processz lokálisan állapotot ment és a többi résztvevőt is mentésre szólítja fel a *checkpoint_request* üzenettel.
2. Ha egy processz ezt megkapja, állapotot ment, és ennek sikeres megvalósítása esetén válaszol a kezdeményezőnek *local_checkpoint_done* üzenettel.
3. Ha a kezdeményezőhöz minden processztől megjött ez a válasz, akkor *global_checkpoint_done* üzenetet küld mindenkinek.
4. Ha egy processz ezt megkapja, akkor lezárja az állapotmentést; ha ezek után visszalépés szükséges, akkor ezt a legutóbbi lezárt állapotmentést használja. A korábbi állapotmentések eldobhatók.

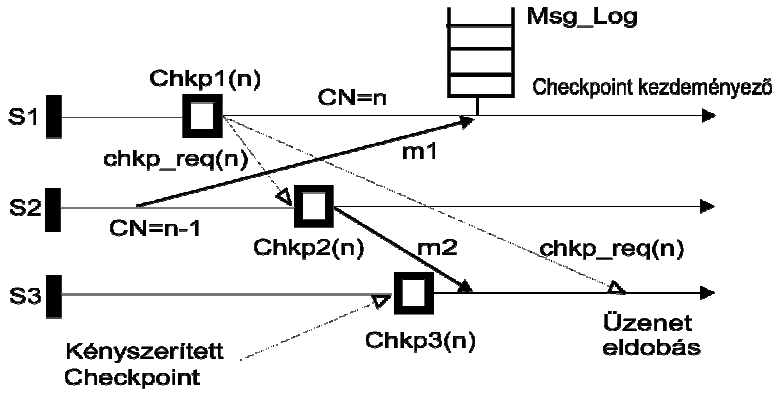
Ez az alapszintű algoritmus még nem gátolja meg az inkonzisztens üzenetek kialakulását: Lehetséges, hogy egy processz a lokális mentése után üzenetet küld egy másik processznek, amit az még a saját állapotmentése előtt (azaz a kezdeményezőtől hozzá küldött *checkpoint_request* üzenet megérkezése előtt) kap meg és dolgoz fel. Ugyancsak nem biztosítja ez az alapszintű algoritmus a függő üzenetek kezelését sem.

5.1.2.1 A (potenciálisan) inkonzisztens üzenetek kezelése

Az inkonzisztens üzenetek kezelésére a következő megoldások jöhetnek szóba:

- **Blokkoló technika**: A processzek nem küldhetnek alkalmazás üzeneteket a lokális mentés után addig, amíg a *global_checkpoint_done* üzenetet meg nem kapták. Így „bevárják” egymás állapotmentését, és elkerülhetők az inkonzisztens üzenetek. Ennek hátránya a nagy idő overhead, ami az alkalmazás üzenetek blokkolása okoz.

- **Nem-blokkoló technika FIFO csatornák esetén**: Ha egy processz az állapotmentése után (még a *global_checkpoint_done* üzenetet megelőzően) alkalmazás üzenetet akar küldeni egy másik processznek, akkor ez előtt elküld neki egy *checkpoint_request* üzenetet. A FIFO csatorna miatt ez biztosan hamarabb érkezik meg a címzetthez, mint az alkalmazás üzenet, így garantáltam megtörténik az állapotmentés akkor is, ha a kezdeményező *checkpoint_request* üzenete még nem ért volna ide.
- **Általános nem-blokkoló technika** (Silva & Silva algoritmusa, 1992): A processzek nyilvántartják, hogy hányadik állapotmentés után vannak, vagyis hányadik állapotmentési intervallumban tartanak. Az üzenetekhez csatolni kell az állapotmentési intervallum sorszámát, így a fogadó, ha az üzenetet megkapja, értesülhet a ”lekésett” állapotmentésről, és az üzenet feldolgozása előtt azt megteheti.
  - A potenciálisan inkonzisztens üzenet detektálható: Az üzenethez csatolt állapotmentési intervallum sorszáma nagyobb, mint a fogadónál az aktuális állapotmentési intervallum sorszáma (az üzenet hamarabb jutott el a fogadó processzhez, mint a kezdeményező *checkpoint_request* üzenete).
  - Az így felismert üzenet kezelhető: A fogadó processz állapotmentést végez az így detektált potenciálisan inkonzisztens üzenet *feldolgozása előtt*. Így elkerüli, hogy ez inkonzisztens legyen. Amikor a kezdeményezőtől megérkezik a *checkpoint_request* üzenet, akkor az már figyelmen kívül hagyható (hiszen megtörtént az állapotmentés).



A Silva-féle koordinált állapotmentés:
Az m2 üzenet feldolgozása előtt S3-nak állapotot kell mentenie

5.1.2.2 A függő üzenetek kezelése

Általános elv, hogy a függő üzenetet az állapotmentéshez kell csatolni, hogy az a visszalépés után visszajátszható legyen.

Ha a processzek az állapotmentési intervallumuk sorszámát a küldött üzenetekhez csatolják (ld. az általános nem-blokkoló technika alapötletét), akkor ez alapján a függő üzenetek is könnyen kezelhetők:

- Függő üzenet detektálása: Kisebb állapotmentési intervallum sorszám van hozzá csatolva, mint a fogadó processz állapotmentési intervallumának sorszáma.
- Függő üzenetek kezelés: El kell menteni üzenettárba, ahonnan visszalépés esetén újrajátszható.

Az állapotmentés kezdeményezője az állapotmentés lezárása előtt meg tud győződni arról, hogy *minden* függő üzenet az üzenettárba került-e, a következők szerint:

- Minden p_i processz nyilvántartja a megelőző mentési intervallumban küldött és a beérkezett üzeneteket egy-egy vektorban: $RECV_i[n]$ és $SENT_i[n]$.

- Ha egy processz állapotot ment, akkor elmenti ezeket a vektorokat is. Ezeket a *local_checkpoint_done* üzenethez csatolja, majd nullázza.
- A kezdeményező, ha minden csatolt vektort megkap, képes visszaszámolni, hogy melyek azok az üzenetek, amelyek el voltak küldve, de még nem érkeztek meg az állapotmentés előtt. Ezek számát egy $MISSING[n]$ vektorba gyűjti (minden processz esetére egy bejegyzés).

$$MISSING[i] = \sum_{k=1}^{n} SENT_k[i] - \sum_{k=1}^{n} RECV_i[k]$$

- Minden p_i processznek, amelyre $MISSING[i] > 0$, egy *catch_missing* üzenetet küld $MISSING[i]$ paraméterrel. Ha az itt megadott számú függő üzenetet a processz elmentette, akkor *missing_saved* üzenetet küld vissza.
- Ha minden szükséges *missing_saved* üzenet megérkezett a kezdeményezőhöz, akkor zárható le a globális állapotmentés a *global_checkpoint_done* üzenettel. A korábbi mentések eldobhatók, és az adott mentés használható visszaléptetéshez.

5.1.3 Kommunikációfüggő állapotmentés

Az állapotmentés és kommunikáció sorrendje (kapcsolata) rögzített a dominó-effektus elkerülése érdekében. Az üzenetekhez csatolt információ alapján a fogadó kiértékelheti, hogy az üzenet sérti-e ezt a kapcsolatot; ha igen, akkor állapotmentést végez az üzenet feldolgozása előtt.

- **Adaptív állapotmentés** (MRS modell): a dominó effektus elkerülhető, ha a következő kapcsolat fennáll minden processzre:

  $$(c_i, [r_i]*, [s_i]*)*$$

  ahol c_i az állapotmentés, r_i az üzenetfogadás és s_i az üzenetküldés az i-edik intervallumban. Ez azt jelenti, hogy üzenet küldése után és üzenet fogadása előtt mindenképpen állapotot kell menteni. Akkor hatékony ez a technika, ha az üzenetek küldése és fogadása jól blokkokra osztható (és közöttük történhet az állapotmentés).
- **Lokális mentés** (local recording): Elosztott állapotmentési technika, amelynek során a processzek vektorokat használnak az állapotmentésről való információ terjesztésére:
  - A processzek sorszámozzák az állapotmentésüket. Minden mentett globális állapotnak (visszalépési vonalnak) van ”tulajdonosa” (ez kezdeményezi az állapotmentést), ennek indexével és ennek lokális sorszámával van címkézve a globális állapot, pl. $\langle p_z, x \rangle$
  - Minden p_i processz tárol egy $CV_i[]$ vektort: $CV_i[j]$ a p_j processz lokális állapotmentésének sorszáma (amiről p_i tud, és amihez csatlakozott).
  - Az üzenetekhez kell csatolni a lokális $CV[]$ vektort.
  - A vett üzenet feldolgozása előtt meg kell vizsgálni ezt a csatolt információt.
    Legyen $RCV_j[]$ a kapott vektor (p_j tudása), $CCV_i[]$ a lokálisan tárolt vektor (p_i tudása).
    Minden z-re meg kell vizsgálni:
    1. Ha $RCV_j[z] > CCV_i[z]$, akkor ebből látszik, hogy p_z azóta állapotot mentett; így p_i-nek is állapotot kell mentenie, és ezt $\langle p_z, RCV_j[z] \rangle$ tulajdonba adni.
    2. Ha $RCV_j[z] < CCV_i[z]$, akkor ezt az üzenetet ”vissza kell dátumozni” (függő üzenet), azaz minden $\langle p_z, k \rangle$ tulajdonú állapotmentéshez adni, ahol $k > RCV_j[z]$
    3. Ha nincs lokális $\langle p_z, RCV_j[z] \rangle$ tulajdonú állapotmentés (kimaradottról később értesül, ld. az 1. pontban nem sorban jönnek létre), akkor ezt létre kell hozni,

mégpedig $\langle p_z, k \rangle$ állapotmentés átmásolásával (ahol k a legkisebb $k > RCV_j[z]$ sorszám), de az aktuális üzenet kizárásával.

5.2 Üzenettárolás alapú visszaléptetés

A visszaléptetéshez az elmentett állapot mellett a normál működés során eltárolt üzeneteket és eseményeket is felhasználják. Csak a meghibásodott processzt léptetik vissza, ezért *alkalmazási feltétel a hibaterjedés korlátozása*. A tárolt állapotra történő lokális visszaléptetéssel és a tárolt események és üzenetek megfelelő sorrendben történő visszajátszásával a hibás processz állapota helyreállítható. A visszaléptetett processz által újraküldött duplikált üzeneteket is detektálni kell.

Alapfeltevés:

- A processzeket nemdeterminisztikus események befolyásolják: ezek lehetnek bejövő üzenetek vagy belső események; az üzenetküldés nem tartozik ezek közé. (A továbbiakban az egyszerűség érdekében mind a nemdeterminisztikus belső eseményekre, mind a bejövő üzenetekre mint „üzenetekre" hivatkozunk.)
- Az üzenetek között a viselkedés determinisztikus.

Ez alapján:

- Független állapotmentéseket kell végezi és az állapotmentések között az üzeneteket tárolni kell. Az üzenettároláshoz annak *tartalma* mellett a *feldolgozási sorrendje* is lényeges a későbbi visszajátszás érdekében.
- Ha feltehetjük, hogy a hiba nem terjedt tovább, akkor *elegendő csak a hibás processzt visszaléptetni*. A helyreállítás az elmentett állapot visszaállításával majd a tárolt üzenetek "visszajátszásával" történik.
- A kiküldendő üzenetekről szintén a tároltak alapján dönthető el, hogy duplikáltak lennének-e (ekkor elhagyhatók).
- Külső környezettel való kommunikáció esetén egyébként is szükséges, hogy az üzenetek rögzítése megtörténjen (a környezet nem visszaléptethető).
- Domino effektus elkerülhető, így az állapotmentések koordinálása elmaradhat.

Probléma: Árva állapotok kialakulása lehetséges, ami több processz visszaléptetését teszi szükségessé.

- *Elveszett üzenet és elveszett állapot*: Ha az üzenettárolás nem stabil tárba történik, akkor ennek sérülése miatt egyes üzenetek tartalma vagy feldolgozási sorrendje elveszhet, azaz az üzenet visszalépés után nem játszható újra (ez az *elveszett üzenet*). Így az az állapot, ami ennek feldolgozása után elérhető, a visszalépés után nem alakulhat ki (ez az *elveszett állapot*).
- *Árva üzenet*: Olyan üzenet, amelyet az elveszett állapotból küldött a processz. Ez már egy „illegális" üzenetnek tekinthető, hiszen az az állapot, ahonnan küldte a küldője, a visszaléptetés miatt nem áll újra elő.
- *Árva állapot*: Olyan állapot, amely az árva üzenet hatására alakult ki. Ennek kiküszöböléséhez szükségessé válik visszalépés az árva üzenetet feldolgozott processzben is!

5.2.1 Optimista üzenettárolás

Az üzenetek tárolása aszinkron módon történik: az üzenet érkezésekor csak gyors (nem stabil) memóriába történik mentés, és ez csak időnként van stabil memóriába másolva. Az alkalmazás nincs addig feltartva, amíg a stabil tárba való mentés történik. Ez esetben optimista feltételezés, hogy hiba nem következik be a stabil memóriába való másolás előtt.

- Hiba esetén elveszhetnek a gyors memóriában tárolt üzenetek, így az előző futás nem játszható vissza determinisztikusan.

- Az elveszett üzenetek miatt árva állapotok keletkeznek, az árva állapottal rendelkező processzeket is vissza kell léptetni (”visszavonni” az elveszett üzenet hatását). Ehhez az elveszett üzenetek miatti *függőségeket tárolni kell*, visszalépéskor ez alapján kell meghatározni a visszalépési vonalat (konzisztens rendszerállapot, amelyben nincs árva állapot).
- Nem elég egyetlen állapotmentés processzenként, hanem több mentés és nemtriviális szemétgyűjtés szükséges.
- A külső környezet felé küldött üzenetek esetén egyetértés (vagy állapotmentés kényszerítése) szükséges a processzek között arról, hogy hiba esetén sem kerül visszavonásra az üzenet (csak helyreállítható állapotból történik üzenetküldés).

Visszalépés:

- Szinkron visszalépés: Minden processz tárolja a függőségi információt (ez az üzenetekhez rendelt intervallum-azonosítók segítségével történhet), hiba esetén kiszámolja a visszalépési vonalat és visszalép.
- Aszinkron visszalépés: A hibás processz visszalép és erről broadcast üzenetet küld; a többi processz a függőségi információ alapján eldönti, hogy árva-e. Ha igen, akkor visszalép, és erről is hasonló broadcast üzenetet küld. Ez így megy, amíg minden processz megfelelően vissza nem lépett.

5.2.2 Pesszimista üzenettárolás

A pesszimista alapfeltevés az, hogy bármikor előfordulhat hiba. Az elveszett üzenet elkerülése érdekében az üzenet addig nem dolgozható fel a fogadó által, amíg meg nem győződött róla, hogy az üzenet tárolása sikeresen megtörtént.

Az alapséma szerint az üzeneteket a feldolgozás előtt stabil tárba kell írni a fogadó processznél.

- Előny: Nem keletkeznek árva állapotok, így a processzek visszaléptetései közötti függőségek. A legutolsó elmentett állapot mindig használható, így nincs szükség többszörös állapotmentésre és egyszerű a szemétgyűjtés.
- Hátrány: A sikeres mentés bevárása (szinkron jellegű üzenettárolás) csökkenti a teljesítményt. Ez kiküszöbölhető a néhány speciális technikával.

5.2.2.1 Publishing: Központosított mentés busz rendszerek esetén

Busz jellegű kommunikációs struktúra (pl. Ethernet) esetén dedikált üzenettároló egység (processz) használható a buszon megjelenő üzenetek és azok sorrendjének tárolására. Újraindulás után innen lehet lekérni az üzeneteket illetve ellenőrizni a potenciálisan duplikált üzeneteket. Egy állapotmentés eldobása esetén az azt megelőző üzenetek törölhetők.

5.2.2.2 Küldőnél történő üzenettárolás (sender-based message logging)

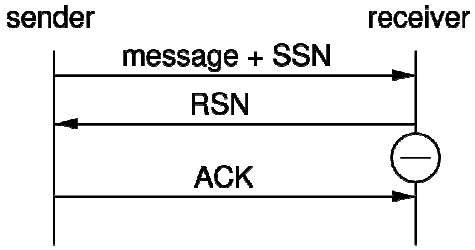
Az üzenetek tárolása a küldőnél történik, gyors (nem stabil) tárba. A küldőnél való tárolás előnye, hogy az üzenetátvitellel párhuzamosan történhet, nem jelent plusz késleltetést. Az üzenet tárolása máshol (a küldőnél) történik, mint ahol annak feldolgozása, így ha a fogadó processz visszalépésre kényszerül, nem lokálisan tárolt üzenetekre kell hagyatkoznia (ha az a feltételezésünk, hogy egyszerre csak egy hibát kell kezelnünk az elosztott rendszerben, akkor egyszerre nem sérülhet meg a fogadó processz és az üzeneteket tároló küldő processz).

Ugyanakkor figyelni kell arra, hogy az üzenetek feldolgozási sorrendjét is rögzíteni kell (és ezt a sorrendet csak a fogadó processz tudja), valamint kezelni kell a visszalépés során keletkező duplikált üzeneteket is.

A megvalósítás a következő protokoll szerint történik:

- Küldő: Küldi az üzenetet hozzácsatolva annak küldőnél nyilvántartott sorszámát (SSN, sender sequence number) valamint tárolja az üzenettárban.

- Fogadó: Veszi az üzenetet és visszaküldi a küldőnek az üzenet nála érvényes feldolgozási sorszámát (RSN, receiver sequence number),
  tárolja az adott küldőtől érkezett legnagyobb SSN sorszámot,
  felfüggeszti az alkalmazói üzenetek küldését (blokkol, így nem keletkezhet árva üzenet).
- Küldő: Tárolja az üzenethez tartozó, a fogadótól kapott RSN sorszámot az üzenettárban,
  majd egy nyugtázó (ACK) üzenetet küld vissza
- Fogadó: Veszi az ACK nyugtázást, ekkor bizonyosodhat meg róla, hogy az üzenetet és annak feldolgozási sorrendjét is sikerült elmenteni. Így befejezheti az alkalmazói üzenetek küldésének felfüggesztését.



Üzenettárolás

Hiba után egy p processz helyreállítása a következők szerint történik:

- Az elmentett legutolsó állapot visszaállítása.
- A p által vett üzenetek visszajátszása:
  - üzenetek kérése (broadcast kérés),
  - a küldők a náluk tárolt üzeneteket újraküldik, ezek feldolgozási sorrendjét az ott tárolt RSN megadja,
  - RSN-nel még nem címkézett üzenetek tetszőleges sorrendben küldhetők.
- A p által a visszalépés után újraküldött üzenetek sorsa:
  - p a visszalépés után az üzeneteket az előző futással egyező SSN sorszámmal küldi újra, ez alapján a vevők figyelmen kívül hagyják a duplikált üzeneteket a náluk tárolt (az adott p-re vonatkozó) SSN alapján;
  - a vevők ugyanazt az RSN sorszámot küldik vissza, mint az előző futáskor.

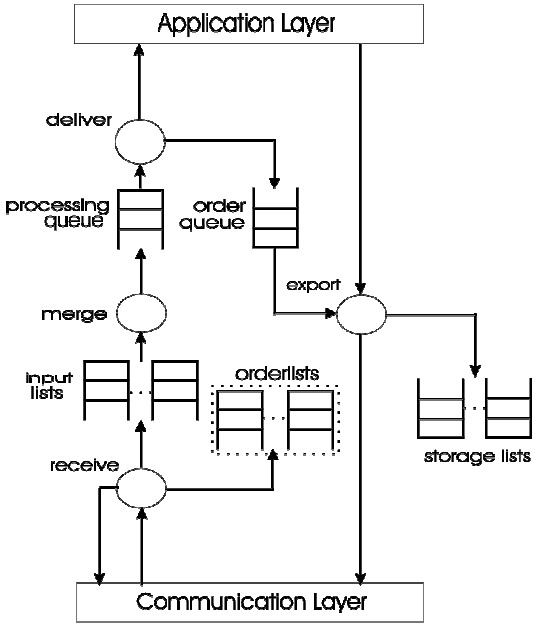
Ha a p processzben egy új állapotmentés történik:

- A küldők értesíthetők, hogy a p-nek szóló tárolt üzenetek törölhetők.

Összefoglalva az egyes sorszámok szerepét:

- A tárolt legnagyobb SSN: ez alapján felismerhetők a duplikált üzenetek (FIFO csatorna esetén; egyébként több SSN érték tárolására van szükség),
- RSN: az újrajátszott üzenetek sorrendezésére használható.

Megvalósítás: Helyreállítási réteg (recovery layer) a kommunikációs és alkalmazási réteg között (bejövő üzenetek feldolgozása, kimenő üzenetek és bejövő RSN értékek tárolása, legnagyobb bejövő SSN tárolása, SSN küldése, RSN küldése).



Az üzenettárolás implementációja

5.2.3 Oksági üzenettárolás

Cél, hogy elkerüljük a küldőnél történő üzenettárolás során bekövetkező blokkolást. Alapötlet az, hogy a feldolgozási sorrendre vonatkozó információt elosztottan tároljuk: ne a küldőnél, hanem éppen ott, ahol az árva állapot kialakulna, ha a feldolgozási sorrendre vonatkozó információ elveszik.

Alapséma:

- p_s küldő processz küldi az m üzenetet és az SSN_s^m sorszámot, m-et lokálisan üzenettárban tárolja (részleges mentés).
- p_r vevő processz fogadja és feldolgozza az üzenetet, a legnagyobb SSN_s-t tárolja, de nem blokkol és nem küldi vissza p_s-nek RSN_r^m-et.

Árva állapot kialakulásának veszélye:

- p_r az m feldolgozása után küld egy n üzenetet (ami m-től függ), ezt a p_q processz feldolgozza. Ha az m üzenet feldolgozási sorrendje nincs tárolva, p_q-ban árva állapot alakul ki p_r processz visszalépésekor.

Alapötlet:

- Mielőtt p_q feldolgozza az n üzenetet, előtte tárolja el az RSN_r^m-et (tehát éppen ott történjen tárolás, ahol a tárolás hiánya miatt árva állapot kialakulhat).
- Ehhez szükséges: p_r a küldött n üzenethez csatolja a $< p_s, SSN_s^m, RSN_r^m >$ információt is, ami alapján p_q-ban a tárolás megtörténhet.
- Minden, az m-től függő n-hez csatolni kell ezt az információt, amíg p_r meg nem győződött arról, hogy valahol a tárolás már megtörtént (pl. kapott választ valamelyik címzett processztől).

Hiba utáni helyreállítás:

- A hibás processz visszalép és broadcast üzenettel bekéri az információkat.
- A küldők az üzenetek tartalmát, a többi processz pedig a feldolgozási sorrendre utaló RSN sorszámokat küldi el, amelyek alapján az üzenetek visszajátszhatók.

Ez az ötlet általánosítható a következőképpen:

- Adott állapotban egy processz nyilvántartást vezet azokról az üzenetekről, amelyek befolyásolták ennek az állapotnak a kialakulását (az oksági reláció alapján). Ezek egy úgynevezett *előzmény gráfban* (antecedence graph, AG) történik: a csomópontok az üzenetek, az élek pedig az ”előbb történt” relációt fejezik ki (minden csomóponthoz két bejövő él tartozik: az előző üzenet a lokális processzben és a bejövő üzenet).
- Minden üzenethez csatolni kell az AG-t (pontosabban csak az adott processzhez előzőleg küldött üzenethez képest a különbséget). Az üzenettel vett AG-t mindig össze kell kombinálni a helyileg tárolttal. Időnként a gyors memóriában tárolt információkat az állapotmentéssel együtt a stabil tárba kell másolni. Az AG stabil tárba mentett részét már nem kell minden üzenethez hozzáfűzni. Külső környezet felé történő üzenetküldés előtt menteni kell az AG-t a stabil tárba.
- Hiba után össze kell gyűjteni az elosztottan tárolt AG részleteket, ebből összeállítani az AG-t és ez alapján visszajátszani a tárolt üzeneteket.

A Family Based Logging technika f hibát tolerál úgy, hogy az üzenetek $f+1$ processzben vannak tárolva.

5.2.4 Az állapotmentés alapú és az üzenettárolás alapú visszalépési technikák összehasonlítása

A overhead, az állapotmentések száma, a szemétgyűjtés és a visszalépés jellegzetességeinek áttekintése:

	Koordinálatlan	Koordinált	Kommunikáció függő	Optimista	Pesszimista	Oksági
	állapotmentés			üzenettárolás		
Overhead	kicsi	közepes	nagy lehet	kicsi	legnagyobb	nagyobb
Állapotmentések szükséges száma	több	1	több	több	1	1
Szemétgyűjtés	komplex	egyszerű	komplex	komplex	egyszerű	komplex
Visszalépés	többszörös visszalépés lehet (domino effektus)	utolsó mentett állapotra	eljárásfüggő	többszörös visszalépés lehet (árva üzenetek miatt)	utolsó mentett állapotra	utolsó mentett állapotra

A processzorok és kommunikációs hálózatok sebességnövekedése és a stabil tár (diszk) sebességének lemaradása miatt az optimum a független állapotmentés és az üzenettárolás felől a koordinált állapotmentés irányába mozdul el. Az üzenettárolás előnye ott jelentkezik, hogy (a hibaterjedés korlátozottsága esetén) elég csak a meghibásodott processzt visszaléptetni.

Az állapotmentés alapú visszalépő helyreállítás használatának egy jellegzetes példája az összetett (több, különböző résztvevő által végrehajtott részműveletből álló) műveletek *atomi jellegű végrehajtásának* biztosítása hibák esetén is: Egy elosztott művelet valósuljon meg, amelyet mégis egy (oszthatatlan) műveletként szeretnénk látni.

- A hibák hatásának kézben tartásához egy alapvető keretet ad, hiszen így a hibák (koncepcionálisan) csak összetett műveletek között történhetnek: Az atomi végrehajtás azt jelenti, hogy az összetett műveletben vagy sikerül minden részműveletet hibamentesen végrehajtani, vagy vissza kell vonni minden részműveletet (így a teljes műveletet is).
- A megvalósítás lépési a következők lehetnek: A processzek egy konzisztens állapotmentést végeznek az összetett művelet megkezdése előtt. A részműveletek végrehajtása után *konzisztens döntéshozatal* szükséges a részműveletek eredményéről (ld. commit protokollok). Ha bármelyik részműveletben hiba történt, akkor a döntés eredménye Abort lesz, ilyenkor a végrehajtás megkezdése előtt mentett állapotra történő visszalépés szükséges.

5.3 Előrelépő helyreállítás

Sok esetben vannak olyan résztvevők, amelyek nem léptethetők vissza, azaz hibák esetén a visszalépő helyreállítás nem megoldható (így ilyen módon atomi jellegű összetett műveletek sem képezhetők az előzőekben leírtak szerint). Ilyen esetek fordulnak elő pl. e-business jellegű folyamatokban, amikor egy hibázó vagy sikertelen partner miatt a hibátlanul dolgozók nem viselik el a visszaléptetést. Ez esetben javasolt technika az előrelépő helyreállítás a hibázó résztvevők (sikertelen részműveletek) kompenzálásával.

Ennek egy módszere a koordinált atomi műveletek (coordinated atomic actions, Web Service Coordinated Actions), ami a kivételkezelés (exception handling) koncepcióját terjeszti ki elosztott rendszerekre. Alapötletei:

- Az egymással együttműködő processzek (egymásba ágyazott) összetett műveleteket hajtanak végre.
- Az összetett műveletek esetén kooperatív előrelépő helyreállítás történik: Ha egy processzben, vagy beágyazott összetett műveletben hiba történik, akkor az összetett műveletben résztvevők együttesen kompenzáló műveleteket hajtanak végre (ez alkalmazásfüggő). A kompenzáló művelet függ attól, hogy milyen és mennyi hiba történt.
- A koordinált atomi műveletekben az esetleges megosztott erőforrások használata tranzakció-szerű: Hiba esetén a megosztott erőforrásokon visszalépő helyreállítás történhet (a kompenzáló műveletektől függően).
- A koordinált atomi műveletek „hívhatnak” más koordinált atomi műveleteket. Ha a hívás sikeres volt, akkor később (a hívó sikertelensége esetén) a hívott koordinált atomi műveletet (és annak erőforrásait) már csak kompenzáló műveletekkel lehet kezelni.

6 Távoli eljáráshívás (RPC)

A távoli eljáráshívás az egyszerű eljáráshívás kiterjesztése arra az esetre, amikor a hívó (kliens) által hívott fél (szerver) egy távoli szerveren helyezkedik el.

Hibamentes végrehajtás forgatókönyve:

- Hívás (kérés) küldése, beérkező kérés sorba állítása, (nyugta küldése,) a kérés feldolgozása, majd válasz küldése

Forgatókönyvek a hibák kezelésére:

- Kérés elveszik: Kérés újraküldése kellene.
- Nyugta elveszik: Újraküldés csak akkor, ha idempotens a kérés, vagy azonosítható az újraküldés (pl. közös köztesréteg van a kliens és a szerver esetén). Egyébként rá kell kérdezni a szerver állapotára.

  Mivel a kérés elvesztése illetve a nyugta elvesztése a kliens szempontjából nem különböztethetők meg, ezért ez utóbbi megoldást kell implementálni.
- Válasz elveszik: Ha a szerver nyugtát küldött, akkor a kliens biztos lehet benne, hogy a szerver a kérés feldolgozásába kezdett (ezért jó a nyugtázás). Rá kell kérdeznie a szerver állapotára (”are you alive?”); a szerver a tárolt válasz vagy újrafuttatás alapján válaszolhat.
- Szerver crash történik a kérés kiszolgálása közben: Kérés újraküldése kellene.
- Szerver crash történik a kérés kiszolgálása után: A szerveren befejeződött a kiszolgálás (pl. tranzakció) a crash előtt. Újraküldés itt is csak akkor történhet, ha idempotens a kérés, vagy azonosítható az újraküldés (pl. közös köztesréteg van a kliens és a szerver esetén). Egyébként rá kell kérdezni a szerver állapotára.

  Mivel ez a hiba és az előző a kliens szempontjából nem különböztethető meg, ezért ez utóbbi hibakezelési megoldást kell implementálni.

- Time-out a kliensnél a szerver lassúsága miatt: A kliens hibakezelést indít (pl. visszalép), ezáltal az eredeti hívás a szerveren árva hívás lesz. Ld. az előző forgatókönyvet: A kliensnek rá kell kérdeznie a szerver állapotára (mi történt ott a kéréssel).
- Kliens crash hiba, majd visszalépéses helyreállítás: Duplikált üzenet keletkezik. Megoldás lehet a szerver visszaléptetése (sokszor nem megoldható) vagy lokális üzenet naplózás a kliensben a duplikált üzenetek kiszűrésére.

RPC szemantikák: Mit garantálhatunk hibamentes (normál) illetve hibás (abnormál) végrehajtás esetén:

Szemantika	Normál befejezés	Abnormál befejezés
At least once (legalább egyszer)	Legalább 1 végrehajtás (több is lehet), idempotens kérésekre	Többszörös végrehajtás is lehetséges (0, 1, részben, többször)
At most once (legfeljebb egyszer)	Pontosan 1 végrehajtás	Nincs többszörös végrehajtás (0, 1, részben)
Exactly once (pontosan egyszer)	Pontosan 1 végrehajtás	Egyszeres végrehajtás (helyreállítással)

Replikált RPC szerverek használatának motivációja: Szerver crash esetén ne kelljen a meghibásodott szerver helyreállítására várni (lekérdezgetéssel), hanem aktív tartalék léphessen a helyére. A replikált szerverek kezelése többféle módon történhet:

- A kliens küld multicast üzenetet: Ez a kliens számára nem transzparens. A beérkező többszörös válaszokról döntenie kell (replikált kliensek esetén replika determinisztikus döntés szükséges). Független kliensek esetén teljes sorrendezés, együttműködő kliensek esetén teljes és oksági üzenet sorrendezés szükséges.
- Gateway használata az üzenetek többszörözésre: Ez a kliens számára transzparens, az üzenet sorrendezés a gateway után szükséges.

7 Összefoglalás

Ebben a fejezetben elosztott rendszerek megvalósításához szükséges, a szolgáltatásbiztonság szempontjából is releváns alapszolgáltatásokat tekintettünk át:

- Tagsági kép nyilvántartása
- Konszenzus
- Atomi döntéshozatal
- Események és üzenetek sorrendezése
- Elosztott helyreállítás (visszalépő és előrelépő)
- Távoli eljáráshívás

Összefoglalásként érdemes elgondolkodni a következő kérdéseken:

- Milyen alapszolgáltatások szükségesek a hibatűréshez használatos tipikus sémák megvalósításához (pl. aktív redundancia, passzív redundancia, visszalépő helyreállítás, előrelépő helyreállítás megvalósításához)?
- Milyen függések jelenhetnek meg az egyes alapszolgáltatások között, azaz egy alapszolgáltatás megvalósítása igényel-e másik alapszolgáltatást (pl. szükséges-e a csoportkommunikáció megvalósításához a tagsági kép szolgáltatás, és viszont)?