Eötvös Loránd Tudományegyetem
Informatikai Kar
Programozási Nyelvek és Fordítóprogramok tanszék

Szinkronizáció osztott rendszerekben

Réti Katalin

Témavezető: Dr. Horváth Zoltán

Budapest

2004. június 15.

Tartalomjegyzék

Bevezetés....3
A rendszer modellje....4
Aszinkron osztott rendszerek....4
Logikai órák....6
Globális állapot....8
A globális állapot rögzítése....11
Osztott felvételek....11
Chandy-Lamport algoritmus....11
Acharya-Badrinath algoritmus....18
A Chandy-Lamport és az Acharya-Badrinath algoritmus összehasonlítása....20
Hullám sorozat FIFO modellre....22
Hullám sorozat nem FIFO modellre....24
A folyamatok színezése....27
Az öt algoritmus....31
Osztályozás alkalmazhatóság szerint....31
Osztályozás működés szerint....32
Melyiket használjuk?....34
A globális állapot alkalmazása....35
Stabil tulajdonság....35
Holtpont....36
Ellenőrzési pontok és visszagörgetés....37
Összefoglalás....41
Irodalomjegyzék....42

Bevezetés

Egy osztott rendszer számítógépekből áll. Ezek a gépek teljesen függetlenek egymástól, nincs közös memóriájuk, nincsenek közös változóik. A kommunikáció üzeneteken keresztül történik. Az üzenetek csatornákon keresztül jutnak el az egyik géptől egy másikhoz. Egy üzenet feladása és megérkezése között mindenképpen eltelik valamennyi idő.

Mivel a gépek függetlenek egymástól, közös órájuk sincsen. Sokszor van arra szükség, hogy meghatározzuk egy osztott rendszer globális állapotát, azaz az egyes gépek állapotának unióját. Közös óra hiányában nem tudjuk elérni, hogy a gépek pontosan egy időben rögzítsék az állapotukat. De nem is ez a fontos, hanem az, hogy a globális állapot értelmes legyen, azaz olyan, amilyet egy külső megfigyelő rögzíthetett volna. Ebben a dolgozatban olyan algoritmusokat fogunk ismertetni, amelyek osztott rendszerek értelmes globális állapotát rögzítik. Előbb azonban meg kell ismerkednünk a logikai idő fogalmával, mert az fontos szerepet játszik az osztott rendszerek szinkronizációjában, így a globális állapot rögzítésében is.

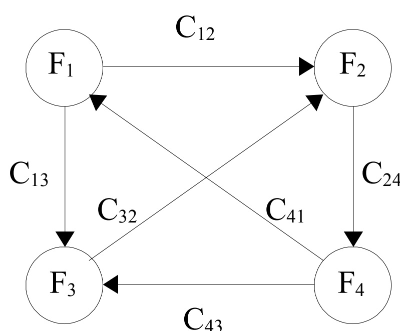
A dolgozat felépítése ennek megfelelően a következő. A második fejezetben bemutatjuk az osztott rendszer modelljét, megismerkedünk a logikai idő és a globális állapot pontos fogalmával. A harmadik fejezetben definiáljuk az osztott felvétel fogalmát, majd részletesen ismertetünk néhány olyan módszert, amelyekkel globális állapotot rögzíthetünk. A negyedik fejezetben különböző szempontok szerint összehasonlítjuk a bemutatott algoritmusokat. Az ötödik fejezetben mutatunk néhány példát arra, hogy mire használható a globális állapot. A hatodik fejezet összefoglalás.

A dolgozat célja az, hogy bemutassuk, hogy milyen módszerekkel lehet globális állapotot rögzíteni. Továbbá az, hogy megvizsgáljuk, hogy melyik algoritmus mikor használható, és milyen egyéb szempontok alapján lehet őket csoportosítani.

A rendszer modellje

Aszinkron osztott rendszerek

Egy aszinkron osztott rendszer folyamatokból áll, amelyeket egyirányú csatornák kötnek össze. Mindegyik folyamat tud mindegyikkel kommunikálni, de nem feltétlenül közvetlenül. Egy ilyen rendszert egy irányított gráffal lehet ábrázolni, ahol a gráf csúcsai a folyamatok, az irányított élek pedig az egyirányú csatornák (1. ábra). Ez a gráf erősen összefüggő, vagyis bármely csúcsából vezet irányított út bármelyik másikba. A folyamatok a csatornákon keresztül küldhetnek egymásnak üzeneteket, ami a kommunikáció egyetlen eszköze az ilyen rendszerekben. A csatornák megbízhatóak, egy üzenet bármilyen hosszú, de csak véges ideig tartózkodhat egy csatornán.



1. ábra: Egy osztott rendszer gráffal ábrázolva

Osztott számításnak nevezzük azt, amikor több folyamat együtt végrehajt egy osztott programot. Egy folyamat önmagában események sorozatát hajtja végre. Egy osztott rendszerben háromféle eseményt különböztetünk meg:

- belső esemény,
- üzenet küldése és
- üzenet fogadása.

Mindegyik eseményt egy adott F folyamat hajtja végre. Egy belső esemény csak az F folyamat állapotát változtathatja meg. Egy *m* üzenet küldése megváltoztathatja F állapotát, és mindenképpen megváltoztatja a C csatorna állapotát, amelyen keresztül F elküldi az üzenetet, hiszen a csatornán lévő üzenetek sorozata kibővül *m*-mel. Az *m* üzenet fogadása F folyamat által megváltoztathatja F állapotát és megváltoztatja a C csatorna állapotát, amelyen keresztül küldték, hiszen *m* eltűnik a C-n levő üzenetek sorozatának éléről.

Sokszor szükség van arra, hogy meg tudjuk állapítani, hogy két esemény közül melyik történt előbb. Ennek érdekében Leslie Lamport bevezette az "előbb történt"

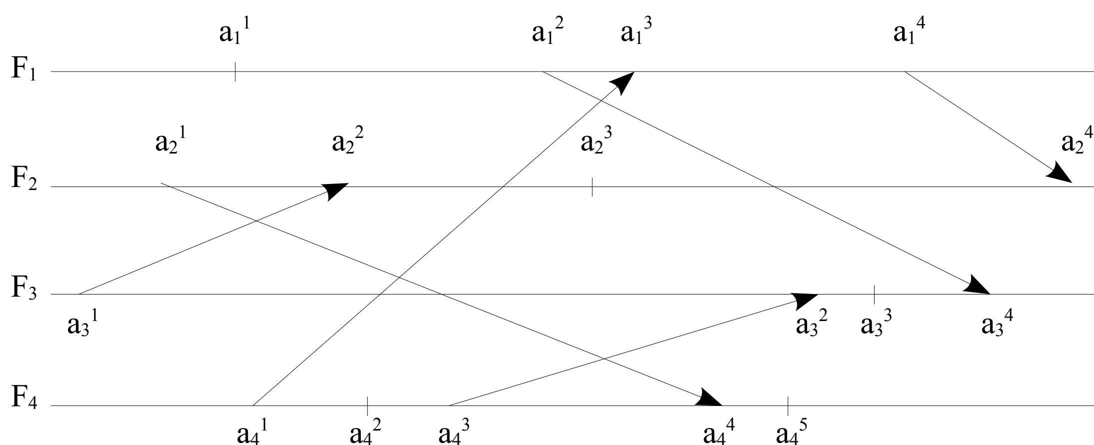
relációt [9], amit $\rightarrow$-lal jelölünk és a következőképpen definiáljuk. A $\rightarrow$ relációt az események halmazán értelmezzük. Ez a legkisebb olyan reláció, ami kielégíti a következő három feltételt.

(1) Ha a és b ugyanannak a folyamatnak az eseményei és a megelőzi b-t, akkor $a \rightarrow b$.
(2) Ha a egy üzenet küldése egy folyamat által, b pedig ugyanannak az üzenetnek a fogadása egy másik folyamat által, akkor $a \rightarrow b$.
(3) Ha $a \rightarrow b$ és $b \rightarrow c$, akkor $a \rightarrow c$, vagyis $\rightarrow$ tranzitív.

Feltesszük, hogy bármely a eseményre nem teljesül $a \rightarrow a$, vagyis $\rightarrow$ irreflexív. A $\rightarrow$ reláció tehát szigorú részbenrendezés, mivel irreflexív és tranzitív. Ha a és b eseményekre se $a \rightarrow b$ se $b \rightarrow a$ nem teljesül, akkor azt mondjuk, hogy a és b konkurens események, és ezt $a \| b$-vel jelöljük.

Annak megállapítása, hogy két esemény közül melyik történt előbb azért fontos, mert az az esemény, amelyik előbb történt lehet, hogy befolyásolta a másikat. Például, ha egy üzenetet nem küldünk el, az nem is érkezik meg. Tehát egy üzenet küldése nem csak időben, hanem okozatilag is megelőzi az üzenet fogadását. Ugyanakkor az, hogy az a és b események között fennáll az $a \rightarrow b$ reláció, nem feltétlenül jelenti azt, hogy az a esemény befolyásolja a b-t, csak azt, hogy befolyásolhatja. Ha az a és b események között nem áll fenn az $a \rightarrow b$ reláció, akkor a nem befolyásolhatja b-t.

Egy osztott számítást egy tér-idő diagramon lehet ábrázolni, ahol a folyamatok egymással párhuzamos, vízszintes egyenesek, a belső események pontok a megfelelő egyenesen, az üzenetek pedig nyilak, amelyek a küldő folyamat egyeneséről indulnak és a fogadó folyamat egyenese felé mutatnak. $a \rightarrow b$ pontosan akkor áll fenn, ha a diagramon el lehet jutni a-ból b-be egyenesek és nyilak mentén. A 2. ábrán egy tér-idő diagram látható. Az ábra szerint például $a_4^1 \rightarrow a_2^4$.



2. ábra: Tér-idő diagram

Logikai órák

Minden feltétel adott ahhoz, hogy bevezessük az órák használatát. Az előző fejezetben bevezetett "előbb történt" reláció segítségével minden eseményhez hozzá fogunk rendelni egy időpontot. Mielőtt ezt megtennénk, megvizsgáljuk, hogy milyen feltételnek kell majd megfelelniük ezeknek az időpontoknak. Legyen C a globális időt reprezentáló függvény. A következő, úgynevezett óra feltételnek kell teljesülnie:

bármely a és b eseményre, ha $a \rightarrow b$, akkor $C(a)<C(b)$.

A feltétel ellenkező irányú teljesülését nem várhatjuk el, mert az azt jelentené, hogy bármely két konkurens esemény egy időben történik.
Minden F_i folyamat minden a eseményéhez hozzárendelünk egy időpontot, $C_i(a)$-t. C_i az F_i folyamat órája. A globális időt reprezentáló függvényt a következőképpen definiáljuk:

$C(a)=C_i(a)$, ha a az F_i folyamat egy eseménye.

A logikai órák rendszerére a következő feltételek kell, hogy teljesüljenek.

(C1.) Ha a és b F_i folyamat eseményei, és a megelőzi b-t, akkor $C_i(a)<C_i(b)$.
(C2.) Ha a egy üzenet küldése az F_i folyamat által, b pedig ugyanannak az üzenetnek a fogadása az F_j folyamat által, akkor $C_i(a)<C_j(b)$.

Nyilvánvaló, hogy ha a fenti két feltétel teljesül, akkor az óra feltétel is teljesül.
Most megvizsgáljuk azt, hogy hogyan kell a logikai órák értékét úgy meghatározni, hogy az óra feltétel teljesüljön. Úgy fogjuk a C_i függvények értékeit meghatározni, hogy kielégítsék a (C1.) és (C2.) feltételeket. A (C1.) feltétel egyszerűen kielégíthető.

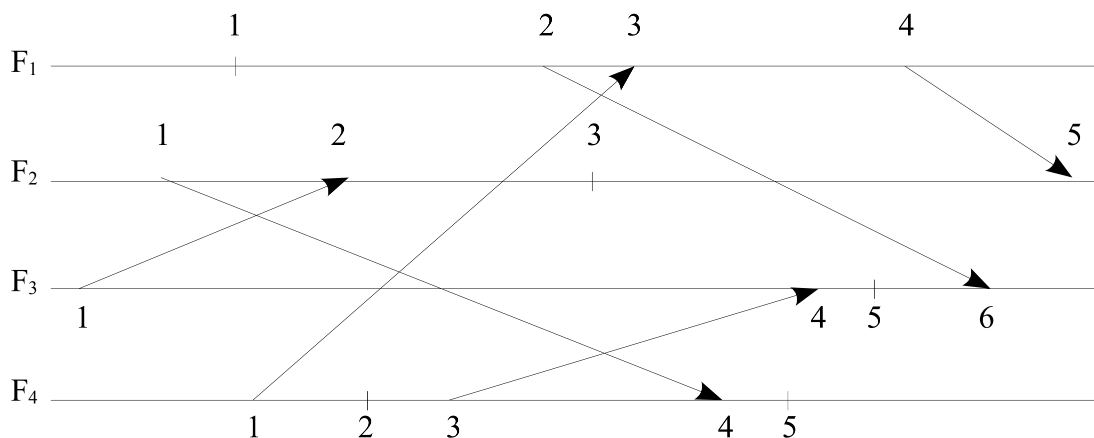
(SZ1.) Minden F_i folyamat megnöveli C_i értékét minden két egymást követő esemény között.

A (C2.) feltétel teljesüléséhez minden m üzenethez hozzárendelünk egy T_m időbélyegzőt, amelyre a következő szabály teljesül.

(SZ2.) a) Ha az a esemény az m üzenet küldése az F_i folyamat által, akkor m időbélyegzője $T_m=C_i(a)$. b) Amikor az F_j folyamat kap egy m üzenetet T_m időbélyegzővel, C_j értékét úgy állítja be, hogy az nagyobb legyen a jelenleginél vagy egyenlő vele, és nagyobb legyen T_m-nél. A 3. ábra az időbélyegzők használatát szemlélteti.

(SZ2.) nyilvánvalóan biztosítja (C2.) teljesülését, hiszen amikor egy folyamat kap egy

üzenetet nagyobbra állítja a logikai óráját, mint az üzenet időbélyegzője, azaz az időpont, amikor az üzenetet küldték.



3. ábra: Lamport időbélyegző

Az (SZ1.) és (SZ2.) szabályok tehát biztosítják az óra feltétel teljesülését. Ezekkel a szabályokkal a C_i függvények logikai órák egy konzisztens rendszerét alkotják.

A fentiekben bevezetett időfogalomnak van egy hiányossága. Ha tudjuk, hogy valamely *a* és *b* eseményekre $C(a)<C(b)$, nem tudjuk eldönteni, hogy $a \rightarrow b$ vagy $a||b$. Ahhoz, hogy ez eldönthető legyen a következő, úgynevezett erős óra feltételnek kell teljesülni:

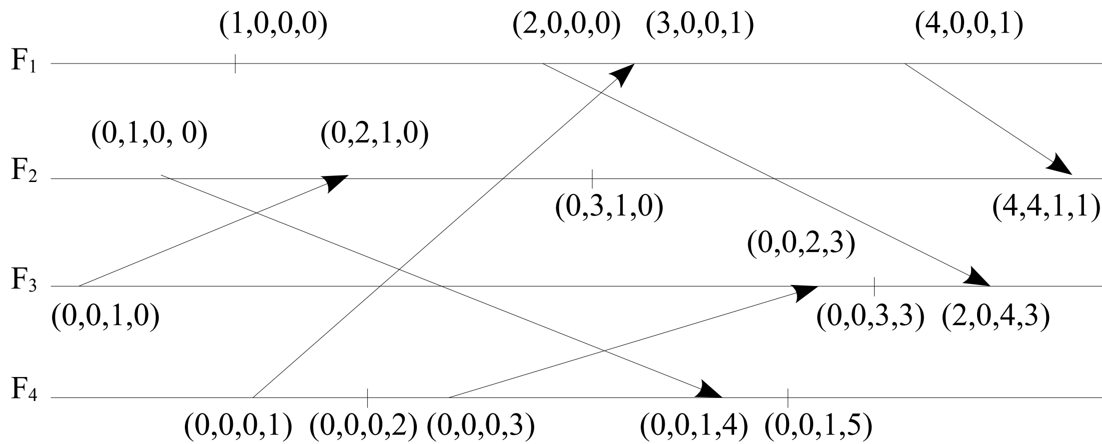
bármely *a* és *b* eseményre, $a \rightarrow b$, akkor és csak akkor, ha $C(a)<C(b)$.

Ezt a feltételt a vektor időbélyegzők bevezetésével tudjuk kielégíteni. Mindegyik folyamat karban tart egy vektort, és amikor üzenetet küld, ezt a vektort csatolja hozzá időbélyegzőnek. Az F_i folyamat V_i vektora *i.* komponensének az értéke a helyi időt reprezentálja, vagyis ezt F_i minden benne bekövetkezett esemény alkalmával növeli. A többi komponenst akkor változtatja, amikor üzenetet kap. A pontos szabályok a következők.

(VSZ1.) Ha *a* belső esemény az F_i folyamatban vagy egy üzenet küldése, akkor F_i megnöveli $V_i[i]$ értékét.
(VSZ2.) Ha *a* az *m* üzenet fogadása az F_i folyamatban, akkor $V_i:=\max(V_i, T_m)$ és F_i megnöveli $V_i[i]$ értékét. A 4. ábra a vektor időbélyegzők használatát szemlélteti.

T_m most is az *m* üzenet időbélyegzője, ami ezúttal egy vektor, és megegyezik a küldő folyamat vektorával a küldés időpontjában. A folyamatok mindig közvetlenül az események bekövetkezte előtt változtatják meg a vektorukat.

Két n hosszúságú vektort a következőképpen hasonlíthatunk össze.



4. ábra: Vektor időbélyegző

$v \leq w \Leftrightarrow \forall\ 1 \leq i \leq n: v[i] \leq w[i]$
$v < w \Leftrightarrow v \leq w \wedge \exists\ i\ \ 1 \leq i \leq n: v[i] < w[i]$
$v \| w \Leftrightarrow \neg\ (v < w) \wedge \neg\ (w < v)$

Legyen V(*a*) az *a* esemény, V(*b*) pedig a *b* esemény időbélyegzője. Ekkor

(1) $a \rightarrow b$, akkor és csak akkor, ha V(*a*)<V(*b*), és
(2) $a \| b$, akkor és csak akkor, ha V(*a*)||V(*b*).

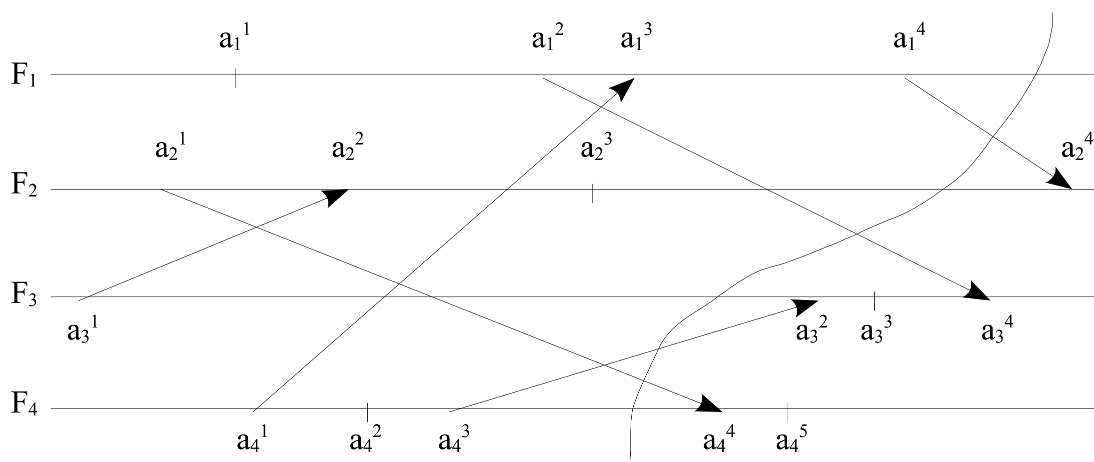
Az (1) állítás az erős óra feltétel, a második pedig annak következménye.

Globális állapot

Egy folyamat a végrehajtása kezdetén valamilyen állapotban van, majd a végrehajtás során az események hatására változik az állapota. Formálisan egy folyamat állapota leírható például a kezdeti állapotával, és az általa végrehajtott események sorozatával. Egy esemény megváltoztatja az őt végrehajtó folyamat állapotát. Egy folyamat lokális történelme az általa végrehajtott események sorozata. Legyen h_i az F_i folyamat lokális történelme. Ekkor $h_i = a_i^1 a_i^2 ....$, ahol a_i^1 az első F_i folyamat által végrehajtott esemény, a_i^2 a második, és így tovább. Ezekre az eseményekre fennáll, hogy $a_i^j \rightarrow a_i^k$, ha j<k, mivel ugyanannak a folyamatnak az eseményei időrendi sorrendben. Egy osztott számítás globális történelme a lokális történelmek uniója, azaz $H = h_1 \cup ... \cup h_n$, ha a rendszerben n folyamat van. Egy osztott számítás egy V vágása a H halmaz egy olyan részhalmaza, amely tartalmazza minden h_i sorozat egy kezdő szeletét. Azok az események, amelyek ezen kezdőszeletek utolsó elemei, a vágás határát alkotják. Egy vágást jól lehet szemléltetni a tér-idő diagramon úgy, hogy húzunk egy vonalat, amelyik pontosan egyszer metszi mindegyik folyamat

egyenesét. A vonaltól balra elhelyezkedő események a vágás elemei. A vonal metszheti üzenetek nyilait is. Akkor mondjuk, hogy egy vágás konzisztens, ha az őt reprezentáló vonal csak úgy metszik nyilakat, hogy azoknak a kiinduló pontja a vonaltól balra, a végpontja pedig a vonaltól jobbra helyezkedik el. Az 5. ábra egy konzisztens vágást mutat. Formálisan a V vágás akkor konzisztens, ha minden *a* és *b* eseményre

$(b \in V) \wedge (a \rightarrow b) \Rightarrow a \in V.$



5. ábra: Konzisztens vágás

Egy vágás határa minden folyamat egy állapotát tükrözi. Legyen a V vágás határa (a_1, a_2, ..., a_n). Ekkor az F_i folyamat abban az állapotban van, amelybe az a_i esemény hatására került. Ezzel el is érkeztünk a globális állapot fogalmához. Egy osztott számítás egy adott időpillanatbeli globális állapota az összes folyamat állapota és a csatornák állapota. Az F_i folyamatból F_j folyamatba vezető C_{ij} csatorna állapota egy adott pillanatban az addig a pillanatig rajta az F_i által elküldött és az F_j által még nem fogadott üzenetek sorozata.

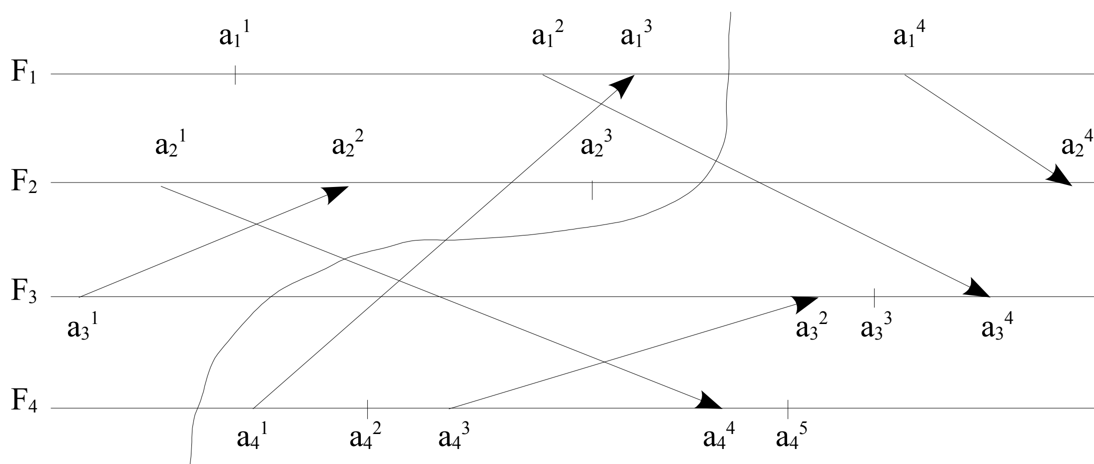
Egy vágás határa meghatároz egy globális állapotot. Egy inkonzisztens vágás határa olyan globális állapotot határoz meg, amely a valóságban sosem fordulhatna elő, hiszen nem lehet a rendszerben olyan üzenet, amit egy folyamat már fogadott, de még nem küldték el. A 6. ábra inkonzisztens vágást mutat.

Egy osztott rendszer adott pillanatbeli globális állapotának meghatározása sok mindenre használható. Legyen Φ egy logikai értékeket felvevő függvény, ami a rendszer globális állapotainak halmazán van értelmezve. Akkor mondjuk, hogy Φ stabil tulajdonság, ha

$\Phi(S) \wedge$ S' elérhető S-ből $\Rightarrow \Phi(S')$.

Vagyis ha a Φ tulajdonság egy osztott számítás egy adott pillanatában fennáll, akkor fennáll a továbbiakban is. Stabil tulajdonság például, hogy egy osztott számítás

befejeződött vagy az, hogy egy osztott számítás holtpontra jutott.



6. ábra: Inkonzisztens vágás

A globális állapot meghatározása arra is jó, hogy egy rendszert lehessen újraindítani, ha valamilyen okból leáll. Ha egy rendszer időről időre rögzíti a globális állapotát, és valamilyen hiba folytán leáll, könnyen lehet újraindítani. Minden folyamatot vissza kell görgetni abba az állapotba, amelyiket a legutóbbi rögzített globális állapot tartalmaz és onnan folytatni a végrehajtást. A megoldás nem ennyire egyszerű, de erről majd a későbbiekben lesz szó.

A globális állapot rögzítése

Osztott felvételek

Egy osztott felvétel egy osztott rendszer adott pillanatbeli globális állapota meghatározásának az eredménye. Mivel nincsen közös globális óra, nem lehet minden folyamat és csatorna állapotát ugyanabban a pillanatban rögzíteni. Mégis arra kell törekedni, hogy az osztott felvétel értelmes legyen, azaz olyan amit egy külső megfigyelő egy időpillanatban készíthetett volna. Az osztott felvételt készítő algoritmust az osztott számítás fölé kell helyezni úgy, hogy az algoritmus ne befolyásolja a számítást.

Az osztott felvételt készítő algoritmusokat alapvetően két csoportra lehet osztani a szerint, hogy feltételezik-e a csatornák FIFO tulajdonságát vagy sem. Egy csatorna akkor FIFO (First In-First Out) tulajdonságú, ha a rajta keresztül küldött üzenetek olyan sorrendben érkeznek meg a fogadóhoz, amilyen sorrendben elküldték őket. Mi először a FIFO tulajdonságot feltételező algoritmusokkal foglalkozunk.

Chandy-Lamport algoritmus

Ez az algoritmus [3] úgy működik, hogy egy vagy több folyamat spontán rögzíti az állapotát, majd küld egy jelzést minden kimenő csatornáján. Amikor egy folyamat először kap jelzést, rögzíti az állapotát. Amikor egy folyamat jelzést kap, ami nem az első, rögzíti annak a csatornának az állapotát, amelyiken a jelzés jött. Az algoritmus pontos leírását két szabály adja, amelyek a következők.

Jelzés küldő szabály

az F folyamat küld egy jelzést minden belőle kiinduló C csatornán keresztül miután rögzítette a saját állapotát és mielőtt további üzeneteket küldene C-n keresztül

Jelzés fogadó szabály

amikor az F folyamat kap egy jelzést a C csatornán keresztül

if F még nem rögzítette az állapotát **then**
    begin F rögzíti az állapotát;
        F rögzíti a C állapotát, mint üres sorozat
    end
else F rögzíti a C állapotát, mint azon üzenetek sorozata, amelyeket F fogadott miután rögzítette az állapotát és mielőtt megkapta a jelzést a C-n keresztül.

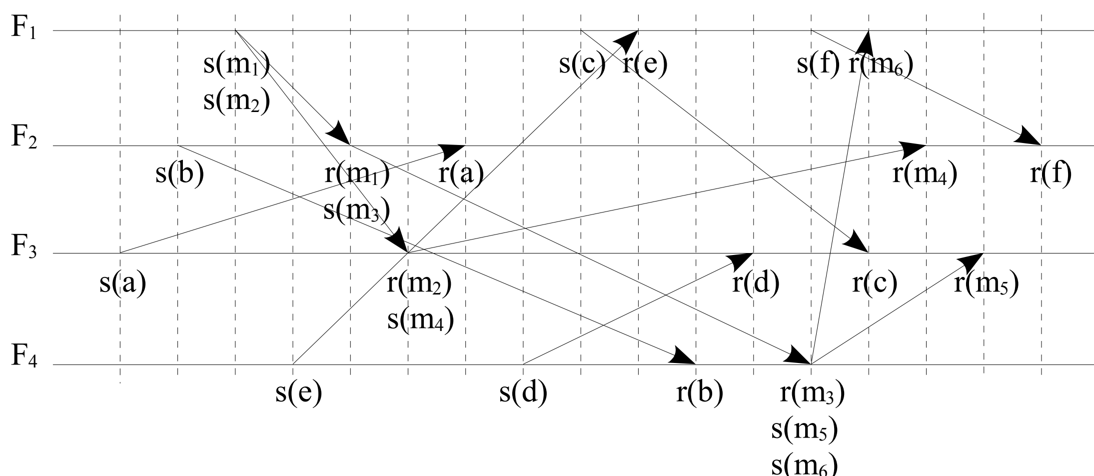
A két szabály biztosítja, hogy ha minden csatornán keresztül érkezik egy jelzés, akkor mindegyik folyamat rögzíti a saját és a felé irányuló csatornák állapotát. Mivel bármely két folyamat között vezet irányított út, és az üzenetek véges ideig maradnak a csatornán, ezért minden folyamat minden felé érkező csatornán kap egy jelzést véges időn belül, így az algoritmus befejeződik. Az algoritmus folyamán minden csatornán pontosan egy jelzés megy keresztül.

Az algoritmus befejeződése után össze kell gyűjteni a folyamatok által rögzített lokális- és csatornaállapotokat, hogy fel lehessen építeni a globális állapotot. Egy egyszerű algoritmus erre az, hogy mindegyik csatorna elküldi a kimenő csatornáin az általa rögzített állapotokat. Amikor egy folyamat először kap ilyen információt, lemásolja és tovább küldi a kimenő csatornáin. Minden rögzített állapot el fog jutni mindegyik folyamathoz véges időn belül, így mindegyik meg tudja határozni a globális állapotot. Egyéb algoritmusok is léteznek az információk összegyűjtésére.

A fent leírt algoritmus konzisztens globális állapotot rögzít. Egy globális állapot akkor konzisztens, ha nem rögzít olyan üzenetet megkapottnak, amelyet nem küldtek el. Jelöljük az m üzenet küldését *s(m)*-mel, fogadását *r(m)*-mel, a rögzített globális állapotot pedig S'-vel. Ezekkel a jelölésekkel a rögzített globális állapot konzisztens, ha

$$r(m) \in S' \Rightarrow s(m) \in S'.$$

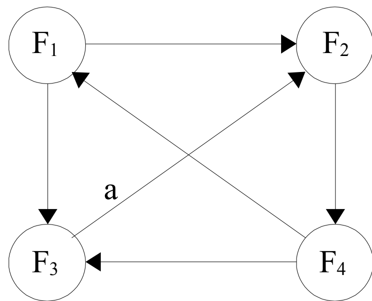
Legyen m egy üzenet, amit az F_i folyamat küld az F_j folyamatnak a C_{ij} csatornán keresztül. Tegyük fel, hogy $s(m) \notin S'$. Ez azt jelenti, hogy az F_i folyamat az előtt rögzítette az állapotát és küldött jelzést a C_{ij} csatornán, hogy az m üzenetet elküldte volna. A csatornák FIFO tulajdonsága miatt ebből következik, hogy az F_j folyamat előbb kapta meg a jelzést F_i-től, mint az m üzenetet, vagyis előbb rögzítette az állapotát. Tehát $r(m) \notin S'$, amiből már következik, hogy a rögzített globális állapot konzisztens.



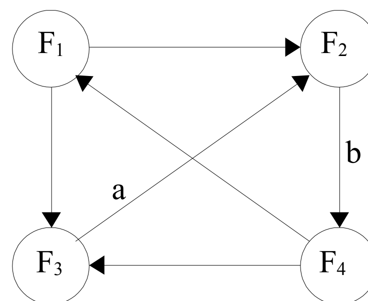
7. ábra: Tér-idő diagram a példához

Tekintsünk egy konkrét példát az algoritmus működésére. A 7. ábrán látható egy osztott számítás tér-idő diagramja, a 8. ábra pedig ugyanezen számítás lefutása alatt a rendszer által felvett globális állapotokat mutatja. Az ábrán jelöljük a jelzéseket is. Külön tüntetjük fel azokat a globális állapotokat is, amik között csak a jelzések helyzetében van különbség és a számítás üzeneteinek helyzetében nincs. Ezek elvileg nem különböző globális állapotok, hiszen az algoritmus (ami a jelzéseket küldi) az osztott számítástól függetlenül fut és nem befolyásolja azt.

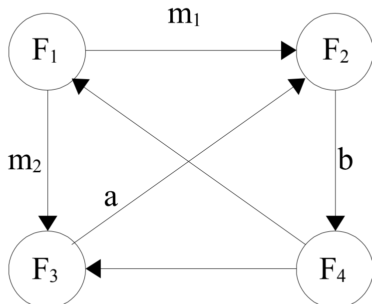
1.



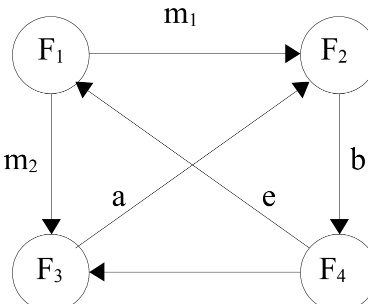
2.



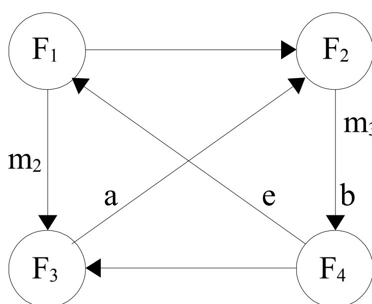
3.



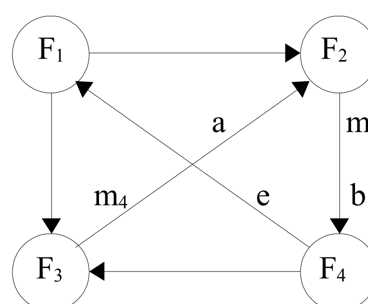
4.



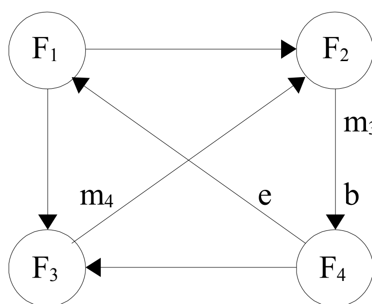
5.



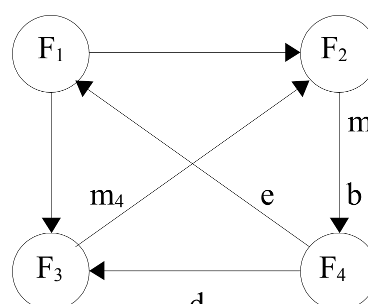
6.



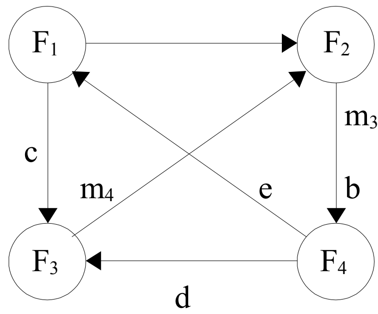
7.



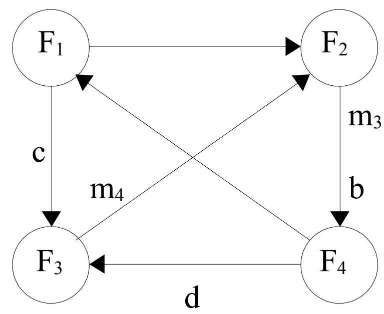
8.



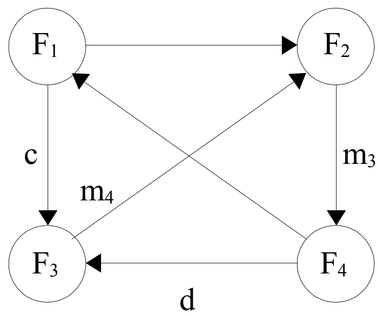
9.



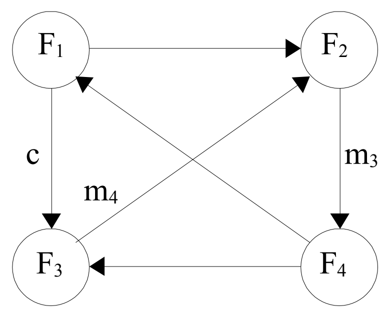
10.



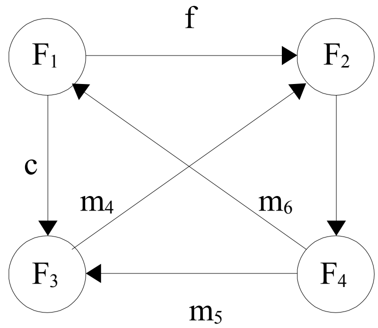
11.



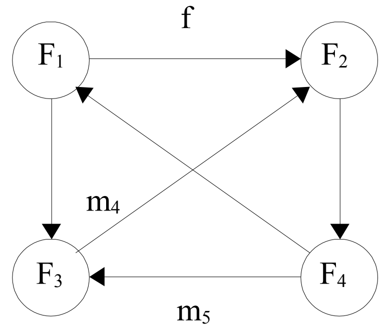
12.



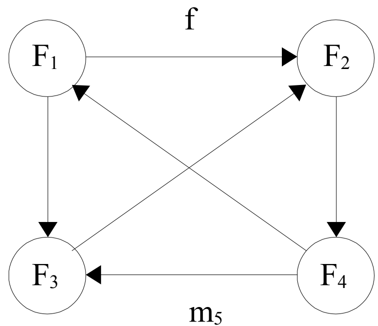
13.



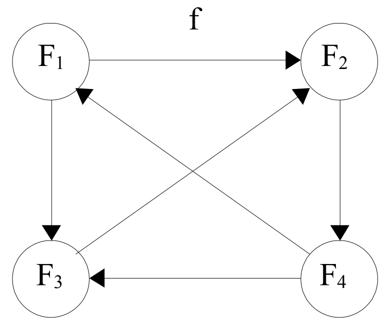
14.



15.



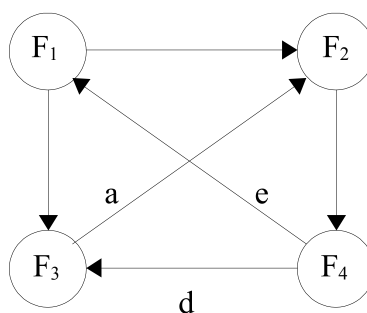
16.



8. ábra: Globális állapotok

Az ábrán a 3. állapotban az F_1 folyamat elindítja az algoritmust. Rögzíti a saját lokális állapotát és elküldi az m_1 jelzést az F_2 folyamatnak, és az m_2 jelzést az F_3 folyamatnak. A 4. állapot azzal zárul, hogy az F_2 folyamat megkapja az m_1 jelzést, rögzíti a saját állapotát, és a C_{12} csatorna állapotát üres sorozatként. Az 5. állapot azzal végződik, hogy az F_3 folyamat megkapja az m_2 jelzést, rögzíti az állapotát és a C_{13}

csatorna állapotát üres sorozatként. A 12. állapot végén az F_4 folyamat megkapja az m_3 jelzést, rögzíti az állapotát, és a C_{24} csatorna állapotát üresnek. A 13. állapot végén az F_1 folyamat megkapja az m_6 jelzést az F_4 folyamattól a C_{41} csatornán keresztül. F_1 már korábban rögzítette a saját állapotát, azóta a C_{41} csatornán keresztül megkapta az e üzenetet, tehát a csatorna állapotát $\{e\}$-nek rögzíti. A 14. állapot azzal fejeződik be, hogy az F_2 folyamat megkapja az m_4 jelzést az F_3-tól a C_{32} csatornán keresztül. Mivel F_2 már korábban rögzítette a saját állapotát, és azóta fogadta az a üzenetet a C_{32} csatornán keresztül, most ennek a csatornának az állapotát $\{a\}$-nak rögzíti. A 15. állapot végén az F_3 folyamat megkapja az m_5 jelzést F_4-től a C_{43} csatornán keresztül. F_3 már rögzítette a saját állapotát, és azóta fogadta a d üzenetet a C_{43} csatornán keresztül. Most tehát ennek a csatornának az állapotát $\{d\}$-nek rögzíti. Ezzel befejeződött az algoritmus, hiszen minden csatornán keresztülment egy jelzés, így minden folyamat és minden csatorna állapota rögzítésre került. Az algoritmus által rögzített globális állapot a 9. ábrán látható.



9. ábra: Az algoritmus által rögzített globális állapot

Az algoritmus által rögzített állapot nem egyezik egyikkel sem, amelyben a rendszer volt az osztott számítás végrehajtása alatt. Mégis ez egy olyan állapot, ami előfordulhatott volna a végrehajtás alatt.

Legyen S' egy osztott számítás az algoritmus által rögzített globális állapota. Az osztott számítás eseményeit két csoportba sorolhatjuk a szerint, hogy a rögzítés előtt vagy utána történtek-e. Legyen $h=(a_1, a_2, ...)$ egy osztott számítás eseményeinek sorozata. Ez egyébként hasonlít a korábban bevezetett történelem fogalomhoz, azzal a különbséggel, hogy az halmaz, ez pedig sorozat, de az elemei ugyanazok. Ha két esemény konkurens, a h sorozatbeli helyüket úgy határozzuk meg, hogy a kisebbik indexű folyamat által végrehajtott kerül előbbre. A h sorozatban előfordulhat, hogy a_{i-1} rögzítés utáni esemény, az utána következő a_i pedig rögzítés előtti esemény. Meg lehet mutatni, hogy a két esemény nem befolyásolhatja egymást, tehát ha felcseréljük őket, akkor is számítást kapunk. Ha minden ilyen eseménypárt felcserélünk, akkor egy olyan számítást kapunk, amelyben minden rögzítés előtti esemény megelőz minden rögzítés utáni eseményt. Ebben a számításban a rögzítés előtti eseményeket követő és a rögzítés utáni eseményeket megelőző globális állapot megegyezik az algoritmus által rögzített globális állapottal. Ennek igazolásához azt kell belátnunk, hogy

(1) minden F folyamat állapota az S'-ben megegyezik az F folyamat állapotával a rá vonatkozó rögzítés előtti események végrehajtása után, és
(2) minden C csatorna állapota az S'-ben a következő: (a rögzítés előtt küldött üzenetek a C csatornán)-(a rögzítés előtti fogadott üzenetek a C csatornán).

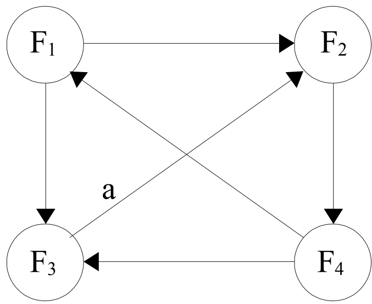
Az (1) állítás triviális. Most bebizonyítjuk a (2) állítást. Legyen C_{ij} az F_i folyamattól az F_j-hez vezető csatorna. A C_{ij} csatorna rögzített állapota az S'-ben azon üzenetek sorozata, amelyek az F_j folyamat kapott a csatornán keresztül az után, hogy rögzítette az állapotát és az előtt, hogy jelzést kapott volna ezen a csatornán. Az F_i által a jelzés küldése előtt a C_{ij} csatornán keresztül küldött üzenetek sorozata pontosan a rögzítés előtt küldött üzenetek sorozata. A (2) állítás innen már következik.

Vizsgáljuk meg az előző példában szereplő osztott számítás eseményeit. Az események sorrendje a következő:

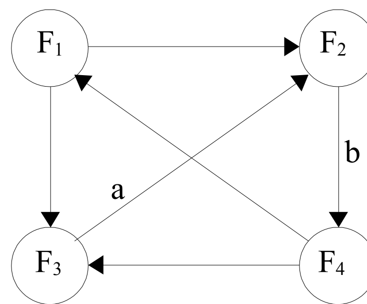
s(a) – rögzítés előtti
s(b) – rögzítés előtti
s(e) – rögzítés előtti
r(a) – rögzítés utáni
s(d) – rögzítés előtti
s(c) – rögzítés utáni
r(e) – rögzítés utáni
r(b) – rögzítés előtti
r(d) – rögzítés utáni
s(f) – rögzítés utáni
r(c) – rögzítés utáni
r(f) – rögzítés utáni.

Az egymást követő rögzítés utáni és rögzítés előtti esemény-párok cseréjével a következő eseménysorozatot kapjuk: *s(a)*, *s(b)*, *s(e)*, *s(d)*, *r(b)*, *r(a)*, *s(c)*, *r(e)*, *r(d)*, *s(f)*, *r(c)*, *r(f)*. Ebben a sorozatban a *b* üzenet fogadása (r*(b)*) az utolsó rögzítés előtti esemény, és az *a* üzenet fogadása (*r(a)*) az első rögzítés utáni esemény. A 10. ábrán látható, hogy a két esemény közötti állapot megegyezik az algoritmus által rögzített állapottal.

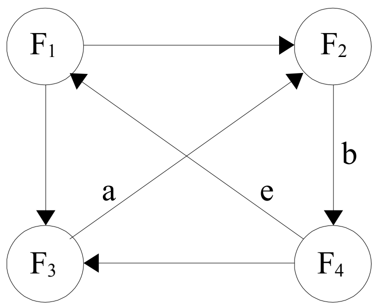
1.



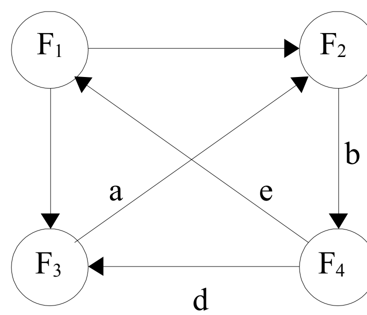
2.



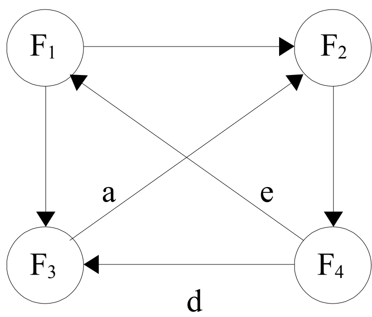
3.



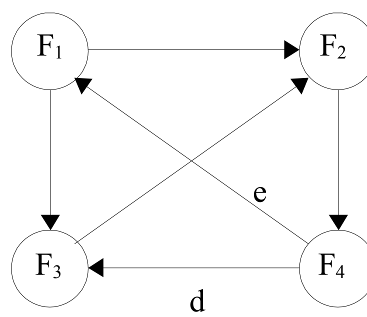
4.



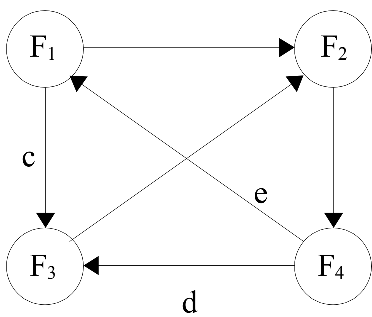
5.



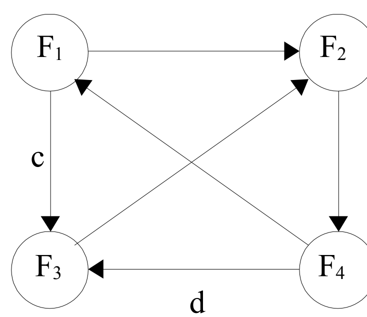
6.



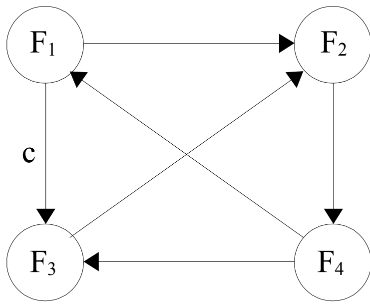
7.



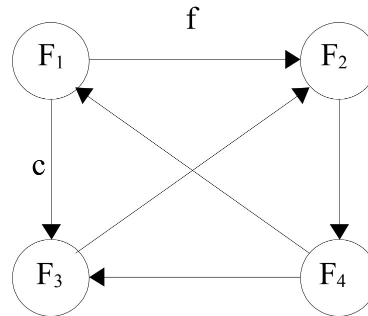
8.



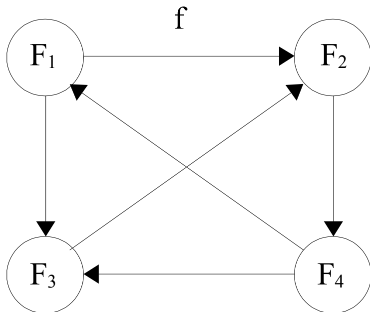
9.



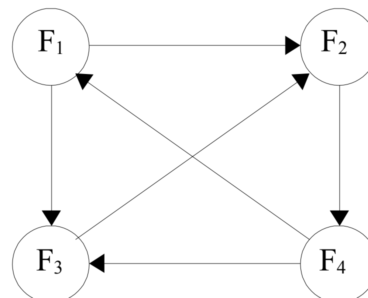
10.



11.



12.



10. ábra: Az események rendezése

Acharya-Badrinath algoritmus

Ez az algoritmus [1] teljesen más felfogást tükröz, mint a Chandy-Lamport algoritmus. Itt van egy koordinátor folyamat, amelyik elindítja az algoritmust, összegyűjti az információkat, és összerakja a globális állapotot.

Az algoritmus megköveteli, hogy mindegyik folyamatból mindegyik másikba vezessen irányított csatorna, valamint azt, hogy az üzeneteket okozati sorrendben kézbesítsék. Az utóbbi a következőt jelenti:

ha m_1 és m_2 két olyan üzenet, amit ugyanannak a folyamatnak küldenek (de nem feltétlenül ugyanaz a folyamat küldi), és $s(m_1) \rightarrow s(m_2)$, akkor
$r(m_1) \rightarrow r(m_2)$.

Az algoritmus megköveteli, hogy mindegyik F_i folyamat karban tartson két vektort, $sent_i$-t és $recd_i$-t. $sent_i[j]$ azon üzenetek száma, amelyeket az F_i folyamat küldött az F_j folyamatnak. $recd_i[j]$ azon üzenetek száma, amelyeket az F_i folyamat kapott az F_j folyamattól.

Most pedig következzen maga az algoritmus.

1. A kezdeményező F_{kezd} folyamat zsetonokat küld:

F_{kezd} küld egy zsetont minden folyamatnak, beleérve saját magát. F_{kezd} miután elküldi a zsetont, fogadja azt.

2. F_i folyamat fogadja a zsetont:

Amikor az F_i folyamat fogadja a zsetont, a következőket teszi

a) F_i rögzíti az LS_i lokális felvételt, ami

- a lokális állapotából, és a
- *sent*$_i$ és *recd*$_i$ vektorok értékéből áll.

b) Fi elküldi *reply*(LS_i)-t F_{kezd}-nek

3. F_{kezd} összeállítja a globális állapotot:

A kezdeményező folyamat, F_{kezd} megvárja míg mindegyik F_i folyamattól megkapja *reply*(LS_i)-t. A lokális állapotok F_{kezd} rendelkezésére állnak. F_{kezd} a *sent* és *recd* vektorok segítségével kiszámítja a csatornák állapotát. Az üzenetek *üzenet_azonosító* formájában lesznek ismertek. A globális állapot tehát a következőkből tevődik össze:

a) Minden F_i folyamat lokális állapota ismert F_{kezd} számára, mert azokat tartalmazzák a *reply* üzenetek.
b) $\forall$ i-re a $C_{kezd\ i}$ csatorna rögzített állapota üres sorozat.
c) $\forall$i $\forall$j, i$\neq$kezd-re:

C_{ij} állapota $\{(recd_j[i]+1),...,(sent_i[j]\}$.

Most megmutatjuk, hogy ez az algoritmus valóban konzisztens globális állapotot rögzít. Tegyük fel, hogy az *m* üzenetet az F_k folyamat küldi az F_l folyamatnak az után, hogy rögzítette a lokális állapotát, tehát $s(m)\notin S$, ahol S a rögzített globális állapot. Meg kell mutatni, hogy ekkor $r(m)\notin S$. Legyen az F_{kezd} folyamat által az F_k-nak küldött zseton *zseton*$_k$, az F_l-nek küldött zseton *zseton*$_l$. Ekkor

$s(zseton)\rightarrow r(zseton_k)$.

Mivel egy folyamat akkor rögzíti az állapotát, amikor megkapja a zsetont, és *m*-et F_k az után küldte, hogy rögzítette a lokális állapotát, ezért

$r(zseton_k)\rightarrow s(m)$.

Mivel a $\rightarrow$ reláció tranzitív, ezért

$s(zseton)\rightarrow s(m)$.

Tehát a zseton küldése okozatilag megelőzi az *m* üzenet küldését. Az üzenetek okozati sorrendben való kézbesítése biztosítja, hogy bármelyik folyamat, amelyik megkapja a zsetont is és az *m* üzenetet is, a zsetont fogja előbb megkapni. Tehát F_l-re teljesül, hogy

$r(zseton_l)\rightarrow r(m)$.

F_l akkor rögzíti a lokális állapotát, amikor megkapja $zseton_l$-t. Mivel az *m* üzenet ez után érkezik, ezért

$r(m) \notin S$.

Ezzel beláttuk, hogy az Acharya-Badrinath algoritmus konzisztens globális állapotot rögzít.

Most megmutatjuk, hogy a csatornaállapotokat helyesen rögzíti az algoritmus, vagyis ha az F_k folyamat az *m* üzenetet az után küldte F_l-nek, hogy az állapotát rögzítette, az *m* üzenet nem eleme a C_{kl} csatorna rögzített állapotának. A C_{kl} csatorna az algoritmus által rögzített állapota: $\{recd_l[k]+1,...,sent_k[l]\}$. Mivel az *m* üzenetet F_k az után küldte, hogy rögzítette a lokális állapotát, ezért az *m* üzenet azonosítója nagyobb, mint $sent_k[l]$, tehát *m* nincs a csatornaállapotként rögzített üzenetek közt.

Még meg kell mutatnunk, hogy az algoritmus helyesen jár el, amikor minden az F_{kezd} folyamatból kiinduló csatorna állapotát üresnek veszi. Legyen *m* egy olyan üzenet, amelyet F_{kezd} F_j-nek küld az előtt, hogy zsetont küldene. Mivel $s(m) \rightarrow s(zseton)$, az üzenetek okozati sorrendben való kézbesítése miatt *m* előbb érkezik meg F_j-hez, mint a zseton. Tehát bármely $C_{kezd\ j}$ F_{kezd}-ből kiinduló csatorna állapota üres a rögzítéskor.

A Chandy-Lamport és az Acharya-Badrinath algoritmus összehasonlítása

Bemutattunk két algoritmust, amelyek osztott felvételt készítenek. A Chandy-Lamport algoritmus osztott rendszerek egy bővebb halmazára alkalmazható, mint az Acharya-Badrinath algoritmus. Egyrészt, mert az üzenetek okozati sorrendben való kézbesítéséből következik a csatornák FIFO tulajdonsága, de ez fordítva nem igaz. Másrészt, mert azon irányított gráfok halmaza, amelyekben bármely két csúcs között van irányított él részhalmaza az erősen összefüggő gráfok halmazának.

Az Acharya-Badrinath algoritmus úgy indul, hogy a kezdeményező folyamat küld egy zsetont minden folyamatnak. A Chandy-Lamport algoritmus úgy kezdődik, hogy egy vagy több folyamat rögzíti a lokális állapotát, és küld egy jelzést minden kimenő csatornáján keresztül. Mindkét algoritmusnál van tehát kezdeményező folyamat, de míg az Acharya-Badrinath algoritmus esetében ez a folyamat végig kitüntetett marad, a Chandy-Lamport algoritmusnál nincs több szerepe, a továbbiakban azt teszi, amit a többiek.

n db folyamat esetén az Acharya-Badrinath algoritmus n db üzenetet (zseton) igényel ahhoz, hogy mindegyik folyamat rögzítse a lokális állapotát. Ugyanehhez a Chandy-Lamport algoritmus esetén a csatornák számával egyező számú üzenetre (jelzés) van szükség. Ha mindegyik folyamatból mindegyik másikba vezet irányított él, akkor a csatornák száma $n*(n-1)$. Ezt az Acharya-Badrinath algoritmus esetén megköveteljük, a Chandy-Lamport algoritmusnál azonban nem. Minimum n db él kell

ahhoz, hogy minden folyamatból minden másikba vezessen irányított út, ami a Chandy-Lamport algoritmus alkalmazhatóságának feltétele. Viszont a két algoritmus működését úgy van értelme összehasonlítani, hogy azonos tulajdonságú rendszerek esetén vizsgáljuk. Azoknál a rendszereknél tehát, amelyekre mindkét algoritmus alkalmazható az igaz, hogy az Acharya-Badrinath algoritmusnak n db üzenetet kell küldenie a lokális állapotok rögzítéséhez, a Chandy-Lamportnak pedig n*(n-1) db-ot.

A Chandy-Lamport algoritmusnál egy folyamat akkor rögzíti a lokális állapotát, amikor az első jelzést megkapja, és még nem rögzítette. Az Acharya-Badrinath algoritmus esetén egy folyamat akkor rögzíti a lokális állapotát, amikor megkapja a zsetont. A lokális állapot rögzítése után a Chandy-Lamport algoritmusnál a folyamat küld egy jelzést minden kimenő csatornáján, mielőtt további üzeneteket küldene, a Acharya-Badrinath algoritmus esetén a folyamat elküldi a lokális állapotát a kezdeményező folyamatnak.

A Chandy-Lamport algoritmus esetén egy csatorna állapotát az a folyamat rögzíti, amelyik a csatorna kimenő végén van. Az Acharya-Badrinath algoritmusnál minden csatorna állapotát a kezdeményező folyamat számítja ki a folyamatoktól kapott vektorok segítségével. A Chandy-Lamport algoritmus tehát dinamikusan rögzíti a csatornák állapotát, míg az Acharya-Badrinath kiszámítja. A Chandy-Lamport algoritmus által rögzített csatornaállapot üzenetek egy sorozatából áll, az Acharya-Badrinath algoritmus által rögzített pedig üzenetazonosítók egy sorozatából, a kezdeményező folyamat nem ismeri az üzenetek tartalmát.

Érdekes különbség a két algoritmus között, hogy a Chandy-Lamport algoritmus akkor ér véget, amikor az összes a globális állapotban úton lévő üzenet megérkezik, az Acharya-Badrinath esetén viszont ezt nem kell megvárni.

A Chandy-Lamport algoritmus csak a folyamatok lokális állapotainak és a csatornák állapotainak meghatározására szolgál, nem része a globális állapot összeállítása. Az Acharya-Badrinath algoritmus teljes, vagyis úgy ér véget, hogy a kezdeményező folyamat kiszámította a globális állapotot. A lokális állapotok összegyűjtéséhez n db üzenetre van szükség.

Az Acharya-Badrinath algoritmusnak hátránya a Chandy-Lamport algoritmussal szemben, hogy osztott rendszerek szűkebb csoportjára alkalmazható. Viszont, ha olyan tulajdonságú az osztott rendszer, amelyre alkalmazható, akkor ha minél kevesebb üzenet küldésével akarjuk megoldani a globális állapot rögzítését, akkor célszerűbb azt alkalmazni, mert csak n darab üzenetre van szükség, míg a Chandy-Lamport algoritmus esetében n*(n-1) db üzenetre. Az Acharya-Badrinath algoritmusnak tehát az az előnye, hogy kevesebb üzenet küldésével rögzíti a globális állapotot.

A Chandy-Lamport algoritmusnak nagy előnye a Acharya-Badrinath algoritmussal szemben, hogy sok olyan esetben alkalmazható, amelyben az utóbbi nem. Vannak tehát olyan esetek, amikor választhatunk, hogy melyik algoritmust használjuk, de van olyan is, amikor a Chandy-Lamport algoritmus használható, az Acharya-Badrinath algoritmus pedig nem.

Mint láttuk, mindkét algoritmusnak van előnye is és hátránya is a másik algoritmussal szemben.

Hullám sorozat FIFO modellre

Egy olyan módszert [7] fogunk bemutatni, amelynek segítségével egy osztott rendszer globális állapotainak egy sorozatát lehet rögzíteni. Ennek a módszernek két változata van: egy olyan rendszerekre, amelyek FIFO tulajdonságú csatornákat használnak, és egy másik olyanokra, amelyeknél a csatornák nem FIFO tulajdonságúak. Most az elsővel foglalkozunk.

A módszert két lépésben ismertetjük. Az első lépésben a folyamatok lokális állapotára koncentrálunk. Egy folyamatnak a lokális állapota rögzítése során össze kell dolgoznia a szomszédos folyamatokkal annak érdekében, hogy lokális állapotaik konzisztensek legyenek, vagyis ne rögzítse az egyik folyamat olyan üzenet fogadását, amelynek elküldését a másik folyamat nem rögzíti. A második lépésben megmutatjuk, hogy a folyamatok hogyan tudják rögzíteni lokális állapotok egy sorozatát úgy, hogy a sorozatok egyező indexű tagjai konzisztens globális állapotot alkossanak.

Tartozzon minden F_i folyamathoz egy CTL_i kontroll folyamat, amelynek a következő feladatai vannak:

(1)figyeli az F_i által fogadott és elküldött üzeneteket úgy, hogy ne zavarja az osztott számítás végrehajtását,
(2)kezeli a kontroll üzeneteket, amiket F_i nem lát, és
(3)rögzíti F_i lokális állapotát.

Nézzük meg, hogy mi kell ahhoz, hogy az F_i és F_j folyamatok - amelyeket a C_{ij} csatorna köt össze - úgy rögzítsék a lokális állapotukat, hogy ne legyen olyan üzenet, amit F_j megkapott és F_i nem küldött el. Legyenek LS_i és LS_j a rögzített lokális állapotok. Tegyük fel, hogy LS_i rögzítésének ideje t_i, LS_j rögzítésének ideje pedig t_j. Ezek az időpontok globális logikai időpontok. Legyen *a* olyan üzenet, amelyet Fi küld F_j-nek a C_{ij} csatornán keresztül az után, hogy CTL_i rögzítette a lokális állapotát, vagyis $t_i<t(s(a))$, ahol *t* a logikai időt reprezentáló függvény. A rögzített lokális állapotok akkor konzisztensek, ha ebben az esetben F_j az után kapja meg az *a* üzenetet, hogy CTL_j rögzíti a lokális állapotát, vagyis ha $t_j<t(r(a))$. Legyen m_{ij} egy kontroll üzenet, amelyet CTL_i küld CTL_j-nek C_{ij}-n keresztül t_i-kor. Ekkor $t(s(m_{ij}))=t_i<t(s(a))$. A C_{ij} csatorna FIFO tulajdonsága miatt így $t(r(m_{ij}))<t(r(a))$, tehát ha $t_j\leq t(r(m_{ij}))$ telesül, akkor a rögzített lokális állapotok konzisztensek lesznek. Összefoglalva: ha F_i-ből F_j-be vezet egy C_{ij} csatorna és CTL_i a lokális állapot rögzítésekor küld egy kontroll üzenetet CTL_j-nek, és CTL_j legkésőbb ezen üzenet fogadásakor rögzíti a lokális állapotot, akkor a két rögzített állapot konzisztens lesz.

Hullámnak nevezünk egy kontroll folyamot, ami minden folyamatot a rendszerben pontosan egyszer érint, és legalább egy folyamattal tudatja, hogy mikor ér véget. Egy hullámhoz a következő események tartoznak:

– $\forall$ F_i *visit(i)*, azaz az F_i folyamatot érinti a hullám,
– *return*, amikor $\forall$ F_i-re *visit(i)* megtörtént.

Egy hullám adatokat közvetít és adatokat gyűjt. Minden folyamatot érintve közli velük a *diff* adatot, és begyűjti tőlük a $coll_i$ adatokat, amelynek segítségével $coll=f(coll_1,...,coll_n)$ meghatározható. Mielőtt egy hullám elindul, minden F_i folyamat tudja $coll_i$-t, és van olyan folyamat, amelyik ismeri *diff*-et. A hullám minden folyamatot érint (*visit(i)*) majd visszatér (*return*). Ez után minden folyamat ismerni fogja *diff*-et, és lesz olyan folyamat, amelyik ismeri $f(coll_1, ..., coll_n)$-t.

Az a feladatunk, hogy minden folyamat lokális állapotok egy olyan $(LS_i^\mu)_{\mu\in N}$ sorozatát rögzítse, hogy $\forall$ μ-re az $S(\mu)=\{LS_1^\mu,..., LS_n^\mu\}$ globális állapot konzisztens legyen. Ehhez hullám sorozatokat fogunk használni.

A hullámok sorozata olyan, hogy

$\forall\mu$: $(\forall i\ visit(i, \mu))\rightarrow return(\mu)\rightarrow(\forall i\ visit(i, \mu+1))$, ahol
$visit(i, \mu)$ a μ-edik *visit(i)*, $return(\mu)$ pedig a μ-edik return.

A $return(\mu)$ tehát elválasztja a $\{visit(i, \mu) \mid$ Fi egy folyamat a rendszerben$\}$ és a $\{visit(i, \mu+1) \mid$ Fi egy folyamat a rendszerben$\}$ halmazokat. Így létezik egy olyan globális állapot, amelyben az első halmaz eseményei szerepelnek, de a második halmaz eseményei nem.

Egy hullám végigmenve az összes folyamaton rögzíti a globális állapotot, S (μ)-t. A hullámok sorozata összehangolja a lokális állapotok rögzítését. Ahhoz, hogy egy hullám sorozat konzisztens globális állapotok sorozatát rögzítse az alábbi szabályokat kell alkalmazni.

(SZ1.) CTL_i rögzíti az LS_i^μ lokális állapotot, amikor $visit(i, \mu)$ esemény megtörténik, vagyis amikor a μ-edik hullám érinti az F_i folyamatot.
(SZ2.) Amikor CTL_i rögzíti LS_i^μ-t, küld egy $m_{ij}(\mu)$ jelzést minden belőle kiinduló C_{ij} csatornán keresztül.
(SZ3.) (fagyasztó szabály) Ha CTL_j az előtt kapja meg az $m_{ij}(\mu)$ jelzést a C_{ij} csatornán keresztül, hogy $visit(j, \mu)$ megtörtént volna, akkor nem engedi, hogy F_j üzenetet fogadjon addig, amíg LS_j^μ lokális állapotot nem rögzíti.

A három szabály biztosítja, hogy $\forall$ μ esetén a rögzített S(μ) globális állapot konzisztens. Korábban beláttuk, hogy ha F_i-ből F_j-be vezet egy C_{ij} csatorna és CTL_i a lokális állapot rögzítésekor küld egy kontroll üzenetet CTL_j-nek, és CTL_j legkésőbb ezen üzenet fogadásakor rögzíti a lokális állapotot, akkor a két rögzített állapot konzisztens lesz. Mivel az F_j folyamat az (SZ3.) miatt nem fogad üzenetet az $m_{ij}(\mu)$ jelzés fogadása és a lokális állapota rögzítése között, ezért az üzenetek szempontjából olyan, mintha CTL_j az $m_{ij}(\mu)$ fogadásával egy időben rögzítené az LS_j^μ lokális állapotot. Így tehát biztosított a konzisztencia.

A módszer segítségével rögzítettük globális állapotok egy sorozatát, de ez a globális állapot nem terjed ki a csatornák állapotára. A hullám sorozatok lehetőséget nyújtanak arra is, hogy az úton lévő üzenetek halmazait, vagyis a csatornák állapotát rögzítsük. A $coll_i$ változók segítségével gyűjthetünk adatokat a folyamatoktól, a *diff*

változó segítségével pedig közvetíthetjük az összegyűjtött adatokat a folyamatok felé.

A FIFO modellre alkalmazott hullám sorozat módszer alapgondolata hasonló a Chandy-Lamport algoritmuséhoz. Mindkét algoritmus jelzések küldésével szinkronizálja a folyamatok lokális állapotának rögzítését. A Chandy-Lamport algoritmusnál a kezdeményező folyamat(ok) kivételével a folyamatok akkor rögzítik a lokális állapotukat, amikor az első jelzést megkapják. A hullám sorozat módszernél a folyamatok akkor rögzítik az állapotukat, amikor a hullám érinti őket, de ha az előtt kapnak jelzést, akkor a jelzés fogadásától az állapotuk rögzítéséig nem fogadnak üzeneteket, tehát konzisztencia szempontjából olyan mintha akkor rögzítenék az állapotukat, amikor a jelzést megkapják.

Mindkét algoritmusnál, miután egy folyamat rögzíti a lokális állapotát, rögtön küld egy jelzést minden kimenő csatornáján keresztül. Az elküldött jelzések száma egy hullámnál ugyanannyi, mint a Chandy-Lamport algoritmusnál.

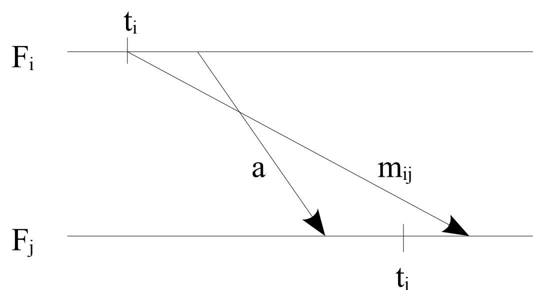
A hullám módszer csatornaállapot rögzítése az Acharya-Badrinath algoritmuséhoz hasonlít. Az utóbbinál láttuk, hogy minden folyamat karban tart két vektort: az egyikben az egyes folyamatoknak küldött, a másikban a tőlük kapott üzenetek számát tárolja. A hullám sorozatok esetén is hasonlóan történik az úton lévő üzenetek számának, vagy maguknak az üzeneteknek a meghatározása. Ott is két vektort tart karban egy folyamat, az egyikben az előző hullám óta az egyes folyamatoknak elküldött üzenetek, a másikban a tőlük kapottak száma, vagy maguk az üzenetek. A hullám sorozatok esetében a hullám összesíti az adatokat úgy, hogy minden folyamat érintése alkalmával megkapja a folyamattól a szükséges infomációt és annak alapján állítja be egy általa kezelt változó értékét. Ez a változó lehet az üzenetek száma vagy halmaza. Az Acharya-Badrinath algoritmus esetén a kezdeményező folyamat gyűjti az adatokat és határozza meg az úton lévő üzeneteket.

A hullám sorozat módszer konzisztens globális állapotok egy sorozatát rögzíti, míg a Chandy-Lamport algoritmus és az Acharya-Badrinath algoritmus egyetlen konzisztens globális állapotot.

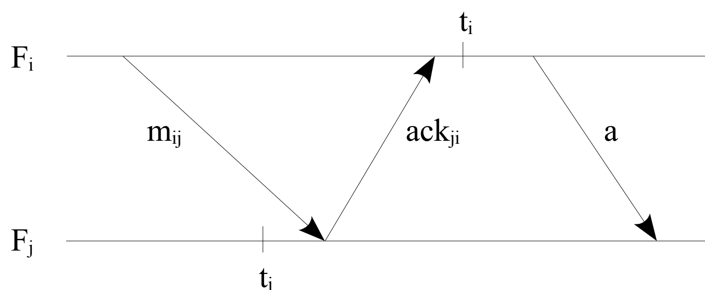
Hullám sorozat nem FIFO modellre

Ez a módszer [7] az előzőnek egy olyan átalakítása, amit használni lehet abban az esetben is, amikor az osztott rendszer csatornái nem FIFO tulajdonságúak.

A FIFO modellre alkalmazható módszernél az első lépésben erősen kihasználtuk a csatornák FIFO tulajdonságát. Ha a C_{ij} csatorna nem FIFO tulajdonságú, akkor előfordulhat az az eset, hogy CTL_i rögzíti a lokális állapotot és elküldi az m_{ij} jelzést F_j-nek, majd F_i elküldi az *a* üzenetet F_j-nek, és az *a* üzenet előbb érkezik meg F_j-hez, mint az m_{ij} jelzés. Ezt a helyzetet a 11. ábra szemlélteti. Ebben az esetben nem elég, ha a CTL_j legkésőbb az m_{ij} jelzés fogadásakor rögzíti F_j lokális állapotát, hiszen ha pont a jelzés fogadásakor rögzíti a lokális állapotokat, akkor olyan LS_i és LS_j lokális állapotokat kapunk, melyek szerint F_i még nem küldte el az *a* üzenetet, F_j viszont már megkapta. Ennek elkerülése végett bevezetjük a nyugta fogalmát, amit egy folyamat akkor küld a másiknak, ha jelzést kapott tőle. Amikor



11. ábra: Nem FIFO csatorna



12. ábra: Nem FIFO csatorna nyugtával

CTL_i rögzíteni akarja az F_i folyamat lokális állapotát, küld egy m_{ij} jelzést a C_{ij} csatornán keresztül. CTL_j legkésőbb a jelzés fogadásakor rögzíti F_j lokális állapotát, és pontosan a fogadáskor küld egy ack_{ji} nyugtát CTL_i-nek. CTL_i akkor rögzítheti F_i lokális állapotát, ha megkapta az ack_{ji} nyugtát. Így a rögzített lokális állapotok konzisztensek lesznek. Legyen t_i az a logikai időpont, amikor CTL_i rögzíti F_i lokális állapotát, t_j pedig amikor CTL_j rögzíti F_j állapotát. Legyen a olyan üzenet, amit F_i a lokális állapotának rögzítése után küld F_j-nek, azaz $t(s(a))>t_i$. Megmutatjuk, hogy ekkor $t(r(a))>t_j$ is teljesül. $t(r(a))>t(s(a))$, mivel egy üzenetet csak később lehet megkapni, mint ahogy elküldték. $t(s(a))>t_i$, mert a-t ilyennek határoztuk meg. $t_i \geq t(r(ack_{ji}))$, mert azt mondtuk, hogy CTL_i az után rögzíti a lokális állapotot, hogy megkapta a nyugtát. $t(r(ack_{ji}))>t(s(ack_{ji}))$, nyilvánvalóan. $t(s(ack_{ji}))=t(r(m_{ij}))$ a nyugta definíciója miatt. $t(r(m_{ij}))\geq t_j$, mert így határoztuk meg t_j-t. Tehát $t(r(a))>t_j$, vagyis beláttuk, hogy az az üzenet, amit F_i a lokális állapota rögzítése után küld F_j-nek, azt F_j a saját lokális állapotának rögzítése után kapja meg.

Ahhoz, hogy a lokális állapotokat összegyűjtsük úgy, hogy konzisztens globális állapotot adjanak, itt is hullámsorozatokat használunk. A követendő szabályokat úgy alakítjuk, hogy alkalmasak legyenek nem FIFO csatornák kezelésére.

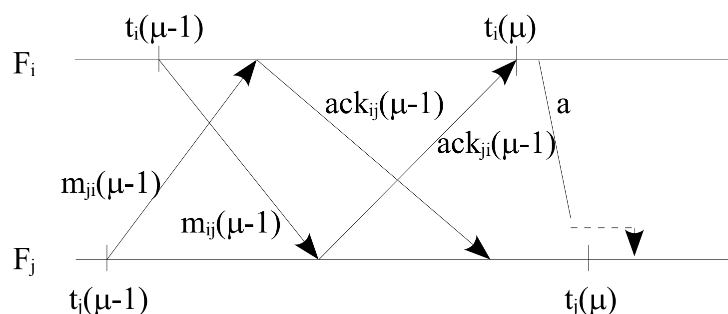
(SZ'1.) CTL_i rögzíti az LS_i^{μ} lokális állapotot, amikor $visit(i, \mu)$ esemény megtörténik, vagyis amikor a μ-edik hullám érinti az F_i folyamatot.
(SZ'2.) CTL_i csak akkor rögzítheti az LS_i^{μ} lokális állapotot, amikor már mindegyik bejövő C_{ji} csatornán megkapta az $m_{ij}(\mu\text{-}1)$ jelzéshez tartozó $ack_{ji}(\mu\text{-}1)$ nyugtát.
(SZ'3.) Amikor CTL_i rögzíti LS_i^{μ}-t, küld egy $m_{ij}(\mu)$ jelzést minden belőle kiinduló C_{ij}

csatornán keresztül.
(SZ'4.) (fagyasztó szabály) Miután CTL_i megkapta az $m_{ji}(\mu\text{-}1)$ jelzést a C_{ji} csatornán keresztül, nem engedi, hogy F_i üzeneteket vagy jelzéseket fogadjon a C_{ji} csatornán keresztül addig amíg nem rögzíti az LS_i^{μ} lokális állapotot.

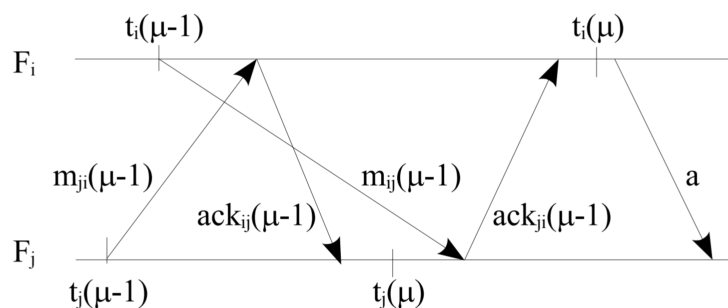
Az (SZ'1.) szabály megegyezik az (SZ1.) szabállyal, az (SZ'3.) pedig az (SZ2.)-vel.

Megmutatjuk, hogy az (SZ'1.)-(SZ'4.) szabályok biztosítják, hogy $\forall\mu$-re a rögzített $S(\mu)$ lokális állapot konzisztens. Legyen a egy üzenet, amelyet az F_i folyamat az után küld az F_j folyamatnak, hogy CTL_i rögzítette az LS_i^{μ} lokális állapotot, vagyis $t(s(a))>t_i(\mu)$. Be kell látnunk, hogy ekkor $t(r(a))>t_j(\mu)$. (SZ'2.) miatt a következő egyenlőtlenségek teljesülnek:
$t(r(a))>t(s(a))>t_i(\mu)\geq t(r(ack_{ji}(\mu\text{-}1)))>t(r(m_{ij}(\mu\text{-}1)))$, tehát az F_j folyamatnál $t(r(a)) >t(r(m_{ij}(\mu\text{-}1)))$.
Két esetet különböztethetünk meg a C_{ij} csatornán. Ezeket az eseteket a 13. és a 14. ábra szemlélteti.

I. Az $m_{ij}(\mu\text{-}1)$ jelzést CTL_j előbb kapja meg, mint hogy rögzíti az LS_j^{μ} lokális állapotot. Ekkor (SZ'4.) miatt az F_j folyamat nem fogad üzeneteket és jelzéseket az $[t(r(m_{ij}(\mu\text{-}1))), t_j(\mu)]$ időintervallumban, tehát $t(r(a))>t_j(\mu)$.
II. CTL_j az után kapja meg az $m_{ij}(\mu\text{-}1)$ jelzést, hogy rögzítette az LS_j^{μ} lokális állapotot. Ekkor mivel $t(r(m_{ij}(\mu\text{-}1)))>t_j(\mu)$, ezért $t(r(a))>t_j(\mu)$.



13. ábra: $m_{ij}(\mu\text{-}1)$ érkezése megelőzi $t_j(\mu)$-t



14. ábra: $t_j(\mu)$ megelőzi $m_{ij}(\mu\text{-}1)$ érkezését

A nem FIFO hullám sorozat módszer a FIFO hullám sorozat módszer egy olyan továbbfejlesztése, ami lehetővé teszi a nem FIFO tulajdonságú csatornák kezelését. Első lépésként itt is azt vizsgáltuk meg, hogy hogyan lehet két szomszédos folyamat állapotát konzisztensen rögzíteni. Azt láttuk, hogy nem elég a jelzések használata, szükség van nyugtákra is. Ez tulajdonképpen a Chandy-Lamport algoritmus alapgondolatának nem FIFO csatornákra való alkalmazása.

A nem FIFO hullám sorozat esetén egy folyamat akkor rögzíti az állapotát, amikor a hullám érinti, ugyanúgy, mint a FIFO hullám sorozatnál. A nem FIFO hullámsorozat módszer szerint egy folyamat nem rögzítheti az állapotát addig, amíg meg nem kapta minden bejövő csatornáján a nyugtát, ami az általa küldött jelzés fogadását nyugtázza. Ilyen szabály a FIFO esetben nincs, hiszen ott nincs is szükség nyugtákra. Mindkét esetben amikor egy folyamat rögzíti az állapotát, küld egy jelzést minden kimenő csatornáján keresztül. A nem FIFO esetben egy folyamat állapotának rögzítése előtt az előző hullámhoz tartozó nyugtákat kell megvárni. Mind a FIFO mind a nem FIFO hullám sorozat esetén egy hullámhoz a csatornák számával megegyező számú jelzés tartozik, és a nem FIFO esetben ugyanennyi nyugta.

A folyamatok színezése

Most egy olyan globális állapotot rögzítő módszert fogunk bemutatni, ami a folyamatok és az események színezésén alapul. Egy folyamat attól függően kap valamilyen színt, hogy rögzítette-e már a lokális állapotát. Egy esemény színe pedig attól függ, hogy milyen színű folyamat hajtja végre. Először a Chandy-Lamport algoritmus színezéses értelmezését mutatjuk be [6], utána pedig egy színezéses módszert a nem FIFO tulajdonságú csatornákkal rendelkező osztott rendszerek globális állapotának rögzítésére.

Tehát először tekintsünk egy osztott rendszert, amelyben a csatornák FIFO tulajdonságúak. Mindegyik folyamat, esemény és üzenet fehér vagy piros. Egy esemény olyan színű, mint a folyamat, ami végrehajtja. Egy üzenet olyan színű, mint az esemény, ami őt küldi. A globális felvételt készítő algoritmus kezdetekor minden folyamat fehér színű. Az algoritmus során minden folyamat pontosan egyszer pirossá válik. Egy folyamat történelme így események egy olyan sorozatából áll, amelynek első valahány tagja mind fehér, a többi pedig mind piros. Amikor az algoritmus befejeződik minden folyamat piros, és a csatornákon nincsenek fehér üzenetek. A rögzített globális állapot a következőkből fog állni:

(1) Minden folyamatnak az az állapota, amelyben akkor volt, amikor a színe fehérről pirosra változott.
(2) Minden csatornán azon fehér üzenetek sorozata, amelyek fogadása piros esemény. Ezek pontosan azok az üzenetek, amelyeket egy folyamat az előtt küldött, hogy rögzítette volna a lokális állapotát, egy másik pedig az után kapta meg, hogy rögzítette a lokális állapotát.

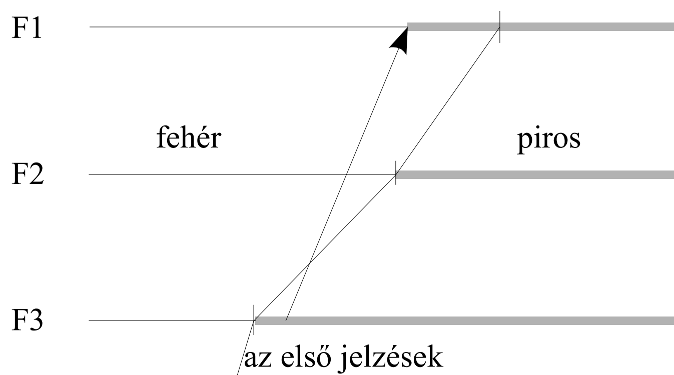
A rögzített globális állapotban nem szerepelhetnek olyan piros színű üzenetek, amelyeket fehér folyamat fogad, hiszen ezeknek az elküldése nem lenne rögzítve, a fogadásuk viszont igen. Mivel ezeket a színezéseket a Chandy-Lamport algoritmusnak megfelelően csináljuk, nem is lesznek ilyen üzenetek rögzítve. Amikor egy folyamat rögzíti a lokális állapotát, abban a pillanatban a színe fehérről pirosra változik, és küld egy jelzést minden kimenő csatornáján keresztül. Amikor egy folyamat megkapja az első jelzést és még fehér, pirosra változik. Mivel minden csatornán keresztülmegy egy jelzés, minden folyamat pirosra fog változni.

Most nézzük azt a helyzetet, amikor a csatornák nem FIFO tulajdonságúak [10]. Ahhoz, hogy a rögzített globális állapot konzisztens legyen, a következő sémát kell követnünk.

(1)Kiinduláskor minden folyamat fehér, és akkor válik pirossá, amikor rögzíti a lokális állapotát.
(2)A fehér folyamatok által küldött üzenetek fehérek.
(3)A piros folyamatok által küldött üzenetek pirosak.
(4)Minden folyamat tetszőleges időpontban rögzíti a lokális állapotát, de az előtt, hogy piros üzenetet kaphatna.

Ezeket betartva, a rögzített lokális állapotok nyilvánvalóan konzisztens globális állapotot fognak alkotni. Ahhoz, hogy az algoritmus befejeződjön, biztosítani kell, hogy minden folyamat rögzítse a lokális állapotát. A rögzített lokális állapotokat a folyamatok elküldik egy kiválasztott folyamatnak.

Az algoritmus úgy indul, hogy egy folyamat spontán rögzíti a globális állapotát, és pirossá változik. Majd küld egy jelzést – akár közvetve, akár közvetlenül – minden folyamatnak, hogy biztosítsa, hogy előbb-utóbb minden folyamat pirossá váljon.



15. ábra: Fehér folyamat piros üzenetet kap

Előfordulhat, hogy egy folyamat előbb kap egy piros üzenetet, mint ahogy a jelzést megkapná. Ezt a helyzetet mutatja a 15. ábra. Ilyenkor rögzítenie kell az állapotát mielőtt még fogadná az üzenetet. Ha nem tudja előre az üzenet színét,

fogadja az üzenetet, de rögzítenie kell az állapotát mielőtt még megváltoztatná az üzenet hatására.

A globális állapotban a csatornákon lévő üzenetek pontosan azok, amelyeket fehér folyamatok küldtek és piros folyamatok fogadtak. Egy lehetséges megoldás a csatornák állapotának rögzítésére az, hogy amikor egy piros folyamat fehér üzenetet kap, annak másolatát elküldi a kezdeményezőnek. Amikor a kezdeményező folyamat megkapta az összes folyamat lokális állapotát és az útban lévő üzeneteket, birtokában van a teljes globális állapotnak. Ezzel a módszerrel az a baj, hogy a kezdeményező folyamat nem tudja, hogy meddig várjon még úton lévő üzenetekre, illetve nem tudja eldönteni, hogy megkapta-e az utolsót. Lehetséges megoldás, hogy minden folyamathoz tartozzon egy számláló, ami számolja a folyamat által elküldött és fogadott üzenetek különbségét. Ha a számlálók értékét a folyamatok elküldik a kezdeményező folyamatnak, az tudni fogja, hogy hány útban lévő üzenet másolatát kell megkapnia, tehát meg lehet határozni az algoritmus befejeződését.

Ha a folyamatok valamiért nem tudják elküldeni a kapott üzenetek másolatát a kezdeményező folyamatnak, más megoldást kell választani az algoritmus befejeződésének meghatározására. Ehhez bevezetjük a vektor számlálók használatát.

Minden F_i folyamat karban tart egy v_i vektort. A v_i vektor j-edik komponense ($i \neq j$) az F_i folyamat által az F_j folyamatnak küldött fehér üzenetek száma. Minden alkalommal, amikor a még fehér F_i folyamat küld egy üzenetet az F_j folyamatnak, növeli $v_i[j]$ értékét eggyel. A v_i vektor i-edik komponense az F_i folyamat által kapott fehér üzenetek számának az ellentettje. Minden alkalommal, amikor az F_i folyamat kap egy fehér üzenetet, csökkenti $v_i[i]$ értékét eggyel.

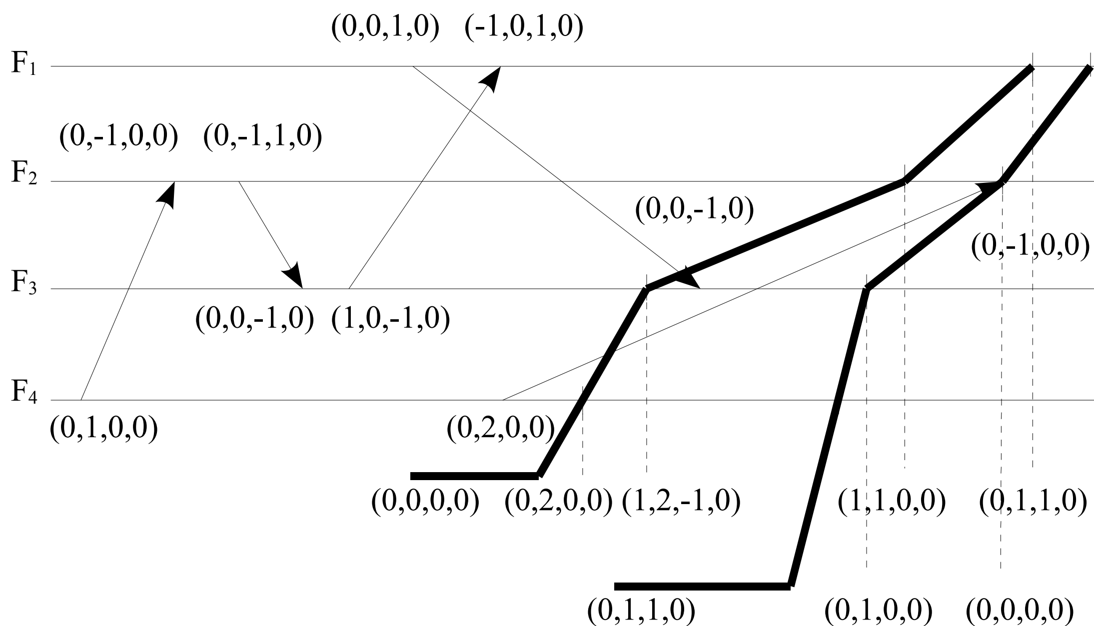
Van egy kontroll üzenet, ami eljut mindegyik folyamathoz. A kontroll üzenet hordoz magával egy c vektort. Amikor a kontroll üzenet az F_i folyamathoz ér, a következőt teszi:

- hozzáadja c-hez v_i-t, $c:=c+v_i$,
- lenullázza v_i-t, $v_i:=0$.

Amikor a kontroll üzenet először érkezik egy folyamathoz, és a folyamat még fehér, pirosra színezi. A kontroll üzenet egyszer vagy kétszer megy körbe a folyamatokon. Az első körben összegyűjti a lokális állapotokat. A kör után a c vektor i-edik komponense az F_i folyamat felé útban lévő fehér üzenetek száma. Második körre, akkor van szükség, ha az első kör után van olyan i, amelyre $c[i]>0$. A második körben a kontroll üzenet minden folyamatnál megvárja, amíg az összes felé útban lévő fehér üzenet megérkezik. Ennek bekövetkeztét az jelzi, hogy $v_i[i]+c[i]=0$ lesz. $v_i[i]$ nulláról indul az első kör után, és minden alkalommal, amikor F_i fehér üzenetet fogad, eggyel kisebb lesz. $c[i]$ pedig az F_i-nek elküldött fehér üzenetek száma. A kettő összege tehát valóban akkor lesz 0, amikor F_i-hez az összes fehér üzenet megérkezik. A második körben a csatornák állapota meghatározható.

A vektor számlálók használatát szemlélteti a 16. ábra. Az ábrán négy folyamat van, mindegyik karban tartja a v_i vektort. A kontroll üzenet első köre után a c vektor értéke (0,1,1,0), tehát két olyan üzenet van, amelyet már elküldtek, de még nem

érkezett meg. Az egyiket az F_2 folyamatnak, a másikat az F_3-nak. A kontroll üzenet elindul a második körre és mindkét folyamatnál bevárja az úton lévő üzeneteket, hogy a *c* vektor értéke végül (0,0,0,0) legyen.



16. ábra: Vektor számlálók használata

A színezéses algoritmus nem követeli meg, hogy a csatornák FIFO tulajdonságúak legyenek, mint ahogy a nem FIFO hullám sorozat módszer sem. A színezéses algoritmusnak előnye a hullám sorozattal szemben, hogy nem kell nyugtákat küldeni. Az eddig bemutatott algoritmusoknál pontosan meg volt határozva, hogy a folyamatok mikor rögzítsék a lokális állapotukat. A színezéses módszernél igaz ugyan, hogy minden folyamat kap egy jelzést, de lehet, hogy még ez előtt rögzítenie kell az állapotát.

A színezéses módszernél használt kontroll üzenet, ami körbemegy a folyamatokon hasonló a hullám sorozat módszernél használt hullámhoz. A kontroll üzenet és a hullám is összegyűjti a folyamatoktól az információt, ami a globális állapot meghatározásához kell. Kontroll üzenetből kettőnek is körbe kell mennie, ahhoz, hogy meglegyen minden szükséges információ, hullámból viszont elég egy.

A színezéses módszernél minden folyamat karban tart egy vektort annak alapján, hogy hány üzenetet küldött és hányat fogadott. Az Acharya-Badrinath algoritmus ugyanezen információ tárolásához két vektort használ.

A színezéses algoritmusnál a vektorok összesítése hasonlóan történik, mint a hullám sorozatoknál. Az elsőnél a kontroll üzenet, a másodiknál a hullám hordoz egy változót, aminek értékét mindig a folyamatok vektorainak segítségével határozza meg.

Az öt algoritmus

Az előző fejezetben öt olyan módszerrel ismerkedtünk meg, amelyek segítségével egy osztott rendszer globális állapotát rögzíteni lehet. Minden módszert a bemutatása után összehasonlítottuk az addig bemutatott módszerekkel. Két módszer egy-egy olyan tulajdonságát hasonlítottuk össze, melyek között valamilyen párhuzam vonható. Ebben a fejezetben a tapasztalataink alapján összefoglaljuk, hogy milyen szempontok alapján lehet az egyes algoritmusokat osztályozni.

Osztályozás alkalmazhatóság szerint

Az egyik leglényegesebb szempont az, hogy egy algoritmust az osztott rendszerek mely csoportjára lehet alkalmazni. Ha meg akarjuk határozni egy konkrét osztott rendszer adott pillanatbeli globális állapotát, csak olyan módszert alkalmazhatunk, amelyikhez adottak az alkalmazhatóság szükséges feltételei.

Először tehát nézzük meg, hogy hogyan lehet az osztott rendszereket csoportosítani. Az osztott rendszert korábban úgy definiáltuk, hogy bármely két folyamata között van irányított út az őt reprezentáló irányított gráfban. Most megnézzük, hogy ez valóban feltétele-e az ismertetett algoritmusok alkalmazhatóságának.

A Chandy-Lamport algoritmusnál a terminálás akkor garantált, ha a kezdeményező folyamat(ok)ból vezet irányított út mindegyik másik folyamatba, hiszen mindegyik folyamathoz el kell jutnia egy jelzésnek. Ahhoz tehát, hogy az algoritmus tetszőleges folyamat(ok) által kezdeményezve termináljon az kell, hogy a gráf erősen összefüggő legyen.

Az Acharya-Badrinath algoritmus alkalmazhatóságához, mint láttuk, ennél egy erősebb feltételre van szükség.

A hullámsorozatok esetén biztosítani kell, hogy a hullám minden folyamatot érintsen. Nem feltétlenül szükséges, hogy bármely két folyamat között vezessen irányított út.

A színezéses módszernél hasonló a helyzet, mint a Chandy-Lamport algoritmusnál. Egy tetszőleges folyamat spontán pirossá változik, és az összes többi folyamathoz el kell, hogy jusson egy jelzés vagy egy piros üzenet. Mivel piros üzenetet olyan folyamattól lehet csak kapni, amihez már eljutott egy jelzés vagy egy piros üzenet, ezért a kezdő folyamattól minden másik folyamathoz kell, hogy vezessen irányított út. Tehát a színezéses módszer alkalmazhatóságának feltétele az, hogy a gráf erősen összefüggő legyen.

Most tekintsük azokat az osztott rendszereket, amelyek megfelelnek a definíciónknak, vagyis van a gráfjukban irányított út bármely két csúcs között. Ezeket csoportosíthatjuk a szerint, hogy a csatornáik FIFO tulajdonságúak-e. A Chandy-Lamport algoritmus és a FIFO hullámsorozat módszer csak olyan osztott rendszerekre alkalmazható, ahol a csatornák FIFO tulajdonságúak. Az Acharya-Badrinath algoritmus alkalmazhatóságához egy ennél erősebb feltételre van szükség, arra, hogy az üzeneteket okozati sorrendben kézbesítsék. A nem FIFO hullámsorozatot és a színezéses módszer nem feltételezi, hogy az osztott rendszer csatornái FIFO tulajdonságúak. Ezt a két módszert természetesen alkalmazni lehet FIFO modellekre

is, de nem érdemes. A nem FIFO hullám sorozatnál a nyugtákkal felesleges terhelni a csatornákat, a színezéses módszernél pedig egyszerűbb a Chandy-Lamport algoritmus.

Osztályozás működés szerint

Az algoritmusokat vizsgálhatjuk olyan szempontból is, hogy hogyan kezdődnek, és van-e koordinátor folyamat. A Chandy-Lamport és a színezéses algoritmus úgy indul, hogy egy (vagy több) folyamat spontán rögzíti az állapotát. A Chandy-Lamport algoritmusnál a kezdeményező folyamatnak további kitüntetett szerepe nincsen. A színezéses módszer egyik változatánál a folyamatok a kezdeményező folyamatnak küldik el a rögzített állapotukat, és az úton lévő üzenetek másolatát. Ha vektorszámlálókat használunk, akkor nincs további kitüntetett szerepe a kezdeményező folyamatnak, de valamelyik folyamatnak el kell indítania a kontroll üzenetet. A hullámsorozatoknál nincs koordinátor folyamat, de a hullámnak el kell indulnia valahonnan. Az, hogy honnan indul és hogyan érinti az összes folyamatot, a hullám implementációjától függ.

Megnéztük, hogy melyik algoritmus hogyan indul, most megvizsgáljuk, hogy melyik hogy működik. Osztályozhatjuk az algoritmusokat a szerint, hogy melyik milyen eszközt használ a szinkronizációhoz. Mindegyik algoritmus használ valamilyen szinkronizáló üzenetet. A Chandy-Lamport algoritmus esetén csak a jelzések végzik a szinkronizációt. A hullámsorozatoknál a hullámok, a jelzések és nem FIFO esetben a nyugták együttesen végzik a szinkronizációt, és határozzák meg, hogy az egyes folyamatok mikor rögzítsék a lokális állapotukat. Az Acharya-Badrinath algoritmus esetén nagyon egyszerű a szinkronizáció. Egyetlen zsetont kell csak küldeni minden folyamatnak, és az amikor megkapja, rögzíti az állapotát. Az üzenetek okozati sorrendben való kézbesítése az, ami garantálja, hogy az ilyen módon rögzített globális állapot konzisztens. A színezéses módszernél a kezdeményező folyamat küld egy jelzést minden folyamatnak, de ennek csak akkor van szerepe, ha ez előbb érkezik meg a folyamathoz, mint az első piros üzenet.

Abból a szempontból is különböznek az algoritmusok, hogy hogyan rögzítik a csatornák állapotát. A Chandy-Lamport és az Acharya-Badrinath algoritmus esetén folyamat rögzíti a csatornák állapotát, a hullámsorozatoknál és a színezéses módszernél (ha vektorszámlálókat használunk) pedig a hullám illetve a kontroll üzenet. A Chandy-Lamport és a színezéses algoritmus akkor fejeződik be, amikor az összes útban lévőnek rögzített üzenet megérkezik a fogadóhoz. A többi algoritmusnál ezt nem kell megvárni, mert azok az elküldött és a fogadott üzenetekből számítják ki a csatornaállapotokat.

Nagyon fontos szempont, hogy melyik algoritmusnak hány üzenetet kell küldenie ahhoz, hogy rögzítse a globális állapotot, hiszen ez meghatározhatja, hogy melyiket akarjuk használni. Ez a kérdés a Chandy-Lamport és az Acharya-Badrinath algoritmus esetén könnyen megválaszolható. A Chandy-Lamport algoritmus a csatornák számával megegyező számú jelzést küld, az Acharya-Badrinath pedig annyit, ahány folyamat van. Az adatok összegyűjtéséhez az utóbbi még egyszer ugyanannyi üzenetet igényel, a Chandy-Lamport algoritmus pedig nem gyűjti össze az adatokat. A hullámsorozatoknál és a színezéses módszernél a jelzések és a nyugták számát meg lehet mondani, de a hullámokhoz szükséges üzenetek és a kontroll üzenetek száma az implementációtól függ.

	Chandy-Lamport	***Acharya-Badrinath***	***FIFO hullám sorozat***	***nem FIFO hullám sorozat***	***nem FIFO színezés***
csatornák FIFO tulajdonságát igényli-e?	igen	igen	igen	nem	nem
okozati kézbesítést igényel-e?	nem	igen	nem	nem	nem
van-e kezdeményező folyamat?	van	van	nincs	nincs	van
van-e koordinátor folyamat?	nincs	van	nincs	nincs	van/nincs
mikor rögzíti a lokális állapotokat?	spontán vagy első jelzés fogadásakor	zseton fogadásakor	hullám érintésekor	hullám érintésekor	spontán vagy mielőtt piros üzenetet fogadna
csatornaállapotot ki rögzíti?	folyamat, amelyik felé mutat	kezdeményező folyamat	hullám	hullám	kontroll üzenet
jelzést/zsetont használ-e?	igen	igen	igen	igen	igen (nem jelentős a szerepe)
nyugtát használ-e?	nem	nem	nem	igen	nem
jelzések/zsetonok/ nyugták száma	csatornák száma	folyamatok száma	csatornák száma	2∗(csatornák száma)	folyamatok száma

17. ábra: Az algoritmusok összehasonlítása

Melyiket használjuk?

Ha adott egy osztott rendszer, aminek rögzíteni akarjuk a globális állapotát, el kell dönteni, hogy melyik algoritmust akarjuk használni. Mindenképpen olyat kell használni, amelyikhez adottak a feltételek. Ha az osztott rendszerben bármely két folyamat között van csatorna, és a rendszer okozati sorrendben kézbesíti az üzeneteket, az Acharya-Badrinath algoritmus használata a legcélszerűbb, mert ehhez kell a legkevesebb üzenetet küldeni, vagyis ez terheli a legkevésbé a rendszert.

A hullámsorozatok használata akkor célszerű, ha globális állapotok egy sorozatát akarjuk rögzíteni, illetve ha valamilyen információkat akarunk begyűjteni a folyamatoktól vagy közvetíteni nekik. A hullámok kiválóan alkalmasak információ gyűjtésére és továbbítására.

Ha nem FIFO csatornák vannak a rendszerben, és el akarjuk kerülni a nyugták használatát, akkor a színezéses módszert célszerű használnunk. Itt azonban meg kell oldani, hogy egy folyamat előbb tudja rögzíteni az állapotát, mint ahogy megváltoztatná az éppen fogadott üzenet hatására.

A Chandy-Lamport algoritmust akkor célszerű használni, amikor a csatornák FIFO tulajdonságúak, és fontos, hogy üzenetek sorozatát rögzítsük csatornaállapotként, ne csak azonosítókat.

Ennek a fejezetnek az eredményeit a 17. ábrán látható táblázatban foglaltuk össze. A következő fejezetben mutatunk néhány példát arra, hogy hogyan lehet alkalmazni a rögzített globális állapotot.

A globális állapot alkalmazása

Korábban elmondtuk, hogy egy osztott rendszer globális állapotának ismerete mi mindenre jó lehet. Ilyen például egy stabil tulajdonság fennállásának megállapítása. Stabil tulajdonság például, hogy "a számítás befejeződött", "a rendszer holtpontra jutott" vagy, hogy "elveszett az összes zseton egy körben".

Azt is mondtuk, hogy arra is használható a globális állapot, hogy ha egy rendszerben az egyik folyamat meghibásodás miatt leáll, újra lehessen indítani az egész rendszert.

Stabil tulajdonság

Mint már korábban említettük, az osztott rendszer globális állapotainak halmazán értelmezett logikai értékeket felvevő Φ függvény, akkor stabil tulajdonság, ha Φ(S)=*igaz* esetén Φ(S')=*igaz* minden az S globális állapotból elérhető S' globális állapotra. Most egy olyan a Chandy-Lamport algoritmuson alapuló algoritmust fogunk bemutatni [3], aminek segítségével megállapítható, hogy egy adott stabil tulajdonság fennáll-e. Az algoritmust a következőképpen definiáljuk.

Bemenet: A Φ stabil tulajdonság.
Kimenet: A *biztos* logikai érték melyre teljesül, hogy:
$(\Phi(S_a) \rightarrow$ *biztos*) és (*biztos* $\rightarrow \Phi(S_f)$), ahol
S_a az a globális állapot, amelyben az algoritmus elkezdődik,
S_f amelyben befejeződik,
$\rightarrow$ a logikai implikáció.

Az algoritmus bemenete tehát a Φ függvény. A Φ(S) függvény értékét egy folyamat határozza meg úgy, hogy alkalmazza a külsőleg definiált Φ függvényt az S globális állapotra. A *biztos* logikai változó értékét egy speciális folyamat, F szimbolizálja a következőképpen:

(1) F belép egy speciális állapotba és ott marad. Ezzel szimbolizálja, hogy a kimenet *biztos*=*igaz*.
(2) F belép egy másik speciális állapotba és ott marad. Ezzel azt szimbolizálja, hogy a kimenet *biztos*=*hamis*.

Ki kell hangsúlyoznunk, hogy a *biztos*=*igaz* a rendszer azon állapotáról ad információt, amelyben akkor van, amikor az algoritmus befejeződik, míg a *biztos*=*hamis* az algoritmus kezdetén fennálló állapotról, hiszen *biztos*=*igaz*-ból következik, hogy Φ fennáll az algoritmus befejeződésekor, míg *biztos*=*hamis*-ból az következik, hogy Φ nem áll fenn az algoritmus kezdetekor. Ha *biztos*=*hamis*, nem tudunk következtetést levonni Φ az algoritmus befejeződésekor való fennállására

vonatkozóan.

A megoldás a stabilitás detektálására:

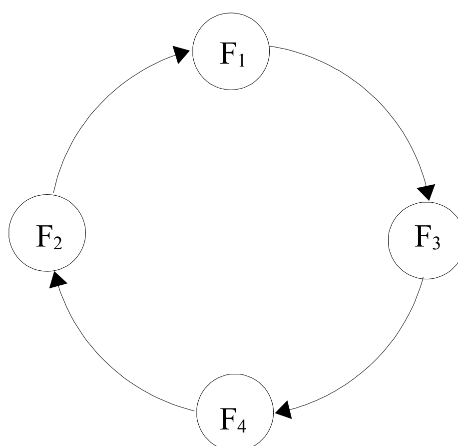
begin
    S' globális állapot rögzítése;
    biztos:=Φ(S')
end.

Ennek az algoritmusnak a helyessége az alábbi állítások következménye.

(1)S' elérhető S_a-ból,
(2)S_f elérhető S'-ből, és
(3)$\Phi(S_i) \rightarrow \Phi(S_j)$ minden S_j-re, ami elérhető S_i-ből.

Holtpont

Egy osztott rendszerhez tartozhatnak olyan erőforrások, amelyeket egy időben csak egy folyamat vehet igénybe. Ilyen például egy nyomtató, amelyre több számítógépről lehet nyomtatni. A holtpont a rendszer egy olyan állapota, amelyben folyamatok egy csoportjának minden tagja erőforrásokat igényel a csoport egy másik tagjától, és meghatározatlan ideig vár arra, hogy a kérése teljesüljön. Egy egyszerű példa a holtpontra, amikor két folyamat lefoglal egy-egy erőforrást, és mindkettő a másik által lefoglalt erőforrásra várakozik.



18. ábra: Holtpont várakozási gráfja

Amikor egy osztott rendszerben holtpont fennállását akarják vizsgálni, a globális állapotot sokszor várakozási gráffal modellezik. A várakozási gráf egy irányított gráf, amelynek csúcsai a folyamatok, és F_i csúcsból akkor vezet irányított él F_j-be, ha F_i kérte egy erőforrás használatát F_j-től, és F_j nem engedélyezte. A rendszer

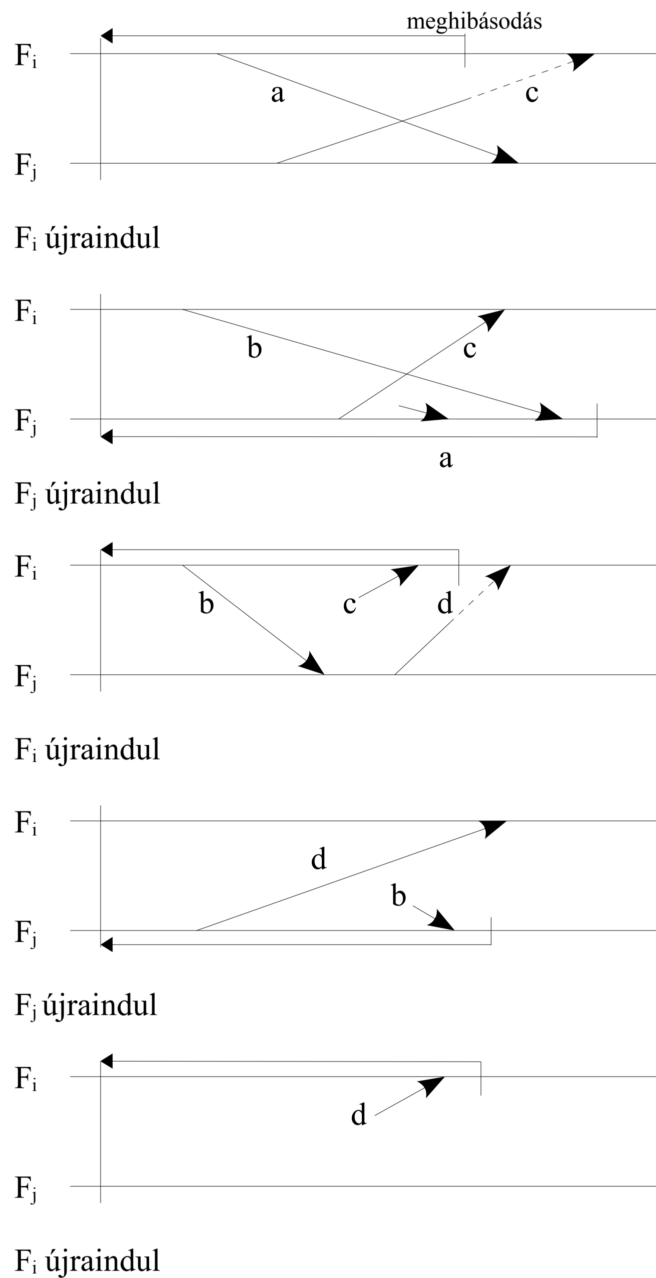
akkor van holtponton, ha a várakozási gráfban van kör. Ilyen helyzetet mutat a 18. ábra. A holtpont megállapítására alkalmas egy stabil tulajdonság fennállását vizsgáló algoritmus, ahol a stabil tulajdonság: “a várakozási gráfban kör van”.

Tekintsünk egy osztott rendszert, ami szerverekből és kliensekből áll [2]. A szerverek szolgátlatásokat nyújtanak, a kliensek igénylik a szolgáltatásokat. A szervernek ahhoz, hogy kiszolgáljon egy kérést, lehet, hogy magának is igényelnie kell egy másik szerver szolgátlatását. A szerverek és a kliensek távoli eljáráshívásokon keresztül kommunikálnak. Amikor egy klienst küld egy kérést egy szervernek, blokkolódik, amíg a kérését nem teljesítik. Ez a rendszer holtpontra juthat. Legyen F_0 a monitor folyamat, ami megállapítja, hogy a rendszer holtpontra jutott. Ehhez a rendszer várakozási gráfját használja. F_0 időnként felvétel készítését kéri a folyamatoktól egy *snapshot* üzenet küldésével. Erre válaszul mindegyik folyamat megküldi, hogy melyik folyamatok várnak rá és blokkolódtak e miatt. Minden Fi folyamat karban tart egy n hosszúságú *blocking* vektort, amelyben *blocking*[j]=*igaz* pontosan akkor, ha F_j folyamat az F_i folyamatra vár, és e miatt blokkolódott. Így amikor egy folyamat *snapshot* üzenetet kap az F_0 folyamattól nincs más dolga, mint elküldeni neki a *blocking* vektorát. A kapott *blocking* vektorok segítségével F_0 meg tudja állapítani, hogy van-e kör a várakozási gráfban. Az F_0 folyamat által futtatott kód a következő.

```
process F0:
        var     wfg: array [1..n] of array [1..n] of boolean;   % wfg[i,j]= ”Fj Fi-re vár”
                j, k: integer;
                m: message;
        while true do
                wait until deadlock is suspected;
                send [type: snapshot] to F1,..., Fn;
                for k:=1 to n do
                        receive m from Fj;
                        wfg[j]:=m.data;
                if (cycle in wfg) then system is deadlocked
        od
end F0;
```

Ellenőrzési pontok és visszagörgetés

Egy ellenőrzési pont egy folyamat egy rögzített és elmentett lokális állapota. Ahhoz, hogy meghibásodás miatt ne kelljen a folyamatot az elejéről indítani, időnként ellenőrzési pontokat kell készítenie, azaz rögzítenie a lokális állapotát. Amikor valami hiba történik, a folyamatot vissza lehet görgetni a legutóbbi ellenőrzési pontjához és onnan újraindítani. Ilyenkor a többi - de nem feltétlenül az összes - folyamatot is vissza kell görgetni ahhoz, hogy a rendszert konzisztens globális állapotból lehessen újraindítani.



19. ábra: Élő holtpont kialakulása

A helyzet nem ennyire egyszerű, mivel a folyamatokat nem lehet pontosan ugyanabban a pillanatban újraindítani. Tekintsük a 19. ábrán mutatott helyzetet. Az F_i folyamat meghibásodás miatt leáll mielőtt megkapná a *c* üzenetet. Visszamegy az ellenőrzési pontjához és értesíti az F_j folyamatot. F_i újraindul az ellenőrzési ponttól, elküldi *b*-t és fogadja *c*-t. Mivel az ellenőrzési pont az a üzenet elküldése előtt készült, most olyan, mintha F_i el se küldte volna *a*-t, de F_j megkapta. Ez az állapot inkonzisztens, tehát F_j-nek is vissza kell mennie a legutóbbi ellenőrzési pontjához, és értesítenie F_i-t. Miután F_j újraindul úgy tudja, hogy nem küldte el *c*-t, viszont F_i

megkapta. Így a globális állapot megint inkonzisztens, Fi-nek megint vissza kell mennie az ellenőrzési ponthoz. Tegyük fel, hogy ezt megteszi mielőtt megkapná a *d* üzenetet. Ezzel *b* elküldése elvész. Hogy a konzisztencia megmaradjon, F_j-nek is megint vissza kell mennie az ellenőrzési pontjához. Miután F_i újraindul, megkapja *d*-t, de vissza kell mennie mivel F_j visszament. Látható, hogy mindkét folyamat örökké arra kényszerül, hogy visszamenjen az ellenőrzési pontjához pedig csak egy hiba történt. Olyan visszagörgetés-újraindítás algoritmust fogunk mutatni [8], amelynél ilyen helyzet nem lép fel. Az algoritmus a lehető legkevesebb folyamatot lépteti vissza az ellenőrzési ponthoz egy hiba felmerülése után. Az ellenőrzési pontok lehetnek egy globális felvételt készítő algoritmus által rögzített, elmentett lokális állapotok.

Feltesszük, hogy az algoritmust egy folyamat hívja meg, amit vissza kell görgetni és újraindítani. Az algoritmus a kétfázisú commit protokollhoz hasonlóan működik. Az első fázisban a kezdeményező F_a folyamat kéri a többi folyamatot, hogy újrainduljon az ellenőrzési pontjától. Az F_a folyamat abban az esetben indítja újra az összes folyamatot, ha mindegyik hajlandó újraindulni. A második fázisban F_a kihirdeti döntését, és a folyamatok végrehajtják. Mivel mindegyik folyamat a kezdeményező döntése alapján cselekszik, ezért az algoritmus befejeződésekor a globális állapot konzisztens lesz.

Mint már említettük, a célunk az, hogy minél kevesebb folyamatot kelljen visszagörgetni. Ha egy F_i folyamatot visszagörgetünk egy olyan állapotba, ami az *a* esemény előtt van, akkor azt mondjuk, hogy Fi visszagörgetése megsemmisíti *a*-t. Ha egy F_i folyamatot vissza kell görgetni, akkor ennek következményeként csak azokat az F_j folyamatokat kell visszagörgetni, amelyek fogadtak olyan üzenetet, aminek a küldését F_i visszagörgetése megsemmisítette. Az F_i a folyamat a "készülj visszagörgetésre" üzenetéhez mindig csatol egy címkét. Az F_j folyamat ennek a címkének a segítségével dönti el, hogy vissza kell-e mennie az ellenőrzési pontjához.

Legyen *a* az utolsó üzenet, amit az F_i folyamat küldött az F_j folyamatnak mielőtt F_i a legutóbbi ellenőrzési pontját készítette volna.

$$last_smsg_i(j) = \begin{cases} t(s(a)), & \text{ha } a \text{ létezik} \\ \infty, & \text{különben} \end{cases}$$

Legyen *b* az utolsó üzenet, amit az F_i folyamat kapott az F_j folyamattól miután F_i rögzítette a legutóbbi ellenőrzési pontját.

$$last_rmsg_i(j) = \begin{cases} t(s(b)), & \text{ha } b \text{ létezik} \\ -\infty, & \text{különben} \end{cases}$$

Amikor az F_i folyamat kéri az F_j folyamatot, hogy újrainduljon, $last_smsg_i(j)$-t csatolja a kéréshez. F_j akkor indul újra az ellenőrzési pontjáról, ha $last_rmsg_j(i) > last_smsg_i(j)$.

Legyen $roll_cohort_i$ azon folyamatok halmaza, amelyeknek az F_i folyamat tud

üzenetet küldeni. Mindegyik F_i folyamatnak legyen egy $willing_to_roll_i$ változója. Amikor valamilyen okból F_i-t nem lehet visszagörgetni, a $willing_to_roll_i$ változó értéke nem. A kezdeményező F_a folyamat úgy indítja el az algoritmust, hogy minden $F_i \in roll_cohort_a$ folyamatnak küld egy "készülj visszagörgetésre" üzenetet, amihez csatolja $last_smsg_a(i)$-t. Az F_i folyamat akkor örökli ezt a kérést, ha $willing_to_roll_i$ igaz, $last_rmsg_i(a) > last_smsg_a(i)$, és F_i még nem örökölt másik visszagörgetés kérést. Ha F_i örökli a kérést, ő is küld egy "készülj visszagörgetésre" üzenetet minden $F_j \in roll_cohort_i$ folyamatnak, és csatolja hozzá $last_smsg_i(j)$-t, különben pedig $willing_to_roll_i$-vel válaszol.

Miután F_i kiküldte a kéréseit, megvárja míg mindegyik $F_j \in roll_cohort_i$ folyamat válaszol. A válasz lehet egyértelmű *igen* vagy *nem*, illetve *nem* abban az esetben, ha F_i észreveszi, hogy F_j meghibásodott. Ha van a válaszok között *nem*, akkor $willing_to_roll_i$ is *nem* lesz, különben nem változik. Az F_i folyamat $willing_to_roll_i$-t elküldi annak a folyamatnak, amelyiktől a kérést örökölte. Attól kezdve, hogy F_i örökli a kérést addig, amíg a kezdeményező folyamattól megkapja a választ, F_i nem küld az osztott számításhoz tartozó üzeneteket.

Ha a kezdeményező F_a folyamat minden $F_i \in roll_cohort_a$ folyamattól megkapja a választ és mindegyik *igen*, akkor a visszagörgetések mellett dönt, különben pedig egyik folyamatot sem görgeti vissza. F_a mindegyik folyamattal közli a döntését. Ha egy hiba miatt egy F_i folyamathoz nem jut el a döntés, akkor F_i blokkolódik addig, amíg nem értesül róla.

Összefoglalás

Ebben a dolgozatban megismerkedtünk a globális állapot fogalmával, és öt olyan módszerrel, amivel azt rögzíteni lehet. Mielőtt bemutattuk volna az algoritmusokat, ismertettük az osztott rendszerek alapvető fogalmait.

A módszerek ismertetése közben meghatároztunk olyan szempontokat, amelyek alapján össze lehet őket hasonlítani. Ezek a szempontok maguktól adódtak. Természetes volt például, hogy megvizsgáljuk, hogy melyik módszer milyen tulajdonságú osztott rendszerekre lehet alkalmazni, vagy hogy melyik hogyan rögzíti a csatornák állapotát.

A Chandy-Lamport és az Acharya-Badrinath algoritmust összehasonlítottuk egymással. Ez nagyon tanulságos volt. A két algoritmus alapvetően különbözik abból a szempontból, ahogyan a csatornák állapotát rögzítik. A többi algoritmus csatornarögzítési módszere a kettő közül valamelyikhez hasonló.

Az egyes algoritmusok bemutatása után leírtuk, hogy milyen párhuzamok vonhatók a már bemutatott algoritmusokkal, illetve, hogy milyen különbségek állapíthatók meg. Az algoritmusok bemutatása után rendszereztük a korábban meghatározott szempontokat, és azok alapján összefogottan vizsgáltuk mind az öt algoritmust.

A szempontokat is két csoportra osztottuk a szerint, hogy az osztott rendszerre vonatkoznak-e, amelyre az algoritmus alkalmazható vagy pedig az algoritmus működésére. Láttuk, hogy az algoritmusok már abban is különböznek, hogy milyen tulajdonságú osztott rendszerekre lehet őket alkalmazni. A működésüket a következő fő szempontok szerint osztályoztuk:

- hogyan kezdődnek,
- milyen eszközt használnak a szinkronizációhoz,
- hogyan rögzítik a csatornák állapotát,
- hány üzenet kell a globális állapot rögzítéséhez.

Miután megvizsgáltuk és összehasonlítottuk az algoritmusokat alkalmazhatóságuk és működésük szerint, elmondtuk, hogy melyik használatát milyen helyzetben javasoljuk. Végül példákat mutattunk a globális állapot alkalmazására.

Irodalomjegyzék

[1] **A. Acharya, B. R. Badrinath:** “Recording distributed snapshots based on causal ordering of message delivery.” *Information Processing Letters*, vol. 44, no. 6, pp. 317-321, 1992.

[2] **Ö. Babaoğlu, K. Marzullo:** “Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms.” In S. Mullender (ed.), *Distributed Systems*, pp. 55-96. Wokingham: Addison-Wesley, 2nd ed., 1993.

[3] **K. M. Chandy, L. Lamport:** “Distributed Snapshots: Determining Global States of Distributed Systems.” *ACM Trans. Comp. Syst.*, vol. 3, no. 1, pp. 63-75, Feb. 1985.

[4] **K. M. Chandy, J. Misra:** “Parallel Program Design: A Foundation.” Addison-Wesley, 1988.

[5] **S. Chen, Y. Deng, P. C. Attie, W. Sun:** “Optimal Deadlock Detection in Distributed Systems Based on Locally Constructed Wait-for Graphs.” *In Proceedings of the 16th Interntional Conference of Distributed Computing System*, pp. 613-619, 1996.

[6] **E. W. Dijkstra:** “The distributed snapshot of K. M. Chandy and L. Lamport.” Tech. Rep. EWD 864a, Univ. of Texas, Austin, Tex., 1984.

[7] **J. Helary:** “Observing Global States of Asynchronous Distributed Applications.” *In Proceedings of International Workshop on Distributed Algorithms*, vol. 392 of *Lect. Notes Comp. Sc.*, pp. 124-135. Berlin: Springer Verlag, 1989.

[8] **R. Koo, S. Toueg:** “Checkpointing and Rollback-Recovery for Distributed Systems.” *IEEE Transactions on Software Engineering*, vol. 13, no. 1, 1987.

[9] **L. Lamport:** “Time, Clocks, and the Ordering of Events in a Distributed System.” *Commun. ACM*, vol. 21, no. 7, pp. 558-565, July 1978.

[10] **F. Mattern:** “Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation.” *Journal of Parallel and Distributed Computing*, vol. 18, no. 4, 1993.

[11] **M. Raynal, M. Singhal:** “Logical Time: Capturing Causality in Distributed Systems.” *IEEE Computer*, vol. 29, no. 2, pp. 49-56, Feb. 1996.

[12] **A. S. Tanenbaum, M. Van Steen:** “Distributed Systems.” Prentince Hall, New Jersey, 2002.