Transakce

PA 150 ◇ Principy operačních systémů

Jan Staudek
http://www.fi.muni.cz/usr/staudek/vyuka/

Verze : podzim 2020

Pojem transakce

- Již staří Římané . . . *transigo – provést*
- V IT: transakce (často budeme uvádět pouze symbol T)–
  - ✓ logická jednotka běhu procesu (vlákna) ve které se zpřístupňují a příp. modifikují datové položky uchovávané v bázi(ích) dat,
  - ✓ resp. posloupnost dílčích, samostatných požadavků klienta na server,
- T zachovává jako celek dvě základní vlastnosti:
  - ✓ atomicita – celá se úspěšně dokončí nebo nemá žádný účinek

    (nemá účinek pokud se nemůže úspěšně dokončit, např. v případě výpadku některého ze serverů (zdrojů) podílejících se na transakci)
  - ✓ izolovanost –

    neovlivňuje se operacemi souběžně realizovanými transakcemi

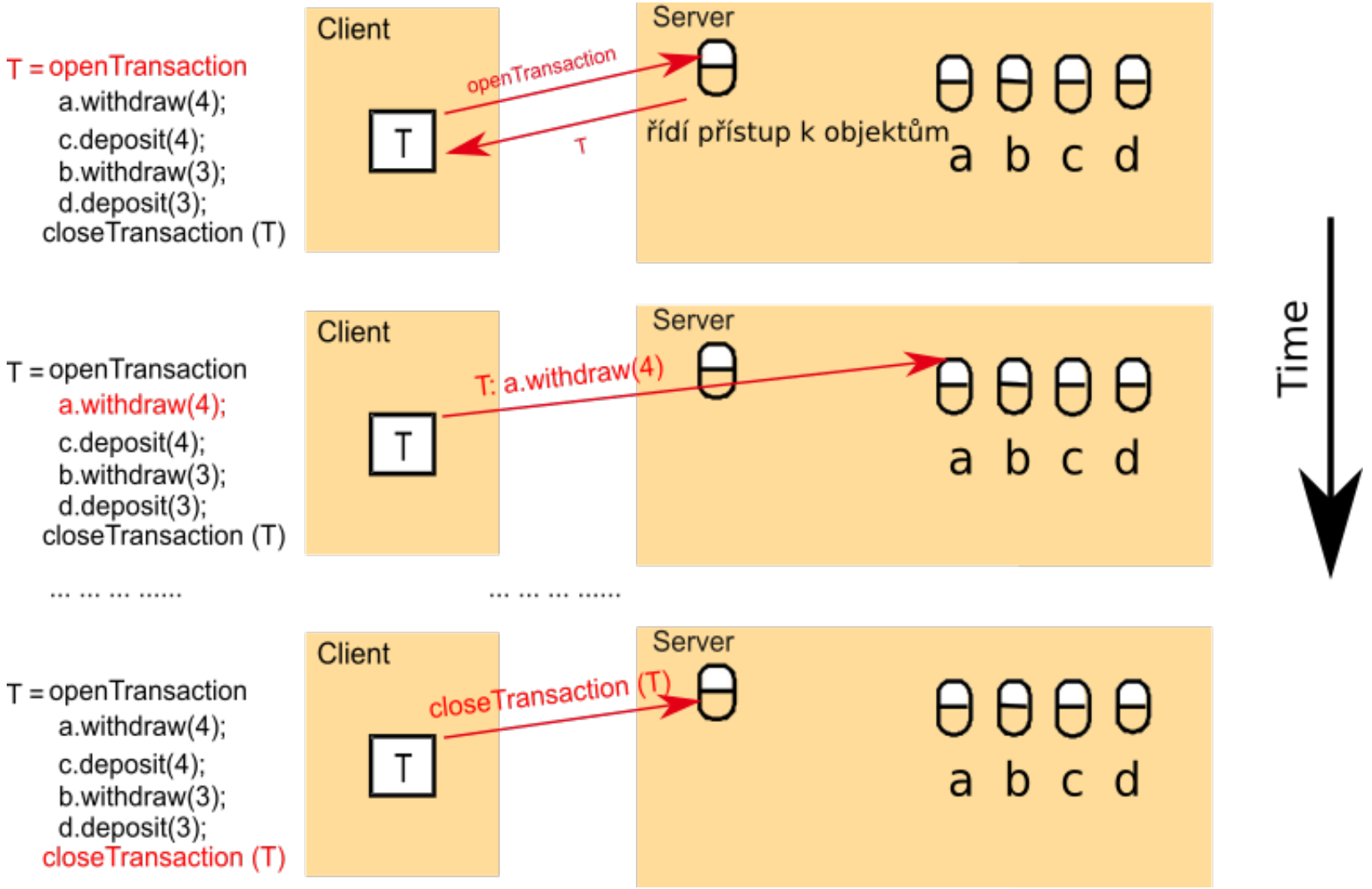
Pojem transakce

- Příklad transakce – objednávka a dodání zboží
  - ✓ Zpracování objednávky i dodávky se může týkat např. skladových, fakturačních a dodavatelských záznamů
  - ✓ Zpracování záznamů musí být synchronizované, např. nesmí se dodat zboží zákazníkovi, aniž by proběhla fakturace
- Transakci vymezují hraniční definiční konstrukty

  začátek transakce (transaction begin, t_begin. . .)
  konec transakce (transaction end, t_end, . . .)

  (explicitně uvádíme pouze v případě nejednoznačnosti vyjádření)
- Transakce se skládá ze všech operací uzavřených mezi začátkem transakce a koncem transakce
- Transakce je jednoznačně identifikovatelná

Konceptuální schéma provedení transakce



Modelová transakce často používaná při výkladu

- Převod částky mezi účty
- Imperativní charakter vyjádření transakce:

```
read (A);      % A, B – DB proměnné čtené/zapisované do/z RAM
A:=A–50;       % A, B – kopie DB proměnných A, B v RAM
write (A);
read (B);
B:=B+50;
write (B);
```

- Objektové vyjádření téže transakce:

```
a.withdraw (50);      % a, b objekty v databázi
b.deposit (50);
```

Pojem transakce a transakčního zpracování

- Cíl transakčního zpracování
  - ✓ Objekty spravované serverem jsou v konzistentním stavu i v případech, kdy jsou tyto zpřístupňované více transakcemi souběžně, a to i v případech obnovy po výpadku výpočetního systému
  - ✓ Konzistentní stav – stav vyhovující jistým integritním omezením
  - ✓ Integrita – celistvost, soudržnost, neporušenost, . . .
- Server spravované objekty udržuje jako obnovitelné objekty
  - ✓ Pro uchovávání informací o objektech, jejichž kopie server dočasně uchovává v RAM, používá server permanentní paměť (disk)
  - ✓ Tyto informace server používá při obnově po výpadku serveru

Pojem transakce a transakčního zpracování

- Transakci klient specifikuje jako množinu operací na objektech, kterou musí servery spravující tyto objekty provádět jako logicky nedělitelnou jednotku zpracování
- Operace jedné transakce si nemohou zpřístupňovat dílčí (mezi)výsledky jiných transakcí – transakce jsou atomické jednotky
- Řízení transakčního zpracování musí zaručit, že
  - ✓ buďto se provede celá transakce a její výsledek se trvale uchová v permanentní paměti nebo
  - ✓ v případě výpadku se účinky nedokončené transakce plně eliminují

Pojem transakce (T) a transakčního zpracování

- Transakční zpracování budeme studovat nejprve na případu transakcí prováděných na v nedistribuovaném systému (multitasking, příp. podporovaný multiprocesorem), na případu distribuovaných transakcí řešených v DS

Příklad prostředí transakčního zpracování

Account
rozhraní vzdáleného objektu -- účtu

deposit(amount)
deposit amount in the account
withdraw(amount)
withdraw amount from the account
getBalance() → *amount*
return the balance of the account
setBalance(amount)
set the balance of the account to amount

account, účet
objekt udržovaný serverem
reprezentovaný vzdáleným objektem
s rozhraním *Account*

Branch
rozhraní vzdáleného objektu -- pobočky banky

create(name) → *account*
create a new account with a given name
lookUp(name) → *account*
return a reference to the account with the given name
branchTotal() → *amount*
return the total of all the balances at the branch

Každá pobočka banky
je reprezentovaná vzdáleným objektem
s rozhraním *Branch*

Podpora transakčního zpracování – TPM

- *Transaction Processing Monitor*, TPM – typicky část middleware
  - ✓ Vrstva služeb mezi aplikační vrstvou a vrstvou „OS & síťování"
    - Typický příklad prostředí pro transakční zpracování – CORBA
    - TPM může být součástí systému řízení báze dat
- TPM zajišťuje, že se každá transakce ukončí a databázi zanechá v neporušeném (konzistentním) stavu,
- TPM podporuje provádění operací vymezujících transakci
  - ✓ t_begin – start transakce, generování id transakce
  - ✓ t_end – ukončení transakce
    - řádně, s výsledkem t_commit, transakce je provedená, nebo
    - násilně, po poruše/výpadku, na žádost, . . . , s výsledkem t_abort, transakce je zrušená, příp. zkrachovalá
  - ✓ násilné ukončení (zrušení, krach) transakce vyvolá v TPM operaci rollback, tj. vrácení transakčního prostředí do původního stavu, ve kterém bylo při startu transakce

Podpora transakčního zpracování – TPM

- Další problémy, které musíme vesměs pomocí TPM řešit
  - ✓ *read* (A); A:=A–50; *write* (A); *read* (B); B:=B+50; *write* (B);
  - ✓ dopady poruch – výpadků hardware, systému, . . .
    a obnov po poruše
    - *read* (A); A:=A–50; *write* (A); VÝPADEK
    - OBNOVA,
      řešená opakovaným spuštěním aplikace vyvolávající transakci
    - *read* (A); A:=A–50; *write* (A); *read* (B); B:=B+50; *write* (B);

      dekrement A se opakuje, aniž by se navrátila iniciální hodnota A

Podpora transakčního zpracování – TPM

✓ dopady souběžnosti – provádění více transakcí souběžně (řádky = čas)

T1:	T2:	
read (A);		
A:=A–50;	*read* (A);	% T2 čte z disku původní hodnotu A
write (A);	A:=A–100;	
read (B);	*write* (A);	% T2 eliminuje účinek T1 na A
B:=B+50;	*read* (B);	% T2 čte z disku původní hodnotu B
write (B);	B:=B+100;	
	write (B);	% T2 eliminuje účinek T1 na B;

Pojem transakce a transakčního zpracování

- Cílem podpory transakčního zpracování
  je zajistit, aby všechny objekty, které jistý server (DB) ovládá (spravuje, řídí), permanentně zůstávaly v konzistentním stavu, a to i v případech:
  - ✓ výpadků serveru – jistý server nepokračuje v běhu a ostatní procesy nemusí být schopné tento stav zjistit
  - ✓ výpadků doručení zpráv – zpráva vložená do výstupního bufferu není doručena do vstupního bufferu
  - ✓ souběžného provádění více transakcí manipulujících se sdílenými daty
- Histrorický původ transakcí – DBMS
  - ✓ *Data Base Management System*, systém řízení báze dat
  - ✓ transakcí se rozumělo provedení programu, který přistupuje do báze dat

Pojem transakce a transakčního zpracování

- Transakce v z pohledu serveru spravujícího zpracovávané objekty:

  identifikovatelné posloupnosti klientských požadavků na operace s objekty

- Klientský pohled na transakci:

  posloupnost operací formovaných definicí jejího počátku a konce do „jednoho kroku“ transformujícího stav serveru

  – z jednoho konzistentního stavu (před spuštěním transakce)

  – do jiného konzistentního stavu (po dokončení transakce)

Požadované vlastnosti transakcí – A, Atomicity

- Buď se provedou všechny operace transakce nebo žádná z nich, požadovaná vlastnost – All or nothing
- Transakce
  buď skončí úspěšně, pak účinky všech jejích operací jsou trvale zaznamenané v relevantních objektech v permanentní paměti,
  nebo selže nebo je úmyslně zrušená,
  pak nevykáže žádný účinek
- provedení A:=A–50; bez provedení B:=B+50; způsobí nekonzistentnost báze dat, dojde ke ,,ztrátě'' peněz
- TPM musí zajistit, aby částečně provedené transakce (tj. při běhu padnou nebo jsou záměrně zrušené) neovlivnily stav báze dat

Požadované vlastnosti transakcí – A, Atomicity

- s kolekcí operací transakce se musí zacházet jako s jednou jedinou operační jednotkou — vždy, za všech podmínek
- zajištění atomicity transakce kritickou sekcí (serializací celých transakcí) je nedostačující řešení, požaduje se zajištění atomicity i v případě poruch (výpadků)
- Informace o modifikaci DB dat je zapisována do souboru na disku – deník, *log*
- Jestliže se transakce nedokončí, v DBS se z deníku obnoví původní hodnoty DB dat
- Po úspěšném dokončení transakce jsou všechny její účinky trvale zapamatované v permanentní paměti, tj. uložené na disku či na jiném, energeticky nezávislém médiu.

Požadované vlastnosti transakcí – C, Consistency

- provedení transakce neporuší konzistenci báze dat,
  vlastnost C, Consistency
  - ✓ konzistentnost lze kontrolovat prověřováním plnění vhodných požadavků na konzistenci, např.:
    – implicitní integritní omezení – $A+B=const$
  - ✓ validní provedení T musí zachovat konzistenci báze dat
    – nově spouštěná transakce vždy musí vidět konzistentní bázi dat
  - ✓ chybně provedená T by mohla způsobit nekonzistentnost báze dat
  - ✓ platí: během provádění T může být konzistence dočasně narušena
    read (A); A:=A–50; *write* (A); nekonzistentní stav – $A+B\neq const$
    read (B); B:=B+50; *write* (B); konzistentní stav – $A+B=const$
  - ✓ už i izolované provedení individuální transakce nesmí narušit konzistenci báze dat
  - ✓ idea zajišťování konzistentnosti: – odpovědnost leží na programátorovi, který transakci navrhuje/kóduje

Požadované vlastnosti transakcí – I, Isolation

- ✓ T_1: *read* (A); A:=A–50; *write* (A); přerušení běhu T_1
- ✓ T_2: *read* (A); *read* (B); *print* (A+B);
- ✓ obnova běhu T_1: *read* (B); B:=B+50; *write* (B);

□ souběžné provádění více transakcí nesmí ovlivňovat výsledek jednotlivých transakcí, vlastnost I, Isolation

- ✓ v uvedeném příkladu vidí T_2 nekonzistentní bázi dat, zatímco provedení T_2 po „*write* (B);" v T_1 by bylo validní
- ✓ systém (TPM) musí zajistit pro každý pár souběžně řešených transakcí T_i a T_j, aby se z hlediska každé transakce souběh jevil
  – jakoby T_j skončila dříve než se spustila T_i
  – nebo jakoby se T_j spustila po dokončení T_i
- ✓ triviální/naivní idea zajišťování izolovanosti – plná serializace transakcí je mnohdy neefektivní řešení

Požadované vlastnosti transakcí – I, Isolation

- ✓ souběžné provádění více transakcí nesmí ovlivňovat výsledek jednotlivých transakcí
- ✓ Souběžné provádění transakcí zanechá systém ve stavu, který je rovnocenný stavu systému v případě, že by se tyto transakce prováděly jedna po druhé, sekvenčně v některém z možných pořadí
- ✓ Zajištění izolace je odpovědností funkční složky DBS případně monitoru transkačního zpracování, TPM, nazývané systém řízení souběžnosti, *Concurrency-Control System*

Požadované vlastnosti transakcí – D, Durability

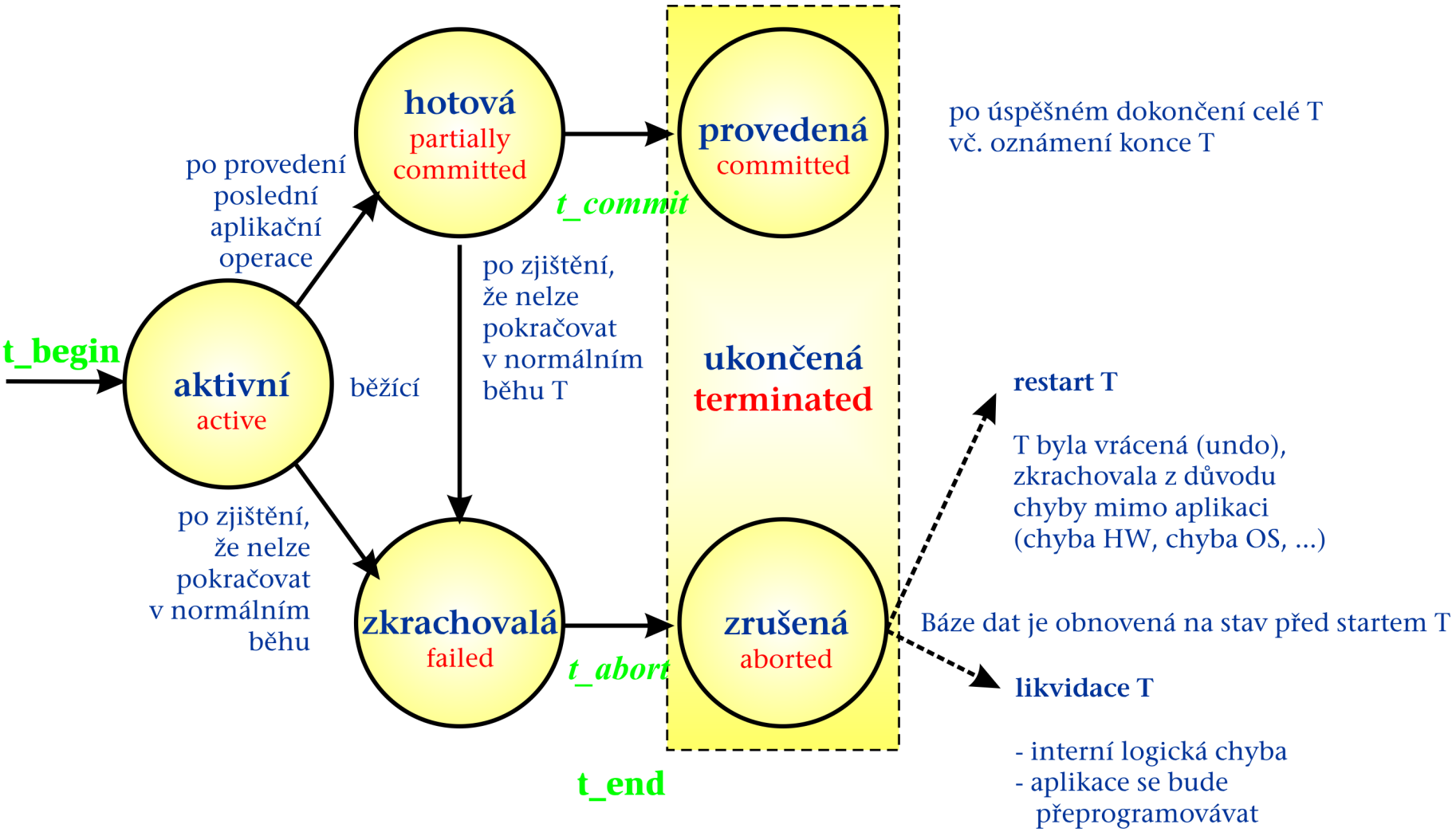
- výsledky transakce jsou stálé, trvalé, vlastnost D, Durability
  - ✓ jakmile se T úspěšně dokončí, změny provedené touto T jsou trvalé, a to i případě poruch/výpadků systému
  - ✓ idea zajišťování trvalosti:

    informace o změnách provedených T v DB se před dokončením T vypisují na disk a při obnově systému po poruše se tyto informace použijí pro rekonstrukci změn
  - ✓ když serverový proces (neočekávaně) padne díky poruše hardware nebo software, změny provedené všemi úspěšně ukončenými (provedenými) transakcemi musí být pamatované v permanentní paměti, aby obnovený (nový) serverový proces mohl objekty obnovit, aby se vyhovělo podmínce All or nothing (atomicity)
  - ✓ Jakmile se transakce úspěšně provede, všechny transakcí provedené modifikace v DB přetrvávají, a to i v případě, že systém po dokončení transakce selže.

Požadované vlastnosti transakcí – D, Durability

- ✓ Výpadek počítačového systému může způsobit ztrátu dat v hlavní paměti, ale data zapsaná na disk se neztratí.
- ✓ Zajištění trvalosti má na starosti funkční část DBS/TPM zvaná správce obnovy, *Recovery Manager*, zajišťující rovněž atomicitu
- ✓ Trvanlivost lze zaručit zajištěním
  - Modifikace provedené transakcí byly zapsány na disk ještě před ukončením transakce.
  - Informace o modifikacích provedených transakcí zapsaná na disk je dostatečná k tomu, aby DBS při restartování systému po poruše aktualizace zrekonstruoval

Stavy transakce



Generická definice transakce pro účely studia

- Transakci (T) charakterizujeme pouze posloupností operací *read* a/nebo *write* pracujících s objekty zpracovávanými transakcí končící operací t_end s výsledkem *t_commit* nebo *t_abort*
  - ✓ aplikační manipulace s objekty v transakci abstrahujeme
  - ✓ implicitně předpokládáme validní transakce zachovávající konzistenci

Hotová transakce, ***partially commited***

- hotovou (*partially committed*) transakci,
  tj. transakci hotovou se všemi aplikačními operacemi,
  koordinátor transakce končí při aktivaci t_end
  s výsledkem *t_commit*
  - ✓ hotová T v tom případě přechází do stavu provedená (*commited*)

Zkrachovalá transakce, *failed*

- na krach, výpadek (z důvodu logické chyby, poruchy hardware systému apod.) reaguje TPM převodem aktivní nebo hotové transakce T do stavu zkrachovalá T
  - ✓ přechodem do stavu zkrachovalá končí T svoji činnost předčasně
  - ✓ zkrachovalou T koordinátor končí při aktivaci t_end s výsledkem *t_abort*
  - ✓ neúspěšně (předčasně) končící T (zkrachovalá) přechází po výsledku *t_abort* do stavu zrušená (*aborted*)
  - ✓ pokud ke zrušení došlo z důvodu interní logické chyby aplikace, aplikace končí, zjevně musí být přeprogramovaná
  - ✓ zrušení T z důvodu výpadku vyvolá operaci vrácení (*rollback*), aby se zrušily veškeré změny, ke kterým díky jejímu dosavadnímu provedení došlo
  - ✓ obnovu původního stavu TPM dosáhne provedením operace undo
  - ✓ účinek provedené T zrušit/obnovit/vrátit (provedením *t_abort*) nelze

Zrušená transakce, *aborted*

- neukončila úspěšně své provedení
- nesmí mít žádný vliv na stav databáze
  - ✓ změny v databázi provedené zrušenou transakcí musí být zrušeny, provedením *undo* zápisů do DB se transakce vrátí zpět, *roll-back*
  - ✓ za zrušení odpovídá *Recovery Manager*, který udržuje deník, *log*. Každá modifikace prováděná transakcí je nejprve zaznamenána do deníku. V záznamu deníku se uvede
    – identifikátor transakce,
    – hodnota datové položky před modifikací a
    – hodnota po modifikaci.
  - ✓ Poté se modifikuje proměnná v DB
  - ✓ Udržování deníku umožňuje provést v případě obnovy po poruše během provádění transakcí
    – jak *redo* k zajištění atomicity a trvalosti
    – tak i *undo* k zajištění atomicity

Zrušená transakce, *aborted*.

- Problém pozorovatelných externích zápisů
  - ✓ výpisy na obrazovku, zaslání e-mail, . . .
  - ✓ čisté řešení – zápisy provádět jen ve stavu provedená transakce
  - ✓ pro dlouhodobé transakce toto řešení nemusí vyhovovat
- Problém nutnosti likvidace provedených transakcí
  - ✓ lze provést pouze adekvátními kompenzačními transakcemi
  - ✓ kompenzační transakce nemůže řešit DBS, TMP, . . .
    musí být řešeny na aplikační úrovni, nemusí to být triviální

Izolovanost souběžných transakcí

- Důvody k souběžnému řešení transakcí
  - ✓ vyšší propustnost a účinnější využití zdrojů (méně prostojů CPU, disků, ...)
  - ✓ snížení doby čekání proti sériovému provádění transakcí
- analogie multiprogramování na úrovni OS
- ne všechna prokládání běhů transakcí jsou validní, DBS/TPM musí koordinovat prokládání běhu souběžných transakcí tak, aby se zabránilo narušení konzistenci databáze.
- Činí tak pomocí mechanismů řízení souběžnosti – správou souběžnosti

Podpora transakčního zpracování

- Transakci vytváří a řídí její ukončení koordinátor transakce, komponenta TPM s rozhraním TPM – aplikace
- Rozhraní TPM – aplikace používá aplikační klient
  - ✓ openTransaction() → *trans;*
    - TPM startuje transakci a vrací jedinečný id transakce, *trans;*
    - strukturální notace používaná v obrázcích – t_begin
  - ✓ closeTransaction(trans) → (commit, abort);

    TPM končí transakci a vrací způsob ukončení:
    - pokud byla operační část transakce hotová , transakce se stane provedená, funkce vrací *commit*
    - pokud transakce zkrachovala, stane se zrušená, funkce vrací *abort*,
    - strukturální notace používaná v obrázcích, – t_end
  - ✓ abortTransaction(trans); explicitní žádost klienta o násilné ukončení (zrušení) transakce

Podpora transakčního zpracování

- Požadavky klienta na server zadávané v rámci transakce musí být identifikované id transakce *trans*
  - ✓ id transakce *trans* může být jedním z parametrů požadavku, např. *X.deposit(trans, amount)*
  - ✓ id transakce *trans* může být implicitně dodávané ke všem požadavkům příslušného klienta mezi

    openTransaction a

    closeTransaction nebo *abortTransaction*

    – takto to řeší např. middleware CORBA

    V příkladech id transakce *trans* explicitně neuvádíme.

Podpora transakčního zpracování

- Transakci může násilně ukončit i server,
  pak musí chybový výsledek sdělit klientovi
- Co dělá server, když neočekávaně dojde k výpadku ?
  - ✓ vypadne server a jeho činnost je posléze obnovená

    nový proces serveru musí násilně ukončit všechny transakce, které nejsou provedené a prostředí vrátit do stavu platného po posledních provedených transakcích

    použije obnovovací operaci (*recovery*, redo), kterou obnoví hodnoty objektů vytvořené provedenými transakcemi
  - ✓ vypadne klient během řešení jím spuštěné transakce,

    server nastavuje pro každou transakci expirační dobu a pokud relevantní klient do ní neprovede volání t_end, ukončí server transakci násilně

Podpora transakčního zpracování

Co dělá klient, když vypadne server ?

- ✓ buďto se klient dozví o výpadku serveru uplynutím časového limitu, transakci neukončuje a považuje ji za zkrachovalou,

  zkrachovalé transakce se na úrovni serveru automaticky likvidují, (princip atomicity)
- ✓ nebo je server ještě v průběhu transakce obnovený, pak transakce není validní a klient o tom musí být informovaný např. při zadání příští operace transakce (výjimkou, . . .)
- ✓ v obou případech klient formuluje plán dokončení nebo zanechání aplikace, jejíž součástí byla postižená transakce (formulaci může konzultovat s lidským uživatelem)

Souběžné řešení transakcí, problém dosažení izolovanosti

- přínos souběžného řešení transakcí
  - ✓ lépe se využívá výkon procesoru a disků a tím se dosahuje vyšší propustnosti transakcí – v době kdy jedna T čeká na konec IO operace a nepoužívá procesor, využívá procesor jiná T
- mechanismy podporované v TPM pro dosažení izolovanosti jednotlivých souběžně prováděných transakcí nazýváme schémata řízení souběžnosti transakcí (*Concurrency control schemes*)
  - ✓ zajišťují řízení interakcí mezi souběžnými T, jejich cílem je zabránit narušení konzistentnosti báze dat při souběžném řešení více T
  - ✓ souběžné provedení více T musí být ekvivalentní jejich sériovému (postupnému) provedení v některém ze všech možných pořadí

    tato vlastnost řešení souběhu transakcí se nazývá serializovatelnost (*serializability*)

Souběžné transakce, serializovatelnost, sériový plán

- Hledáme nástroj pro zachování konzistence báze dat při souběžném řešení transakcích
- Nechť každá T jisté skupiny transakcí zachovává izolovaným provedením konzistenci báze dat (T je validní)
- Pak každé postupné (sériové) provedení těchto transakcí rovněž zachovává konzistenci báze dat
- Postupné (sériové) řešení více T lze zajistit jejich implementací jako kritické sekce
  - ✓ toto může být mnohdy příliš restriktivní, tudíž neefektivní
- Efektivnější provádění transakcí se dosáhne zajištěním serializovatelnosti souběhu transakcí pomocí služeb TPM řízených algoritmy (schématy) řízení souběžnosti transakcí

Souběžné transakce, serializovatelnost, seriový plán

- Schémata řízení souběžnosti transakcí implementují jistý plán
- Plán vymezuje možná pořadí provádění instrukcí transakcí – konkrétně vymezuje, které (nekonfliktní) operace v souběžně řešených T lze provádět v čase překrývaně
  - ✓ Kdy jsou operace v souběžných transakcích nekonfliktní ?
    Pokud operují se stejnou proměnnou pouze operacemi *read*, nebo pokud operují s různými proměnnými
- Plán je sériový plán, pokud se instrukce, náležející jedné transakci, v něm objevují jako kontinuální bloky, tj. jsou provedeny nepřerušovaně
  - ✓ pro N transakcí existuje $N!$ sériových plánů
  - ✓ všechny sériové plány jsou validní, protože předpokládáme, že všechny podle nich realizované transakce jsou validní a jsou prováděné bez přerušení

Souběžné transakce, serializovatelnost, sériový plán

- Plán je nesériový plán pokud reprezentuje provádění transakcí, která se v čase se překrývají (multiprocesing) příp. se po částech prokládají (multitasking)

  ✓ Počet různých sériových plánů
  pro n (sekvenčně prováděných) transakcí je $n!$.

  Počty souběžně řešených transakcí mohou být stovky, tisíce, ...

  ✓ pro ilustraci – $\mathbf{100! = 9.332621544 \times 10^{157}}$,
  doba existence vesmíru se odhaduje na cca $\mathbf{13 \times 10^9}$ let

  ✓ Vzhledem ke všem možným způsobům prokládání operací souběžně řešených transakcí, počet všech možných nesériových plánů
  je MNOHEM větší než $n!$

Souběžné transakce, serializovatelnost, sériový plán

- Výsledek provedení nesériového plánu
  - nemusí nutně být nesprávný
  - je správný, pokud se jedná o tzv. serializovatelný plán
- Plán jisté skupiny T je serializovatelný, pokud je ekvivalentní některému sériovému plánu provedení této skupiny T
- Na bázi hledání serializovatelnosti (*Serializability*) transakcí hledáme serializovatelný plán (tj. plán, který není nutně sériový), který specifikuje jisté časové pořadí provádění instrukcí souběžných T tím, že
  - ✓ určuje pořadí provádění všech instrukcí všech T řešených souběžně,
  - ✓ zachovává pořadí provádění instrukcí, ve kterém se instrukce vyskytují v jednotlivých T
  - ✓ a zajišťuje, aby výsledek prokládaných provedení transakcí byl ekvivalentní jejich některému sériovému provedení

Konfliktní operace, konfliktově serializovatelný plán

- Z hlediska dosažení serializovatelnosti hrají roli tzv. konfliktní operace
  - ✓ Mějme plán S řešící transakce T_i a T_j a operace O_i a O_j v transakcích T_i a T_j
  - ✓ Operace O_i a O_j jsou konfliktní, pokud přistupují ke stejné datové položce a alespoň jednou operací je operace write
- Nejsou-li operace O_i a O_j konfliktní, pak plány S a S', které se liší pouze časovým pořadím provedení O_i a O_j, jsou ekvivalentní
- idea zajištění ekvivalentnosti plánů – konfliktová serializace
  - ✓ Konfliktní operace na všech zpřístupňovaných objektech v souběžně řešených transakcích podle serializovatelného plánu se musí provádět v pořadí shodném s pořadím těchto operací v některém ze sériových plánů provedení těchto transakcí

Konfliktní operace, konfliktově serializovatelný plán

- mezi operacemi *read* a *write* mohou transakce provádět nad lokálními proměnnými transakce (kopiemi DB hodnot) libovolné operace
  - ✓ v dále uváděných zjednodušených plánech vesměs ignorujeme všechny instrukce jiné než *read* a *write* hodnot pamatovaných v bázi dat (datových položek), uvádíme pouze instrukce *read* a *write*
- Pokud lze z nesériového plánu S vytvořit ekvivalentní sériový plán S' pouhou změnou pořadí provedení nekonfliktních operací, pak nesériový plán S je konfliktově serializovatelný
- Transakční monitor musí připustit souběžné provádění transakcí pouze podle konfliktově serializovatelných plánů

Pravidla určující konfliktnost operací

Operations of different transactions		*Conflict*	*Reason*
read	*read*	No	Because the effect of a pair of *read* operations does not depend on the order in which they are executed
read	*write*	Yes	Because the effect of a *read* and a *write* operation depends on the order of their execution
write	*write*	Yes	Because the effect of a pair of *write* operations depends on the order of their execution

Příklady sériových plánů P1: $< T_1 ; T_2 >$ a P2: $< T_2 ; T_1 >$

- ✓ T_1 přenáší 50 Kč z účtu A na účet B
- ✓ T_2 přenáší 10% hodnoty z účtu A na účet B
- ✓ požadavek konzistence: $A+B=konst$

P1

T_1	T_2	A	B
read (A)		1000	
A:=A-50			
write(A)		950	
read (B)			2000
B:=B+50			
write(B)			2050
	read (A)	950	
	x:=A*0,1		
	A:=A-x		
	write(A)	855	
	read (B)		2050
	B:=B+x		
	write(B)		2145

855 + 2145 = 3000

čas

P2

T_1	T_2	A	B
	read (A)	1000	
	x:=A*0,1		
	A:=A-x		
	write(A)	900	
	read (B)		2000
	B:=B+x		
	write(B)		2100
read (A)		900	
A:=A-50			
write(A)		850	
read (B)			2100
B:=B+50			
write(B)			2150

850 + 2150 = 3000

Příklad plánů, které nejsou sériové

- ✓ plán P3 není sériový, je serializovatelný (ekvivalentní plánu P1)
- ✓ plán P4 není sériový a nezajišťuje platnost omezení $A+B=konst$
  správce souběžnosti nesmí realizaci takového plánu povolit

P3

T_1	T_2	A	B
read (A)		1000	
A:=A-50			
write(A)		950	
	read (A)	950	
	x:=A*0,1		
	A:=A-x		
	write(A)	855	
			2000
read (B)			
B:=B+50			
write(B)			2050
	read (B)		2050
	B:=B+x		
	write(B)		2145

855 + 2145 = 3000

čas

P4

T_1	T_2	A	B
read (A)		1000	
A:=A-50			
	read (A)	1000	
	x:=A*0,1		
	A:=A-x		
	write(A)	900	
	read (B)		2000
write(A)		950	
read (B)			2000
B:=B+50			
write(B)			2050
	B:=B+x		
	write(B)		2100

950 + 2100 = 3050

Příklad plánů, které nejsou sériové

- plán P5 je nesériový a není konfliktově serializovatelný,
  správce souběžnosti nesmí realizaci takového plánu povolit
  transakce T3 a T4 musí být řešeny sériově,
  $< T3, T4 >$ nebo $< T4, T3 >$.

P5

T_3	T_4
read (Q)	
	write (Q)
write (Q)	

přípustné plány běhu transakcí T3 a T4

T_3	T_4
read (Q)	
write (Q)	
	write (Q)

T_3	T_4
	write (Q)
read(Q)	
write (Q)	

Problémy souběžnosti transakcí

- Sériové provedení transakcí T a U, plán 1
  - ✓ Iniciální stavy účtů a, b a c nechť jsou 100, 200, 300 $
  - ✓ integritní omezení $a+b+c=const$

Transaction *T*:	**Transaction *U*:**
balance = b.getBalance(); *b.setBalance(balance*1.1);* *a.withdraw(balance/10)*	*balance = b.getBalance();* *b.setBalance(balance*1.1);* *c.withdraw(balance/10)*

balance = b.getBalance();	$200
*b.setBalance(balance*1.1);*	$220
a.withdraw(balance/10)	$80

***a* + *b* + *c* = 80 + 220 + 300 = 600**

balance = b.getBalance();	$220
*b.setBalance(balance*1.1);*	$242
c.withdraw(balance/10)	$278

***a* + *b* + *c* = 80 + 242 + 278 = 600**

Problémy souběžnosti transakcí

- Sériové provedení transakcí T a U, plán 2
  - ✓ Iniciální stavy účtů a, b a c nechť jsou 100, 200, 300 $
  - ✓ integritní omezení $a+b+c=const$

Transaction *T*:	**Transaction *U*:**
balance = b.getBalance(); *b.setBalance(balance*1.1);* *a.withdraw(balance/10)*	*balance = b.getBalance();* *b.setBalance(balance*1.1);* *c.withdraw(balance/10)*
	balance = b.getBalance(); $200
	*b.setBalance(balance*1.1);* $220
	c.withdraw(balance/10) $280
balance = b.getBalance(); $220	
*b.setBalance(balance*1.1);* $242	
a.withdraw(balance/10) $78	
***a** + **b** + c* **= 78+ 242+** *280* **= 600**	*a* **+ *b* + *c* =** *100* **+ 220+ 280 = 600**

Problémy souběžnosti transakcí

- Problém ztracené korekce (korekce v T přepisuje korekci v U)
  - ✓ Iniciální stavy účtů a, b a c nechť jsou 100, 200, 300 $

Transaction *T*:		Transaction *U*:	
balance = b.getBalance(); *b.setBalance(balance*1.1);* *a.withdraw(balance/10)*		*balance = b.getBalance();* *b.setBalance(balance*1.1);* *c.withdraw(balance/10)*	
balance = b.getBalance();	$200		
		balance = b.getBalance();	$200
		*b.setBalance(balance*1.1);*	$220
*b.setBalance(balance*1.1);*	$220		
a.withdraw(balance/10)	$80		
		c.withdraw(balance/10)	$280

  - ✓ $a+b+c=80+220+280=580$, nekonzistence

Problémy souběžnosti transakcí

- Jiné (validní) souběžné provedení transakcí T a U, ekvivalentní jejich sériovému provedení T, U

Transaction *T*:	Transaction *U*:
balance = b.getBalance() *b.setBalance(balance*1.1)* *a.withdraw(balance/10)*	*balance = b.getBalance()* *b.setBalance(balance*1.1)* *c.withdraw(balance/10)*

balance = b.getBalance()	$200		
*b.setBalance(balance*1.1)*	$220		
		balance = b.getBalance()	$220
		*b.setBalance(balance*1.1)*	$242
a.withdraw(balance/10)	$80		
		c.withdraw(balance/10)	$278

Problémy souběžnosti transakcí

- Problém získání nekonzistentního stavu (v transakci W)
  - ✓ Iniciální stavy účtů A, B nechť jsou každý 200 $

Transaction *V*:		Transaction *W*:	
a.withdraw(100) *b.deposit(100)*		*aBranch.branchTotal()*	
a.withdraw(100);	$100		
		total = a.getBalance()	$100
		total = total+b.getBalance()	$300
b.deposit(100)	$300		

  - ✓ součet výšek účtů A a B je ale 400 $, nikoli 300 $

Problémy souběžnosti transakcí

- Sériové provedení provedení transakcí V a W je validní

Transaction *V*:		Transaction *W*:	
a.withdraw(100); *b.deposit(100)*		*aBranch.branchTotal()*	
a.withdraw(100);	$100		
b.deposit(100)	$300		
		total = a.getBalance()	$100
		total = total+b.getBalance()	$400

Testování serializovatelnosti

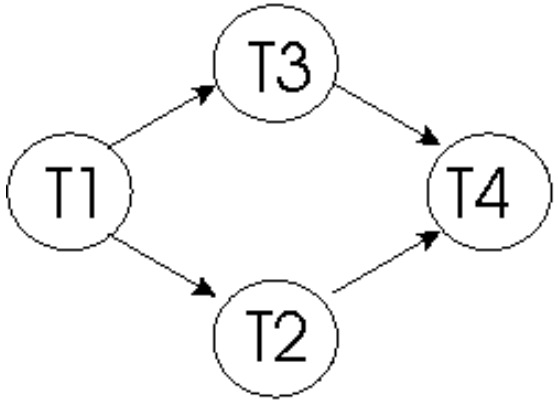
- mějme plán množiny transakcí T_1, T_2, T_3,... T_n
- tento plán je serializovatelný pokud určuje sériově ekvivalentní prokládání operací transakcí, ve kterém je jejich kombinovaný účinek shodný s kombinovaným účinkem některého jejich sériového provedení
- Shodný účinek $\equiv$ čtecí operace vrací shodné hodnoty a finálně jsou výsledné hodnoty modifikovaných objektů shodné
- Nutnou a postačující podmínkou sériově ekvivalentního prokládání provedení transakcí je, aby se všechny páry konfliktních operací v těchto transakcích provedly na všech zpřístupňovaných objektech ve stejném pořadí

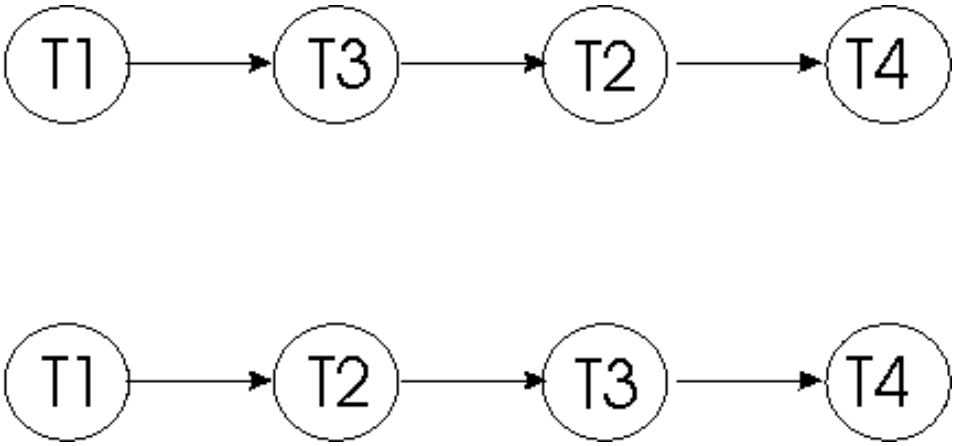
Testování serializovatelnosti

- vypracujme precendenční graf
  - ✓ uzly – transakce
  - ✓ hrana vede z T_i do T_j, jestliže jsou tyto dvě transakce v konfliktu a T_i zpřístupnila datovou položku (kvůli níž vzniká konflikt) dříve
  - ✓ hranu pojmenujeme jménem relevantní datové položky
- plán je konfliktově serializovatelný pokud je jeho precendenční graf acyklický
  - ✓ klasický algoritmus detekce cyklu v grafu má složitost n^2, kde n je počet uzlů
- je-li precedenční graf acyklický, sérializovatelnost lze ilustrovat topologickým tříděním grafu
  - ✓ určení lineárního uspořádání, které je konzistentní s částečným uspořádáním precendenčního grafu

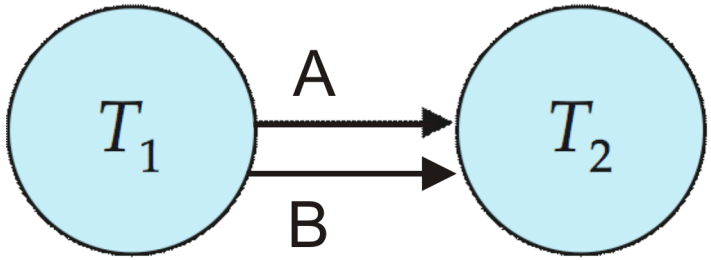
Topologické třídění

- acyklický graf a jeho dvě možná lineární uspořádání

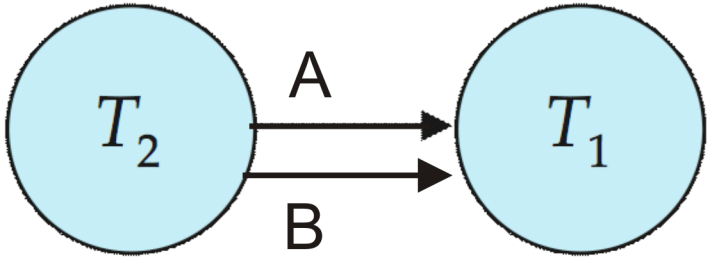




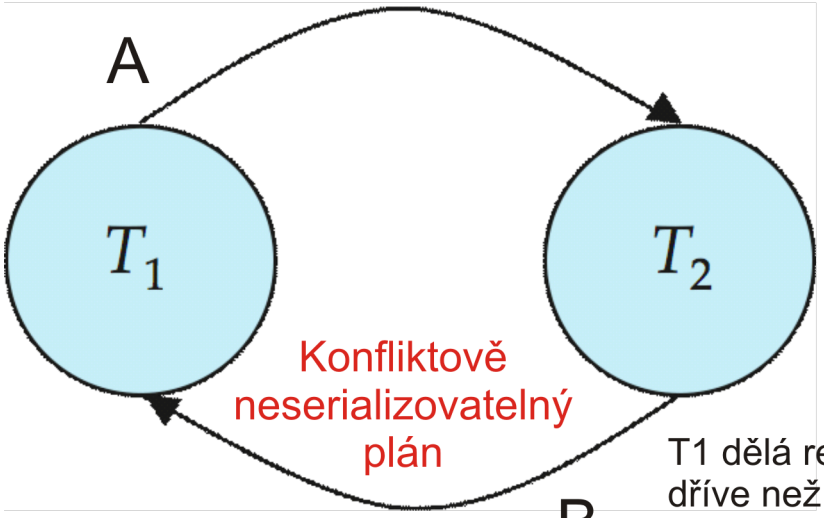
Precendeční grafy plánů P1, P2 a P4



Plán P1



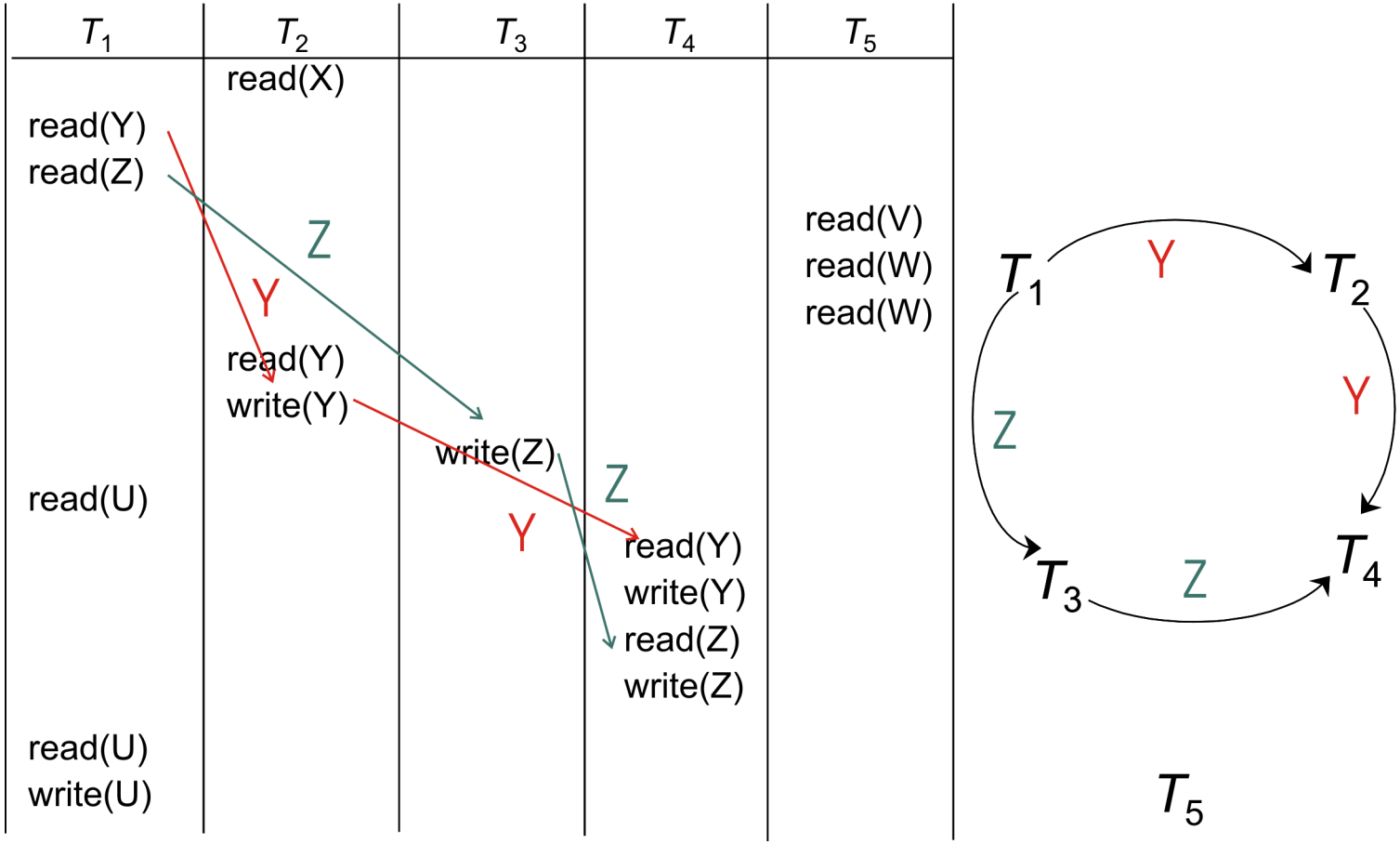
Plán P2



Plán P4

T1 dělá read(A)
dříve než T2 dělá write(A)
,
T2 dělá read(B)
dříve než T1 dělá write(B)

Příklad serializovatelného plánu



- ✓ možná serializace: $T_5 \rightarrow T_1 \rightarrow T_3 \rightarrow T_2 \rightarrow T_4$
- ✓ jiná možná serializace: $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4 \rightarrow T_5$

Vizuální vs. konfliktová ekvivalence

- ✓ dva plány mohou poskytovat stejný výsledek, ale nemusí být konfliktově ekvivalentní

T_1	T_5
read (A)	
$A := A - 50$	
write (A)	
	read (B)
	$B := B - 10$
	write (B)
read (B)	
$B := B + 50$	
write (B)	
	read (A)
	$A := A + 10$
	write (A)

Plán není konfliktově ekvivalentní se sériovým plánem <T1, T5>

write(B) v T5 je v konfliktu s read(B) v T1
tj. T5 -> T1

write(A) v T1 je v konfliktu s read(A) v T5
tj. T1->T5

Plán není serializovatelný, v precedenčním grafu je cyklus, přesto dává správný výsledek shodný se sériovým plánem <T1, T5>

- ✓ vizuální ekvivalence plánů se řeší obtížně, jde o NP-úplný problém, v praxi se vesměs aplikuje konfliktová ekvivalence

Obnova po poruše (výpadku), obnovitelný plán

- Studujeme nyní vliv poruch (výpadků) na souběžné provádění transakcí
- Hledáme omezení vymezující ze serializovatelných plánů tzv. obnovitelné plány
- Obnovitelný plán respektuje omezení zajišťující při souběžně řešených transakcích zachování obnovitelnosti (*Recoverability*) báze dat po poruše (výpadku)
  - ✓ jestliže T_i z jakéhokoliv důvodu zkrachuje, je nutné pro zachování vlastnosti atomicity při jejím zrušení eliminovat (undo) efekty způsobené jejím prováděním do doby než zkrachovala
  - ✓ to stejné platí pro každou T_j, která je závislá na T_i, tj. která čte data modifikovaná v průběhu T_i do doby jejich čtení v T_j

Problém obnovitelnosti plánu, neférové čtení, dirty read

- mějme souběžně řešené transakce

  T_1: *read*(Q), *write*(Q), *read*(P); a T_2: *read*(Q)

- nechť TPM umožní hotovou transakci T_2 prohlásit za provedenou dříve než je prohlášená za provedenou T_1

Problém obnovitelnosti plánu, neférové čtení, dirty read

- Neférové čtení způsobuje interakce mezi
  čtením v jedné transakci a
  dřívějším zápisem do stejného objektu v jiné transakci
- následující plán v tom případě není obnovitelný z důvodu

✓

T_1:	T_2:
read(Q)	
write(Q)	
	read(Q)
read(P)	
	t_commit
t_abort	

tzv. neférového čtení (*dirty read*)

nezpůsobí zkrachování i T_2

- ✓ tj. T_1 krachuje po prohlášení T_2 za provedenou a ruší se (*t_abort*)
- ✓ pro zachování atomicity se musí rovněž zrušit T_2,
  poněvadž četla data vytvořená T_1, což už ale nelze provést,
  T_2 je již provedená a provedenou transakci eliminovat (undo) nelze

Problém obnovitelnosti plánu, předčasný zápis

	Hodnota A		Hodnota A
Transaction *T*:		**Transaction *U*:**	
a.setBalance(105)		*a.setBalance(110)*	
	$100		
a.setBalance(105)	$105		
		a.setBalance(110)	$110

- ✓ Mnohé DBS implementují rušení transakce obnovou na hodnoty objektů před všemi jejími zápisy (*before image*)
- ✓ Jestliže se provede T a poté se zruší U, bude hodnota $a = 105$, OK
  jestliže se zruší T po provedení U, bude hodnota $a = 100$ a ne 110
  jestliže se zruší T po té i U, bude hodnota $a = 105$ a ne 100
- ✓ Aby výsledky obnovy byly správné, musí být zápisy v transakcích zpožděné do doby provedení nebo zrušení dřívějších transakcí, které modifikovaly stejný objekt

Obnovitelný plán

- v systému se souběžně řešenými transakcemi se obvykle (přirozeně) požaduje, aby
  každý plán byl serializovatelný A obnovitelný
- plán je obnovitelný, pokud platí:
  - ✓ pro každý pár transakcí T_i a T_j,
    ve kterém T_j čte datovou položku dříve zapsanou T_i,
    stav *t_commit* ukončující T_i nastane před *t_commit* ukončujícím T_j
  - ✓ T_j lze prohlásit za provedenou po prohlášení T_i za provedenou
- následující plán už obnovitelný je
  - ✓

T_i:	T_j:
read(Q)	
write(Q)	
	read(Q)
read(P)	
t_abort	způsobí zkrachování i T_j, ta není dosud provedená

Kaskádní vracení

- *Cascading Rollbacks*
- zkrachování jedné T může způsobit návrat více transakcí
- v následujícím plánu zkrachování T_1 vyvolá i návraty T_2 a T_3
  - ✓ nechť žádná T dosud neprovedla *t_commit*, plán je tudíž obnovitelný
-

T_1:	T_2:	T_3:
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)
t_abort		

- kasdádní vracení může představovat vysokou zátěž

Plány bez kaskádních vracení

- ke kaskádnímu vracení nemůže dojít, pokud platí:
  pro každý pár transakcí T_i a T_j,
  ve kterém T_j čte datovou položku dříve zapsanou T_i,
  se *t_commit* ukončující T_i se provede před relevantní operací *read* v T_j
- každý plán bez kaskádního vracení je rovněž obnovitelný
- je vysoce žádoucí se omezovat pouze na plány bez kaskádního vracení
- negativní důsledek prosazování plánů bez kaskádních vracení – omezení souběžnosti

Obnovitelný plán – striktní (přísné) provádění transakcí

- transakce zpožďují čtení a zápisy tak, aby zabránily neférovým čtením a předčasným zápisům
- při striktním (přísným) prováděním transakcí se čtení a zápisy jistého objektu v transakci zpožďují do doby provedení či zrušení dřívějších transakcí, které daný objekt modifikovaly
- striktní (přísné) provádění transakcí podporuje dosažení požadované vlastnosti obnovitelnosti, omezuje však stupeň souběžnosti

Řízení souběžnosti

- TPM musí poskytnout mechanismus, který zajistí,
  aby se při souběhu transakcí realizovaly plány
  - ✓ konfliktově serializovatelné a
  - ✓ obnovitelné, lépe však obnovitelné bez kaskádních vracení
- omezení na sériové plány jsou zbytečně restriktivní
- cílem protokolů pro řízení souběžnosti je zajištění,
  aby plány byly konfliktově serializovatelné a
  obnovitelné bez kaskádního vracení
  - ✓ protokoly obvykle nevycházejí z analýzy precendenčních grafů
  - ✓ protokoly obvykle prosazují pravidla, která zajistí vyhýbání se neserializovatelným plánům
  - ✓ protokoly se vzájemně liší svojí náročností na výkon procesoru a množstvím souběžnosti, které dosahují

Řízení souběžnosti

- Nutnou i postačující podmínkou sériově ekvivalentního prokládání provedení transakcí je, aby se všechny páry konfliktních operací v těchto transakcích provedly na všech zpřístupňovaných objektech ve stejném pořadí
- plán je obnovitelný, pokud platí:
  - ✓ pro každý pár transakcí T_i a T_j,
    ve kterém T_j čte datovou položku dříve zapsanou T_i,
    stav t_commit ukončující T_i nastane před t_commit ukončujícím T_j
  - ✓ T_j lze prohlásit za provedenou po prohlášení T_i za provedenou
  - ✓ při striktním (přísným) prováděním transakcí se čtení a zápisy jistého objektu v transakci zpožďují do doby provedení či zrušení dřívějších transakcí, které daný objekt modifikovaly
  - ✓ striktní (přísné) prováděn transakcí podporuje dosažení požadované vlastnosti obnovitelnosti, omezuje však stupeň souběžnosti

Metody zajišťování izolovanosti souběžných transakcí

- Zámky – zajišťují přidělení sdílené položky transakci po dobu nutnou pro zajištění serializovatelnosti
  - ✓ zámky jsou dvojího typu
    - sdílené, připouštějí násobnost operací *read* s vyloučením operací *write* do sdílené položky
    - exkluzivní – zajišťují výlučný přístup transakce k položce
- Časová razítka
  - ✓ každá transakce má jedinečné časové razítko dané dobou jejího vzniku
  - ✓ při vzniku konfliktu transakce přistupují ke sdíleným proměnným v pořadí časových razítek
  - ✓ pokud toto pořadí nelze zajistit (došlo by např. ke v čase zpětnému zápisu hodnoty položky) transakce způsobující konflikt krachuje a restartuje se v pozdějším čase, s novým časovým razítkem

Zámky, transakční protokol na bázi zamykání

- ✓ nejčastější forma implementace plánovače transakcí

- zámek, *lock*
  - ✓ zámek – blokovací prvek udržovaný ke každé datové položce
  - ✓ zámky spravuje správce zámků – proces/služba TPM
  - ✓ zámek lze zamknout a odemknout (služby TPM)
  - ✓ k datové položce má povolen přístup ta transakce, která zamkne její zámek
  - ✓ zámky jsou dvojího typu – sdílený pro *read*, exkluzitivní pro *write* a existují pravidla, kdy lze ten který typ zámku zamknout
  - ✓ transakce jsou pomocí zamykání a odemykání zámků synchronizované tak, aby jejich účinky na sdílená data byly ekvivalentní některému jejich sériovému zpracování

- často používaná synonyma
  - ✓ zamknutí zámku, získání zámku, vlastnění zámku, ...
  - ✓ odemknutí zámku, vrácení zámku, uvolnění zámku, ...

Zámky, transakční protokol na bázi zamykání

- transakční protokol na bázi zamykání
  - ✓ množina pravidel vymezující chování transakcí při zamykání a odemykání zámků
  - ✓ omezuje množinu možných plánů na konfliktově serializovatelné plány
  - ✓ zaručuje konfliktovou serializovatelnost, pokud všechny plány, které determinuje, jsou konfliktově serializovatelné
- Typy zámků, režimy přístupu k datům
  - ✓ Sdílený zámek – (režim $shared, lock - S$)

    pokud T_i vlastní $lock\text{-}S$ položky Q,
    může obsah položky Q číst, ale nesmí do položky Q zapisovat
  - ✓ Exkluzivitní zámek – (režim $exclusive, lock - X$)

    pokud T_i vlastní $lock\text{-}X$ položky Q, je jedinou transakcí, která může obsah položky Q číst a / nebo do položky Q zapisovat

Zámky, transakční protokol na bázi zamykání

- De facto se jedná o úlohu čtenáři-písaři
  - ✓ žádá-li T_i zajistit pomocí získání zámku *lock-X* exkluzivní přístup k Q a jiná transakce už získala jakýkoliv zámek položky Q, T_i musí čekat na uvolnění tohoto zámku
  - ✓ žádá-li T_i provést pomocí získání zámku *lock-S* sdílený přístup ke Q a jiná transakce už získala zámek *lock-X* položky Q, T_i musí čekat na uvolnění tohoto zámku *lock-X*
  - ✓ žádá-li T_i provést pomocí získání zámku *lock-S* sdílený přístup ke Q a zámek *lock-S* položky Q už získala jiná transakce, T_i získá zámek *lock-S* též
- Když se transakce stane provedená či zkrachovalá, server (TPM) odemkne transakcí explicitně neodemčené zámky

Transakční zamykací protokol

- Zamykání položek pouze na dobu individuálních přístupů
  k položkám dat nezajistí
  serializovatelnost souběhu transakcí

- Potřebujeme protokol řízení souběžnosti transakcí
  povolující kdy transakce může datovou položku
  zamknout a odemknout tak,
  aby se dosáhlo serializovatelnosti souběhu transakcí

Transakční zamykací protokol

- Transakční zamykací protokol omezí počet možných plánů na serializovatelné plány
- Množina těchto plánů je podmnožinou všech možných plánů
- Nás zajímají pouze ty transakční zamykací protokoly, které povolují pouze konfliktově serializovatelné plány, které
  - ✓ jsou ekvivalentní některému z jejich seriových plánů
  - ✓ vzájemně se se liší pouze pořadím provedení nekonfliktních operací

    platí, že operace jsou konfliktní, pokud operují se stejnou položkou a alespoň jedna z nich je *write*

Dvoufázový transakční protokol na bázi zamykání

- *Two-Phase Locking Protocol*, 2PLP
- Jeden z protokolů, který zajišťuje konfliktovou serializovatelnost
  - ✓ Pokud není známá žádná další informace o způsobu zpracování dat je 2PLP nutný a postačující protokol pro zajištění konfliktové serializovatelnosti
- Princip 2PLP – jakmile transakce odemkne některý zámek, nesmí už zamknout žádný další zámek
- Transakce proto vydává zamykací a odemykací požadavky ve dvou fázích,
  v zamykací (růstové) fázi a
  v odemykací (couvací) fázi

Dvoufázový transakční protokol na bázi zamykání

- růstová fáze
  - ✓ získávání zámků, serializování transakcí podle pravidel:

    a) pokud objekt není zamknutý, zamkne se a transakce pokračuje
    b) pokud je objekt uzamknutý konfliktním zámkem jinou transakcí, transakce čeká na jeho odemknutí
    c) pokud je objekt uzamknutý nekonfliktním zámkem jinou transakcí, objekt se sdílí, transakce získá zámek a pokračuje
    d) pokud je objekt již zamknutý stejnou transakcí, zamknutí se uplatní a transakce pokračuje. (Pokud uplatnění brání konfliktní zámek, použije se pravidlo b)
- couvací fáze
  - ✓ uvolňování zámků – lze provádět POUZE PO posledním získání zámku
- Uváznutí 2PLP neřeší – zamezení uváznutí lze dosáhnout např. vnucením pevného řazení zpřístupňovaných dat

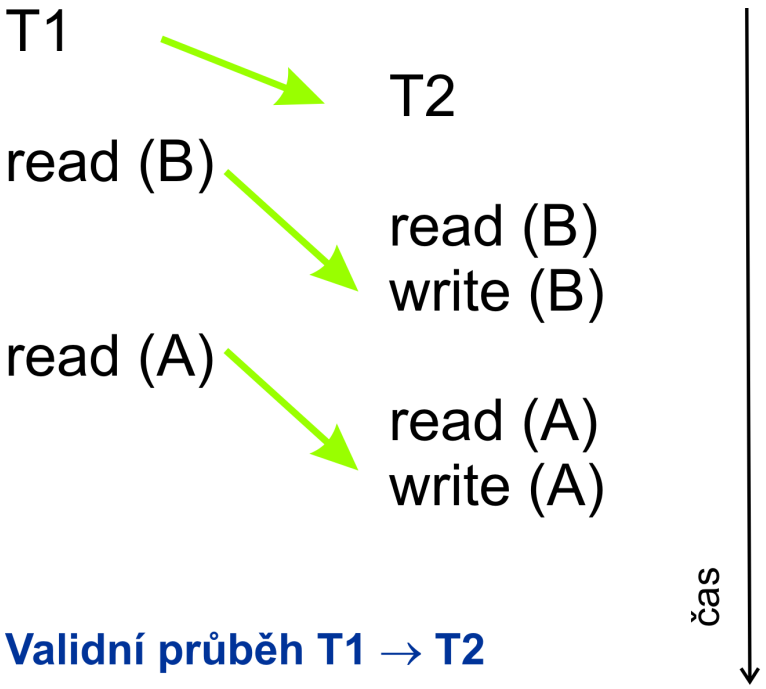
Protokol řízení souběhu transakcí s časovými razítky

- Řízení souběžnosti transakcí založené na jejich časovém řazení
  - ✓ TPM zaznamenává pro každý sdílený objekt dobu posledního čtení a zápisu při každé operaci s objektem
  - ✓ na bázi časových razítek transakcí daných okamžikem jejich vzniku rozhoduje, zda
    - lze operaci provést bezprostředně,
    - lze operaci provést později (transakce bude čekat) nebo
    - se musí operace odmítnout (transakce krachuje, klient ji může restartovat)
  - ✓ požadavek transakce na zápis je validní pouze když objekt byl naposledy čtený nebo zapisovaný dřívější transakcí nebo nebyl dosud zpřístupněný
  - ✓ požadavek transakce na čtení je validní pouze když objekt byl naposledy zapisovaný dřívější transakcí nebo nebyl dosud zpřístupněný

Protokol řízení souběhu transakcí s časovými razítky

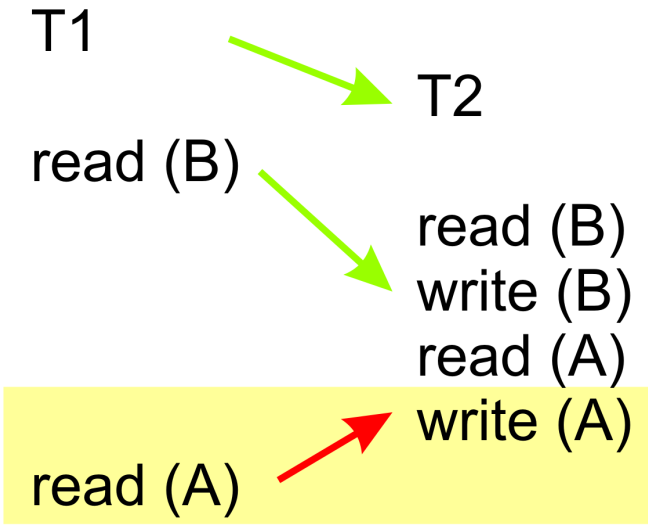
- Při startu transakce T_i se T_i přiděluje časové razítko, TS_i
  - ✓ TS_i je jedinečné, v jednom okamžiku dochází k jediné události
  - ✓ TS_i definuje pozici transakce T_i na časové ose
- $TS_i < TS_j$ – transakce T_i předchází transakci T_j, je starší
- $TS_i > TS_j$ – transakce T_i následuje po transakci T_j, je mladší
- Požadavky transakcí se totálně řadí podle příslušných časových razítek transakcí
  - ✓ požadavek transakce na *write* objektu je validní pouze když objekt byl naposled čtený / zapisovaný dřívější (starší) transakcí a nebo nebyl dosud zpřístupněný vůbec
  - ✓ požadavek transakce na *read* objektu je validní pouze když objekt byl naposled zapisovaný dřívější (starší) transakcí a nebo nebyl dosud modifikovaný

Protokol řízení souběhu transakcí s časovými razítky



Validní průběh T1 → T2

T1 i T2 čtou nemodifikované objekty A a B
Mladší T2 zapisuje objekty A i B až poté, co starší T1 objekty A i B přečetla



Nepřípustný průběh T1 → T2

Starší T1 čte objekt A až poté, co ho mladší T2 modifikovala.

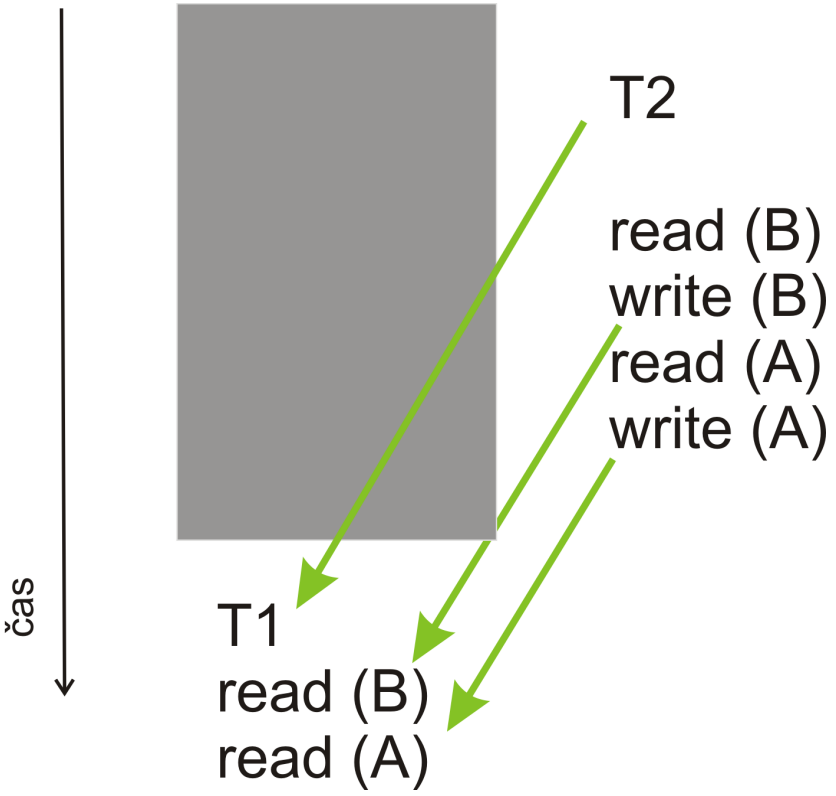
Protokol řízení souběhu transakcí s časovými razítky

Validní průběh T1 a T2

T1 byla zrušena a obnovena v novém čase, takže T2 → T1 a platí:

Starší T2 čte nemodifikované objekty A i B

Mladší T1 čte objekty A i B poté, co starší T2 objekty A i B zapsala



Distribuovaná transakce, koncept

- Distribuovaná transakce zahrnuje operace prováděné ve více serverech (uzlech DS) formou dílčích subtransakcí
- Uzly DS mohou vzájemně komunikovat, obecně – každý s každým
- Musí být zachován bázový princip transakce – atomicita, A
  - ✓ Buďto se provedou všechny operace určené programovou jednotkou řídicí transakci nebo se neprovede žádná z nich
- Musí být zachovány pochopitelně i ostatní vlastnosti transakcí – CID
- Subtransakce realizované v různých uzlech DS proto na ukončování své transakce kooperují
  - ✓ pomocí koordinačních procesů realizovaných na úrovni middleware (TPM)

Distribuovaná transakce, koncept

- Použitá technologie – model klient-server
- Klient – aplikační proces běžící v některém uzlu DS
  - ✓ aktivuje realizaci transakce T žádostí zaslanou na koordinátora této transakce T (server běžící v některém uzlu DS),
  - ✓ řeší program definující transakci spočívající ve volání posloupností operací řešících transakci (subtransakcí) vesměs rozmístěných v různých uzlech DS např. podle jimi spravovaných objektů apod.
  - ✓ aktivuje krach nebo prohlášení hotové transakce T za provedenou žádostí zaslanou na koordinátora jím spuštěné transakce T
- Server – systémový proces běžící v některém uzlu DS
  - ✓ koordinátor transakce T, je součástí TPM, middleware
  - ✓ zajišťuje start a ukončení distribuované transakce s cílem zajistit atomicitu transakce aktivované klientem

Distribuovaná transakce, koncept

- Průběhy operací subtransakcí v jednotlivých uzlech DS řídí správci , funkčnosti na úrovni TPM
  - ✓ Subtransakce náležející transakci T jsou spouštěné klientem, v uzlech DS jsou aktivované dynamicky voláním metod z klienta
  - ✓ správci transakcí
    - jsou procesy běžící v uzlech DS, jsou součástí TPM, middleware
    - řídí validnost souběžnosti (sub)transakcí
    - udržují deník pro účel obnovy po výpadku
    - spolupracují s koordinátorem transakce na rozhodnutí zda se transakce prohlásí za provedenou nebo zda krachuje
  - ✓ algoritmy obnovy i řízení souběžnosti se musí respektovat distribuované prostředí

Distribuovaná transakce, koncept

- Klient zahajující transakci zasílá požadavek openTransaction (n.b. t_begin) koordinátorovi transakcí řídícímu chod zahajované transakce, C_i (služba v TPM v určeném uzlu i)
- Koordinátor transakcí C_i klientovi vrací id transakce, T – jednoznačný identifikátor transakce v DS (např. IP adresa i + pořadové číslo transakce v i) a přebírá roli koordinátora transakce T, který openTransaction pro transakci T provede
- Proces klienta a procesy, řešící požadavky klienta jako části distribuované transakce, musí být schopní s koordinátorem transakce komunikovat, obv. via RPC, aby mohly koordinovat svoje akce při přechodu transakce do stavu provedená (*commit*) nebo zrušená (*abort*),

Distribuovaná transakce, koncept

- Koordinátor transakce T registruje existenci transakce T a posléze i koordinuje její ukončení (*commit* / *abort*)
- Uzly zúčastněné na řešení transakce T tvoří tzv. „kohortu T", nazýváme je participující uzly na transakci T
- části transakce realizované v participujících uzlech, subtransakce, se obvykle aktivují voláním metod klientem
- v participujícím uzlu transakce T se zřizuje objekt participant, který je odpovědný za sledování stavu obnovitelných objektů v daném serveru zpřístupňovaných transakcí T a kooperuje s koordinátorem při ukončování transakce T
- participanti se registrují u koordinátorů svých transakcí operací join

Distribuovaná transakce, koncept

- koordinátor T si udržuje seznam participantů transakce T
- participanti (subtransakce) si relevantní službou svých TPM udržují referenci na svého koordinátora a komunikují s ním při ukončování transakce, *commit* / *abort*
- účast participanta na transakci:
  - registrace u koordinátora +
  - správa objektů zpřístupňovaných subtransakcí +
  - kooperace s koordinátorem při ukončování transakce
- buďto všichni participanti své subtransakce provedou (a potvrdí) – transakce se stane provedenou, *commit*
- nebo všichni participanti zruší akce provedené v rámci subtransakcí – transakce zkrachuje (*abort*)

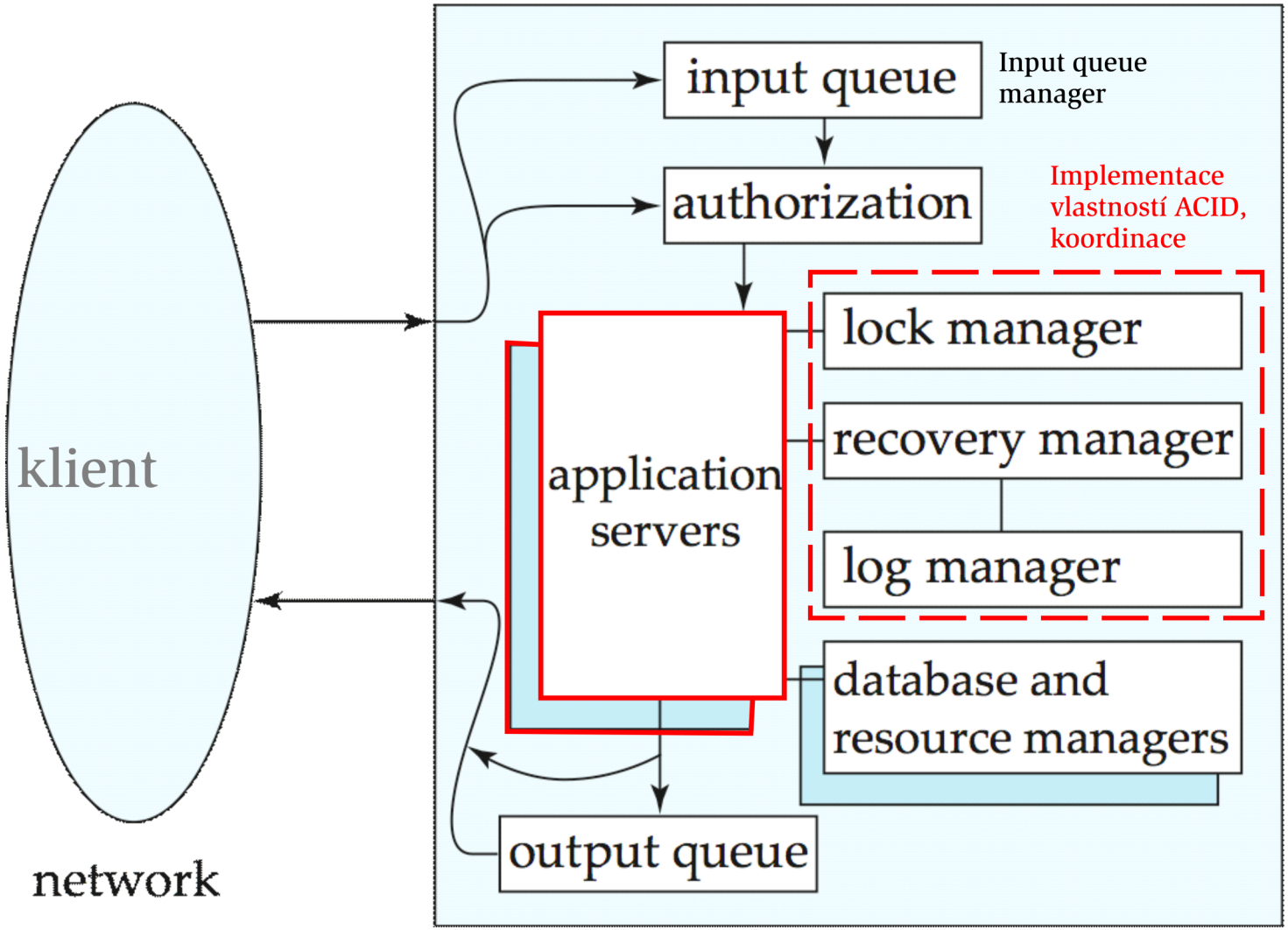
Distribuovaná transakce, koncept

- Koordinátor transakce
  - registruje spuštění transakce,
  - registruje participanty,
    příp. sám dělí transakci na subtransakce a
  - řeší, koordinuje akce při ukončování transakce

- Správce transakce v uzlu kohorty, participant
  - udržuje deník pro obnovu transakcí
  - participuje na schématu řízení souběžnosti

- Každý participant může kdykoliv volat koordinátora požadavkem abortTransaction, pokud z nějakého důvodu není schopný jeho uzel v řešení subtransakce pokračovat

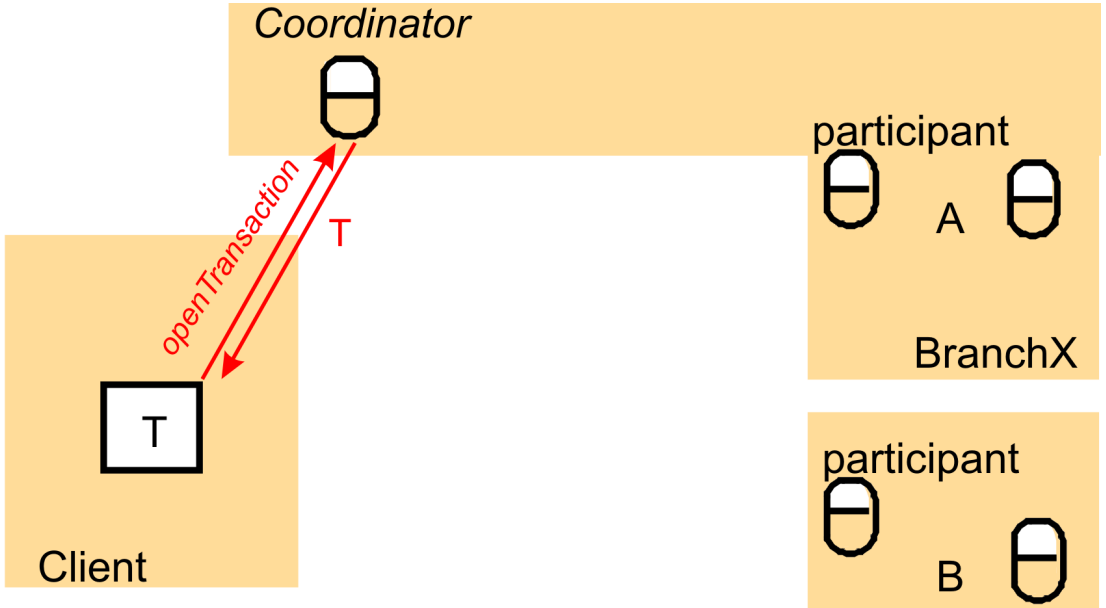
Konceptuální schéma funkčních komponent TPM



Příklad, distribuovaná bankovní transakce

- Klientova transakce zahrnuje práci s účty A,B, C a D
  - ✓ účet A spravuje server v pobočce X
  - ✓ účet B spravuje server v pobočce Y
  - ✓ účty C a D spravuje server v pobočce Z
  - ✓ transakce
    - přenáší 4 USD z účtu A do do účtu C a
    - přenáší 3 USD z účtu B do účtu D
- Koordinátor transakce může byt situovaný v kterémkoliv serveru
  - ✓ např. v severu pobočky X, která je např. umístěna v centrále banky

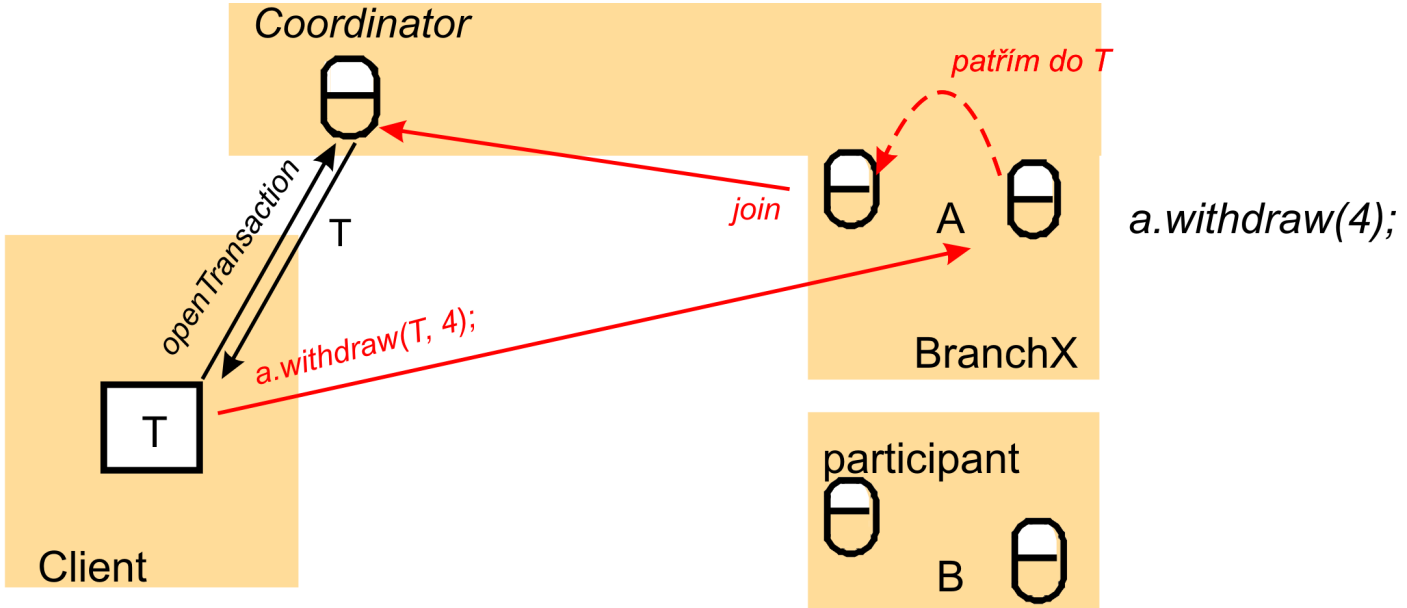
Příklad, distribuovaná bankovní transakce



T = openTransaction
a.withdraw(4);
c.deposit(4);
b.withdraw(3);
d.deposit(3);
closeTransaction

Note: the coordinator is in one of the servers, e.g. BranchX

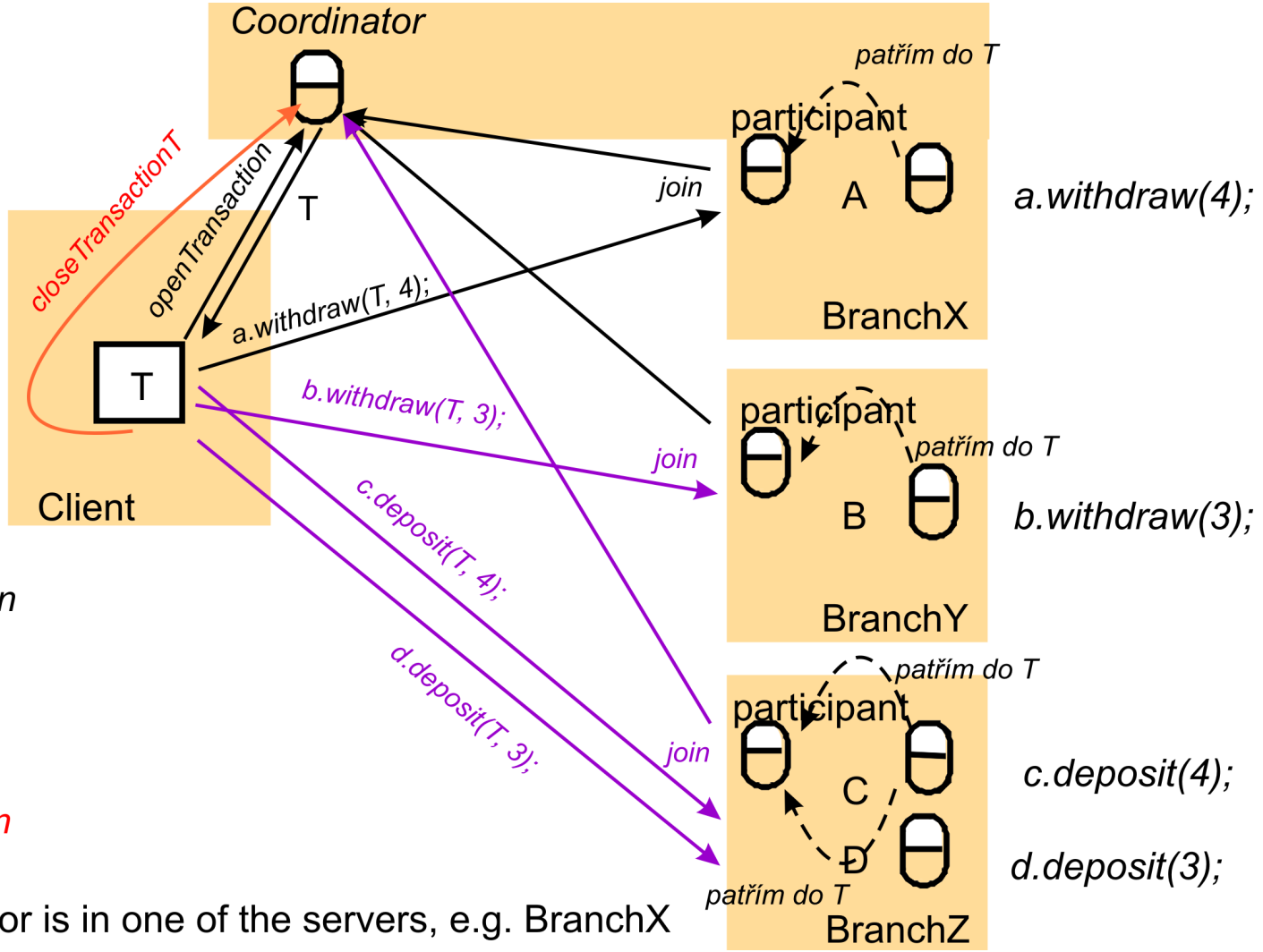
Příklad, distribuovaná bankovní transakce



T = openTransaction
a.withdraw(4);
c.deposit(4);
b.withdraw(3);
d.deposit(3);
closeTransaction

Note: the coordinator is in one of the servers, e.g. BranchX

Příklad, distribuovaná bankovní transakce



T = openTransaction
a.withdraw(4);
c.deposit(4);
b.withdraw(3);
d.deposit(3);
closeTransaction

Note: the coordinator is in one of the servers, e.g. BranchX

Příklad, distribuovaná bankovní transakce

- klient otevře transakci a získá její identifikátor, T
- Posléze klient vyvolává metody předepsané programem transakce realizované v rámci otevřené transakce např. *b.withdraw (T,3)* v serveru Y
- Invokovaný objekt *b* v serveru Y informuje objekt *participant* v tomto serveru o tom, že náleží transakci T
- pokud tak objekt *participant* v serveru Y dosud neučinil, informuje objekt *participant* koordinátora, že náleží transakci T
- po aplikačním dokončení transakce, klient nakonec informuje koordinátora transakce o konci transakce požadavkem closeTransaction

Řešení problému ukončení transakce – Commit protocols

- Problém – zajištění atomicity (distribuované) transakce
- Klient požádal koordinátora o otevření transakce
- Poté klient postupně vyžádal provádění subtransakcí na více serverech
- Nakonec klient žádá koordinátora o closeTransaction, ukončení transakce formou *commit* nebo *abort*
  - ✓ *abort* – např. se mu nepodařilo se spojit s některým uzlem nebo zjistil narušení konzistence DB, . . .
- Atomické ukončení distribuované transakce se musí řídit jistými pravidly, protokolem
- Jedná se o převedení hotové transakce na provedenou transakci – *commit*, proto commit protocol

Jednoduchý, 1-fázový, commit protokol

- O typu ukončení (provedení / krach) transakce rozhoduje klient, který transakci vyvolal
- Koordinátor postupně (opakovaně) sděluje participantům podle požadavku klienta zda mají provést *commit* nebo *abort*,

  a to opakovaně dokud mu každý participant nesdělí, že u sebe provedl *commit* nebo *abort* jím řešené subtransakce
- Pro řešení ukončení distribuované transakce je jednoduchý, 1-fázový, commit protokol nedostatečný
  - ✓ Důvody – viz dále

Jednoduchý, 1-fázový, commit protokol

- Pro řešení ukončení distribuované transakce je jednoduchý, 1-fázový, commit protokol nedostatečný
  - ✓ Když klient rozhodne ukončit transakci řádně, *commit*, nedává tím možnost žádnému participantu či koordinátorovi aby on jednostranně rozhodl o zkrachování transakce
  - ✓ Např. díky zamykání může mezi servery vzniknout uváznutí, které vede ke zkrachování transakce, ale o tom se klient nedozví, dokud nevydá další požadavek na server, dokud mu neuplyne time-out, . . .
  - ✓ Např. koordinátor nemusí vědět, že jistý server měl během řešení transakce výpadek a obnovoval svoji činnost v průběhu distribuované transakce a koordinátor by měl tudíž transakci abortovat

2-fázový commit protokol

- Jedná se o zvláštní případ problému Byzantských generálů
  - ✓ distribuovaná dohoda subtransakcí zda transakci končit řádně či krachovat v prostředí s výpadky uzlů
  - ✓ Lokální atomicita subtransakce nestačí, cílem je globální atomicita, ta nemůže být zaručena implicitně.
  - ✓ Globální synchronizace distribuované transakce může skončit v některých uzlech provedením subtransakce a v jiných krachem
  - ✓ To ohrožuje globální atomicitu a tudíž konzistenci DBS
  - ✓ Dosažení atomicity na globální úrovni vyžaduje použít synchronizační protokol, který zajistí jednoznačný konečný výsledek pro každou distribuovanou transakci bez ohledu na výpadky uzlů.

2-fázový commit protokol

- Two-Phase Commit Protocol, (2PCP nebo také 2PCP)
  - ✓ synchronizační protokol, který řeší problém atomického ukončení
  - ✓ umožňuje každému participantu jednostranně abortovat svoji část transakce
  - ✓ jestli jedna subtransakce krachuje, krachuje celá transakce
  - ✓ jestliže se řádně ukončí všechny subtransakce, řádně se ukončuje celá transakce
- Ukončení transakce pomocí 2PCP koordinuje koordinátor, výsledkem rozhodnutí koordinátora může být
  - ✓ *commit* – potvrzení provedení subtransakcí ve všech uzlech s_i a převedení transakce do stavu provedná
  - ✓ *abort* – zrušení účinků subtransakcí ve všech uzlech s_i, pokud alespoň jedna subtransakce zkrachovala, a zrušení transakce

2-fázový commit protokol

- 2PCP zajišťuje, že
  - ✓ distribuovaná transakce se buď řádně ukončí
    a všechny její účinky se stanou trvalé ve všech participujících uzlech
  - ✓ nebo distribuovaná transakce zkrachuje
    a všechny její účinky ve všech uzlech se zruší,
    jako by se transakce nikdy nevykonala.
- Spuštění 2PCP
  - ✓ Každá transakce má v některém uzlu DS koordinátora.
  - ✓ Poté, co transakce z hlediska klienta dokončí všechny aplikační operace, klient požádá koordinátora o ukončení transakce
  - ✓ koordinátor spouští 2PCP.
  - ✓ detaily běhu 2PCP viz dále

Řešení ukončení transakce, Two-Phase Commit Protocol

- 2PCP se skládá ze dvou fází,
  jmenovitě z hlasovací fáze a z fáze vydání rozhodnutí.
- Hlasovací fáze
  - ✓ Koordinátor žádá všechny participanty transakce o závazek,
    že subtransakci řádně ukončí, když ve fázi vydání rozhodnutí
    k tomu dostanou pokyn – hlasují ano / ne pro ukončení transakce
  - ✓ Jakmile participant obdrží výzvu k hlasování,
    ověří na něj navázanou subtransakci z pohledu konzistence dat.
  - ✓ Pokud lze subtransakci řádně ukončit (tj. prošla validace), hlasuje ano.
  - ✓ V opačném případě hlasuje ne,
    subtransakci následně bez dalšího čekání krachuje,
    ruší subtransakcí způsobené účinky a
    uvolní se všechny subtransakcí držené zdroje (zámky, . . .).

Řešení ukončení transakce, Two-Phase Commit Protocol

- Ve fázi vydání rozhodnutí koordinátor transakce rozhoduje
  - ✓ zda řádně ukončit transakci, pokud jsou všichni participanti připraveni řádně ukončit transakci (hlasovali ano)
  - ✓ nebo zda zkrachovat, pokud se kterýkoliv participant rozhodl transakci zkrachovat (hlasoval ne).
- V případě vydání rozhodnutí řádně ukončit transakci, koordinátor vysílá všem participantům zprávy *commit* (řádně ukončit), transakce se stane provedenou
- V případě vydání rozhodnutí zkrachovat transakci koordinátor odesílá zprávy *abort* (zkrachovat), a to pouze těm participantům, kteří jsou připraveni řádně ukončit transakci (hlasovali ano), transakce se stane zkrachovalou

Řešení ukončení transakce, Two-Phase Commit Protocol

- Jestliže participant hlasoval ano, nemůže subtransakci jednostranně ani řádně ukončit ani zkrachovat, musí čekat, dokud od koordinátora neobdrží konečné rozhodnutí.
- Participant je tudíž po neurčitou dobu blokovaný (okno nejistoty), čeká na rozhodnutí koordinátora.
- Jakmile participant obdrží finální rozhodnutí koordinátora, rozhodnutí respektuje a vykoná odpovídající akce a uvolní všechny zdroje, které subtransakce drží, a obvykle odešle zpět koordinátorovi potvrzení (ACK)
- Zámky vyžádané transakcí si transakce musí držet až do svého ukončení

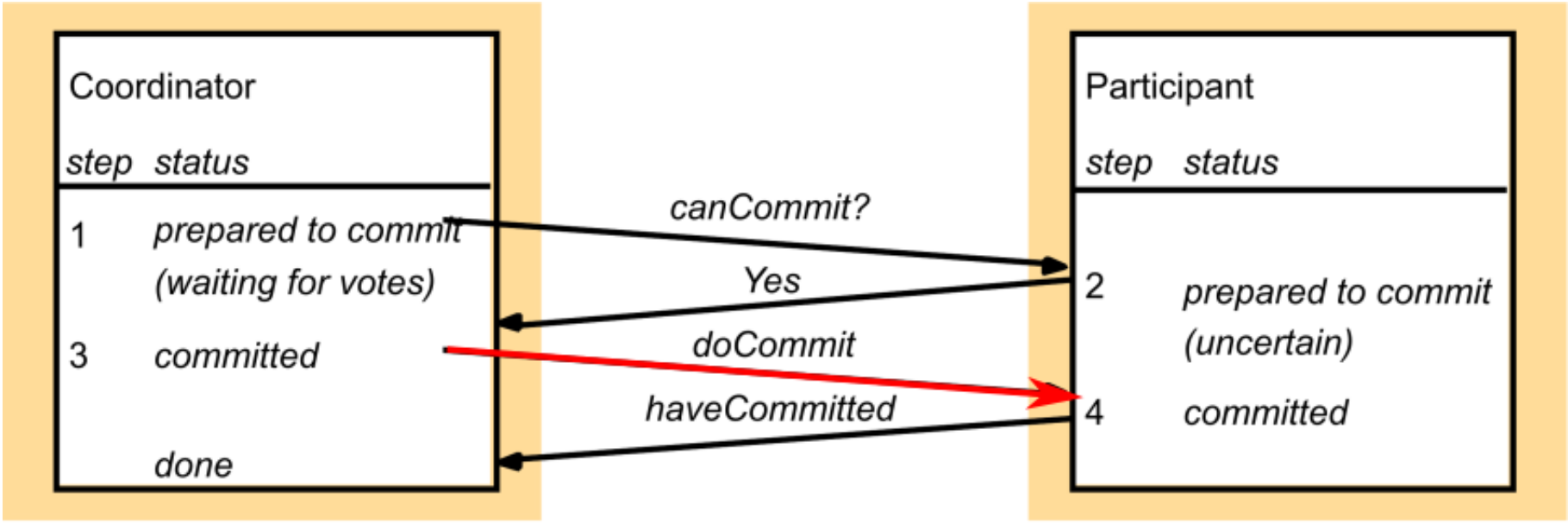
Řešení ukončení transakce, Two-Phase Commit Protocol

- Jakmile koordinátor obdrží potvrzení od všech participantů, pak v případě, že hlasovali ano, zapomene na transakci, vymaže veškeré informace týkající se transakce z protokolové tabulky udržované v hlavní paměti.
- Odolnost 2PCP vůči výpadkům se dosahuje zaznamenáváním průběhu protokolu do deníků udržovaných jak koordinátorem, tak i participanty.
- Deníky se udržují ve stabilních, energeticky nezávislých pamětech, výpadky je nezničí.
- Koordinátor zapisuje do deníku přímo (bez vyrovnávání toku dat na úrovni OS), ještě před odesláním svého rozhodnutí participantům.

Řešení ukončení transakce, Two-Phase Commit Protocol

- Stejně tak každý participant zapisuje do svého deníku
  záznam *prepared* před zasláním hlasu ano a
  záznam *decision* před zasláním potvrzení rozhodnutí.
- Když koordinátor protokol ukončuje, zapisuje už běžným
  (vyrovnávaným) zápisem do deníku záznam *end*
- Tento záznam indikuje, že všichni participanti obdrželi
  rozhodnutí, a že žádný z nich se v budoucnosti nebude
  dotazovat na stav transakce.
- Po té koordinátor může na transakci z pohledu 2PCP (trvale)
  zapomenout

Stavový diagram 2PCP



1 koordinátor

n participantů

koordinátor zasílá
všem participantům
když mu všichni participanti
odpoví Yes

Two-Phase Commit Protocol, rekapitulace

- První fáze – hlasovací fáze
  - ✓ koordinátor transakce vyzve participanty ke sdělení zda transakci řádně ukončit nebo krachovat
  - ✓ každý participant sděluje zda transakci ukončit nebo krachovat
  - ✓ jakmile participant řekne ukončit, nesmí posléze říci krachovat
  - ✓ tj. když participant řekne ukončit, musí si být jistý, že bude schopný dokončit svoji část *commit* protokolu, i když mezi tím vypadne a obnoví svoji činnost, deník musí mít zapsaný na disku
  - ✓ jakmile participant má všechny svoje objekty změněné v průběhu transakce uchované v permanentní paměti, je připravený kdykoliv posléze potvrdit řádné ukončení transakce
  - ✓ v permamentní paměti musí mít poznačený i stav připravený

Two-Phase Commit Protocol, rekapitulace

- Druhá fáze – vydání rozhodnutí o hlasování
  - ✓ koordinátor sdělí participantům rozhodnutí, výsledek hlasování
  - ✓ každý participant realizuje rozhodnutí
  - ✓ jestliže alespoň jeden participant hlasoval *abort*, krachovat, rozhodnutí zní zkrachovat transakci
  - ✓ jestliže všichni participanti hlasovali *commit*, ukončit, rozhodnutí zní ukončit transakci – *commit*
- Problémy
  - ✓ musí se zajistit, aby hlasovali všichni participanti
  - ✓ musí se zajistit, aby všichni realizovali společné finální rozhodnutí
  - ✓ triviální řešení těchto problému je v prostředí bez poruch
  - ✓ 2PCP ale musí správně pracovat i v případech výpadků serverů, ztrát zpráv nebo dočasných výpadků schopnosti komunikace mezi servery

Model poruch pro Two-Phase Commit Protocol

- Reálný distribuovaný systém =
  asynchronní distribuovaný systém
- Servery mohou vypadávat, zprávy se mohou ztrácet
- Podpůrný komunikační systém odstraňuje porušené a duplikované zprávy
- Nejedná se o byzantinské chyby
  - ✓ Server buďto vypadne nebo se řídí zaslanými zprávami
- Připomenutí – byzantinská chyba
  - ✓ proces může nastavit chybný obsah zprávy nebo na žádost vracet chybnou hodnotu, může duplikovat odpověď, . . .
  - ✓ byzantinskou chybu nelze detekovat pozorováním, zda proces odpověděl na invokaci, proces může odpověď vynechat

Model poruch pro Two-Phase Commit Protocol

- Two-Phase Commit Protocol je příklad protokolu pro dosažení dohody
  - ✓ Obecně platí –

    dosažení dohody není v plně asynchronních systémech možné
- Two-Phase Commit Protocol v asynchronním prostředí dohody dosahuje za omezující podmínky:
  - ✓ výpadky procesů (serverů) jsou maskované obnovou vypadlých procesů novými procesy, jejichž stav je nastavený podle informací uchovávaných v permanentní paměti a informací držených jinými procesy
  - ✓ proces možná po jistou dobu nereaguje, ale když reaguje, reaguje validně

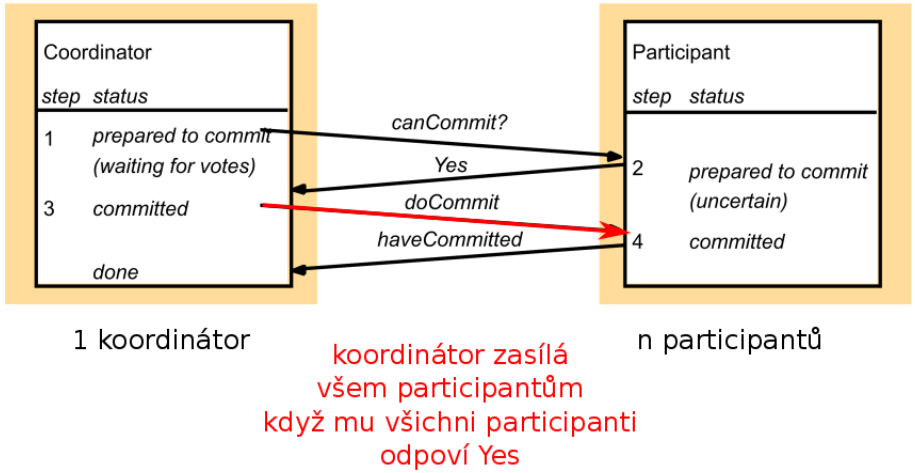
Model poruch pro Two-Phase Commit Protocol

- v každém uzlu/participantu musí správce transakcí musí udržovat deník, *write-ahead log*
  - ✓ aby bylo možné při restartu uzlu po jeho výpadku účinky subtransakce zrušit (viz přednáška Transakce):
    - objekt se modifikuje pouze po zaznamenání *undo* info do logu
    - před potvrzením změn ze zaznamenají *redo* a *undo* logy do stabilní paměti
- Přípomínka – protokol 2PCP je navržený
  - ✓ s cílem umožnit participantu zkrachovat, zrušit svoji část transakce (z důvodu zachování atomicity krachuje celá transakce)
  - ✓ pro práci v asynchronním (distribuovaném) systému, ve kterém mohou vypadávat uzly (servery) a ztrácet se zprávy
    - se nejedná o byzantinské (libovolné) chyby, server buď vypadne nebo se řídí vyslanými zprávami
    - od podpůrného systému *request-reply* se očekává, že odstraní všechny porušené, příp. duplikované zprávy

Komunikace v 2PCP

- Koordinátor v průběhu transakce, až na sdělení participanta o připojení k transakci (*join*), s participanty nekomunikuje
- Požadavek na *commit* nebo *abort* transakce předává koordinátorovi klient
- Pokud klient žádá při ukončování transakce abortTransaction nebo když některý participant hlásí *abort*, koordinátor informuje participanty okamžitě
- Víme již, že 2PCP nastupuje do své role v okamžiku, kdy klient požaduje commitTransaction (ukončit transakci)
  - ✓ V 1. fázi koordinátor žádá všechny participanty, aby řekli, zda jsou připraveni na *commit*
  - ✓ ve 2. fázi koordinátor participantům říká, ať udělají akce odpovídající *commit* nebo *abort*

Komunikace v 2PCP



- Složitost při N členech kohorty
  - ✓ 3N zpráv, ve 3 časových rundách
  - ✓ haveCommited (ACK) se nepočítá, 2PCP funguje i bez ní
    zpráva slouží k indikaci možnosti odstranit zastaralé informace
- Stav neurčitosti v participantu = čekání na finální rozhodnutí

Výpadky při řešení 2PCP

- Komunikace mezi koordinátorem a participanty může selhat, důvody
  - některý server vypadne
  - nebo komunikační systém ztratí zprávu
- Řešitelnost dosažení dohody se podporuje sledování časových výpadků (time-outs) – systém je pseudoasynchronní
  - ✓ Po časovém výpadku musí proces čekající na událost provést relevantní akci
  - ✓ Každý krok, ve kterém se čeká na událost, musí být ošetřený hlídáním časového limitu
  - ✓ V asynchronním systému uplynutí časového limitu nemusí implikovat trvalou poruchu serveru

Řešení ukončení transakce, Two-Phase Commit Protocol

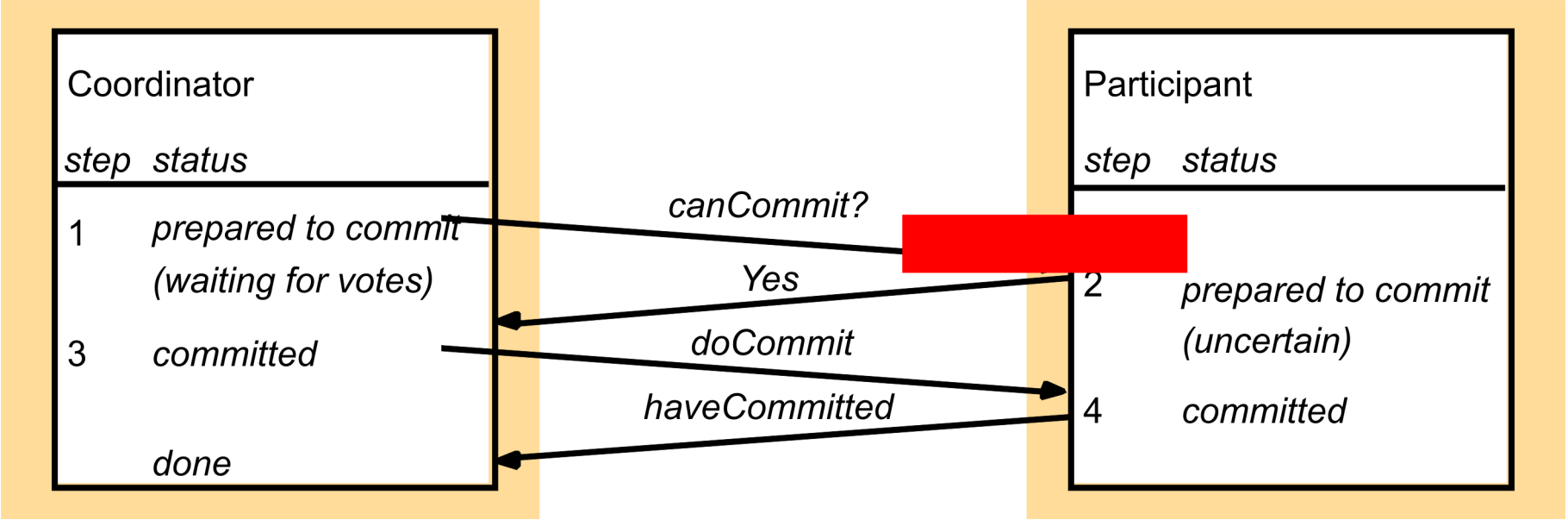
- Zotavení ve 2PCP
  - ✓ Výpadky uzlů a komunikací jsou detekované timeouty, hlídáním uplynutí časových limitů.
  - ✓ Když činný uzel detekuje výpadek evokuje Recovery Manager, jehož úkolem je výpadek zvládnout.
  - ✓ V 2PCP může selhat uzel s koordinátorem nebo uzel participanta.
  - ✓ V 2PCP jsou čtyři místa, kde může dojít k selhání komunikace.

V 2PCP jsou čtyři místa, kde může dojít k selhání komunikace

- Participant čeká na zprávu s výzvou k hlasování.

  Participant ještě nehlasoval.

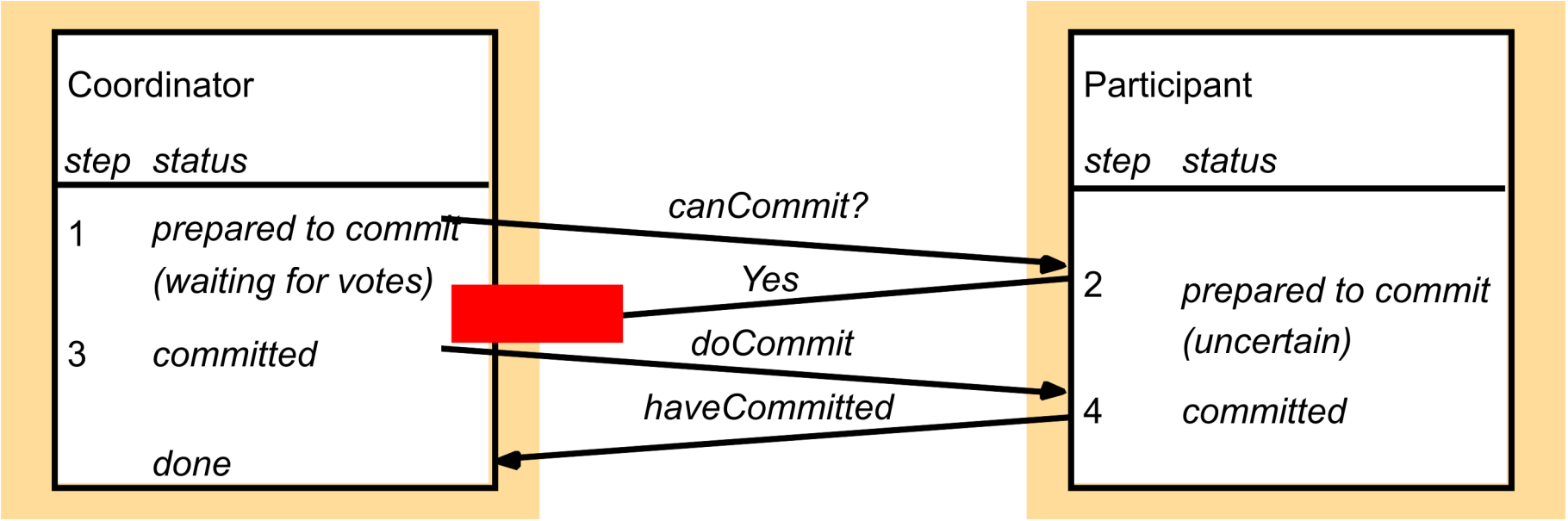
  V tomto případě participant může rozhodnout o zkrachování subtransakce jednostranně.



V 2PCP jsou čtyři místa, kde může dojít k selhání komunikace

- Koordinátor čeká na hlasování participantů.

  Vzhledem k tomu, že koordinátor dosud neučinil konečné rozhodnutí, žádný participant se nemohl řádně ukončit, kordinátor může rozhodnout zkrachovat.

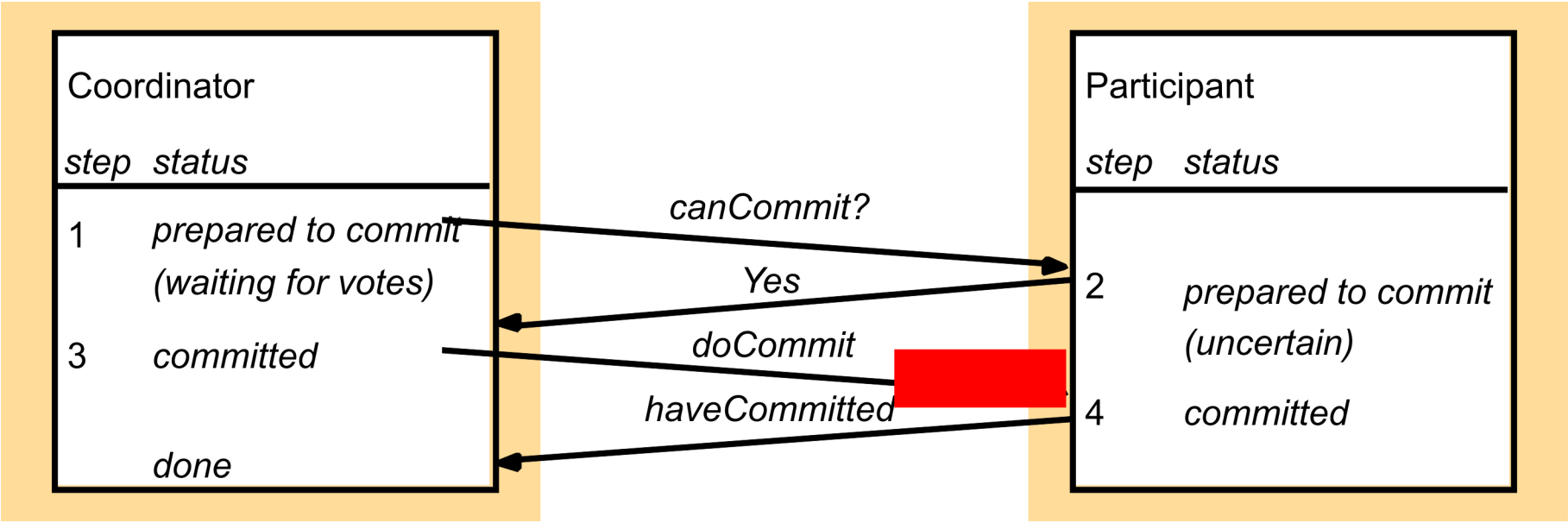


V 2PCP jsou čtyři místa, kde může dojít k selhání komunikace

✓ Participant hlasoval ano, ale neobdržel rozhodnutí ukončit/zkrachovat.

V tomto případě se participant nemůže rozhodnout jednostranně, protože není jasné, jaké rozhodnutí koordinátor vydá.

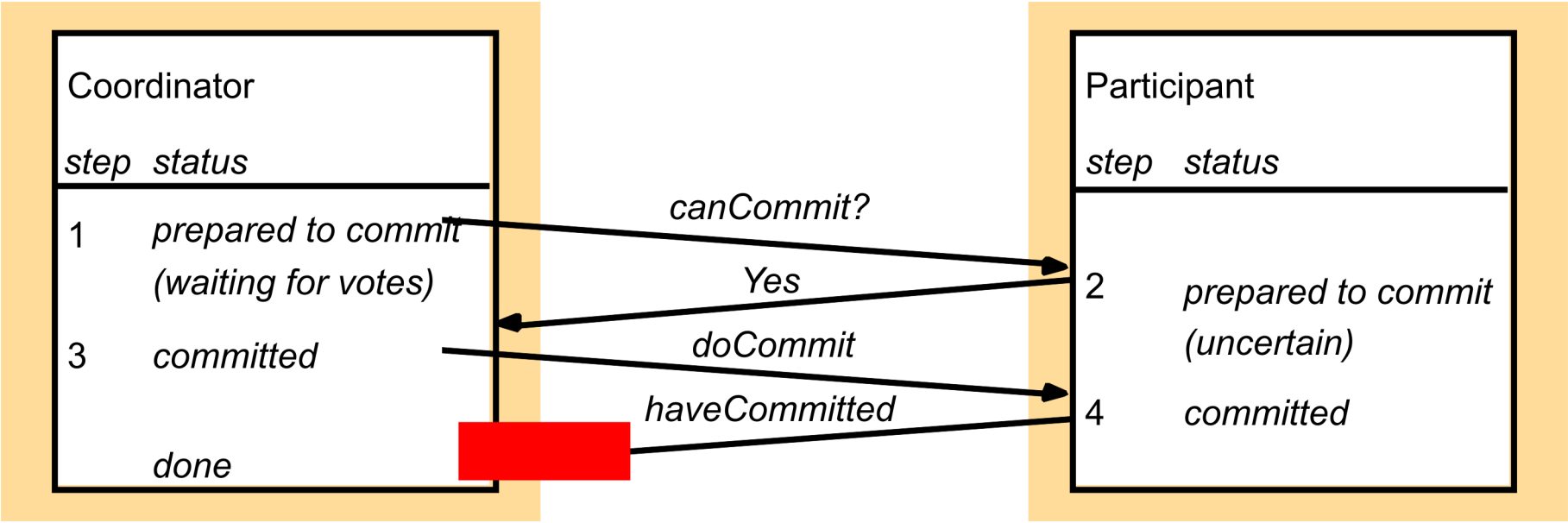
Participant je v tomto případě blokován, dokud znovu nenaváže komunikaci s koordinátorem a poté, co ji znovu naváže, participant požádá od koordinátora o konečné rozhodnutí, prosadí pokračování protokolu a potvrdí koordinátorovi rozhodnutí.



V 2PCP jsou čtyři místa, kde může dojít k selhání komunikace

- Koordinátor čeká na potvrzování od participantů.

  V tomto případě koordinátor po obnově komunikace znovu posílá rozhodnutí těm participantům, kteří rozhodnutí dosud nepotvrdili.



Řešení ukončení transakce, Two-Phase Commit Protocol

- Připomenutí
  - ✓ Koordinátor nemůže jen tak jednoduše vymazat informace týkající se transakce z protokolové tabulky nebo ze svého stabilního deníku dokud neobdrží potvrzení od všech participantů.

Obnova koordinátora po výpadku uzlu koordinátora

- ✓ koordinátor po restartu čte deník ze stabilní paměti a znovu vytváří protokolární tabulku tak, aby zachycovala stav 2PCP pro všechny probíhající transakce před výpadkem.
- ✓ Transakce, které byly aktivní podle koordinátora před jeho výpadkem a nemají v deníku záznam *decision*, koordinátor krachuje.
- ✓ Transakcím, které začaly 2PCP a ještě ho nedokončily před výpadkem (tj. transakce, které mají v jeho deníku záznamy *decision* bez odpovídajících záznamů *end*) koordinátor dokončí protokol zasláním rozhodnutí a vyčká na jejich potvrzení.

  Jelikož někteří z participantů již mohli obdržet rozhodnutí před výpadkem a prosadili ho, mohli tito účastníci již zapomenout, že tato transakce kdy existovala.

  Takoví účastníci jednoduše odpoví slepým potvrzením, kterým indikují, že již obdrželi a prosadili rozhodnutí před výpadkem.

Obnova participanta po výpadku uzlu participanta

- ✓ vyhledává ve svém deníku existenci transakcí, které jsou ve stavu ,,připravená na ukončení''
  (tj. mají v deníku záznam *prepared* bez záznamu *decision*).
- ✓ Pro vyhledanou transakci participant požádá koordinátora o zaslání rozhodnutí a poté co rozhodnutí obdrží prosadí rozhodnutí a potvrdí.
- ✓ Koordinátor bude vždy schopný reagovat na tyto dotazy,
  protože transakci nemůže zapomenout dříve,
  než obdrží potvrzení všech participantů.

Výpadky při řešení 2PCP

- Participant hlasoval v 1. fázi 2PCP Yes a čeká na sdělení výsledku dohody – *commit* / *abort*
  - ✓ Participant je ve stavu neurčitosti (*uncertain*)
    netuší jak bude znít rozhodnutí, nemůže rozhodnout jednostranně
  - ✓ Nemůže uvolnit objekty držené pro aktuálně řešenou transakci pro použití jinými souběžnými transakcemi
- Výpadek koordinátora v tomto kroku protokolu
  - ✓ participant musí čekat na obnovu koordinátora
  - ✓ po uplynutí časového limitu může požádat o zopakování zprávy o rozhodnutí
  - ✓ po x zopakovaní takových žádostí krachuje
  - ✓ případně se může dotazovat ostatních participantů jak znělo rozhodnutí
  - ✓ pokud ale tito budou rovněž ve stavu neurčitosti, nedozví se ale nic

Výpadky při řešení 2PCP

- Participant nedostal po ukončení svých operací dotaz canCommit?
  - ✓ Nezbývá nic jiného po uplynutí časového limitu než jednostranně udělat *abort*
- Koordinátor nedostal do uplynutí časového limitu hlas od některého participanta
  - ✓ Rozhoduje *abort* transakce
  - ✓ Toto rozhodnutí rozešle všem participantům
  - ✓ pokud některý opožděný participant bude poté hlasovat *commit*, zůstane ve stavu neurčitosti (řešení viz výše)

Hodnocení 2PCP

- 2PCP je blokující protokol,
  - drží si zamčené zdroje během čekání na zprávu, neuvolňuje je,
  - ostatní procesy, které tyto zdroje požadují, čekají
- když vypadne uzel s koordinátorem,
  kohorta sama transakci neukončí,
  zdroje držené v uzlech kohorty budou držené do doby dokud uzel s koordinátorem neobnoví svoji činnost
- 2PCP je konzervativní,
  při nejistotě dává přednost zrušení (*abort*)
  před dokončením (*commit*)

Prevence uváznutí souběžných transakcí

- každá transakce si zamkne všechna data dříve než se spustí
  - ✓ eliminace nutné podmínky uváznutí – *hold-and-wait*
  - ✓ velmi neefektivní řešení,
    zvláště pro distribuované prostředí nepoužitelné
- nebo se definuje se pořadí zamykání dat
  - ✓ eliminace postačující podmínky uváznutí ve *wait-for* grafu
  - ✓ úplné uspořádání dat + 2PLP aplikovaný na toto pořadí
- nebo se aplikují priority transakcí odvozené z běhu času
  - ✓ starší transakce má vyšší prioritu, má právo čekat
- nebo se připustí preempce transakce,
  která drží potřebné zdroje
  - ✓ eliminace nutné podmínky uváznutí *no preemption*

Prevence uváznutí na bázi rozhodovacích TS

- rozhodovací TS se transakci přiděluje při jejím vytvoření a při případném návratu transakce zpět na začátek se nemění
  - ✓ Wait-Die Scheme, čekej nebo zemři (zemři = dělej se znovu)
    - starší T čeká na mladší T dokud tato neuvolní datovou položku
    - mladší T nečeká na starší T, vždy je vrácená (zemře) a a spouští se znovu, ale se zachováním věku (možná i vícekrát, ale postupně stárne a jednou bude nejstarší . . .)
  - ✓ Wound-Wait Scheme, poraň (ať se dělá někdo znovu) nebo čekej
    - starší T nečeká na mladší T dokud tato neuvolní datovou položku, raní ji (tj. prosadí její vrácení, ale se zachováním věku)
    - mladší T čeká na starší T dokud tato neuvolní datovou položku
- v reále schéma Wait-Die Scheme často způsobuje více návratů než schéma Wound-Wait Scheme

Prevence uváznutí transakcí na bázi TS, Wait-Die Scheme

- Založeno na nepreemptivní technice
- Když se T_i iniciálně startuje, získá rozhodovací TS_i
- Nechť při uplatnění žádosti T_i s TS_i o jistý zámek drží tento zámek T_j s TS_j
  - ✓ je-li žádající T_i starší než okamžitý vlastník prostředku, – transakce T_j, $TS_i < TS_j$, T_i počká na uvolnění zámku a získá ho po té, jakmile jej držitel zámku, tj. transakce T_j, uvolní
  - ✓ je-li žádající T_i mladší než okamžitý držitel zámku – T_j, $TS_i > TS_j$, T_i je vrácená, žádost o přidělení opakuje, ale se stejným TS_i
- Takže platí – jestliže T_i požaduje zámek držený T_j, smí T_i čekat (*wait*) jen když $TS_i < TS_j$ (T_i je starší než T_j). Jinak je T_i vrácená v historii zpět na vydání žádosti (*dies*, zemře).

Prevence uváznutí transakcí na bázi TS, Wait-Die Scheme

- Wait-Die Scheme zajišťuje, že transakce může zemřít vícekrát,
  ale ne nekonečně krát, jednou přijde její čas,
  kdy si na uvolnění prostředku určitě počká,
  protože bude ze všech transakcí nejstarší
- Příklad:

  Transakce T_1, T_2 a T_3 mají TS s hodnotami postupně 5, 10, resp. 15.

  Jestliže T_1 ($TS_1 = 5$) požaduje zámek držený T_2 ($TS_2 = 10$),
  T_1 bude čekat na uvolnění zámku transakcí T_2

  Jestliže T_3 ($TS_3 = 15$) požaduje zámek držený transakcí T_2
  ($TS_2 = 10$), pak T_3 zemře, tj. bude vrácená na zopakování žádosti o
  přidělení zámku, časové razítko se jí zachová a tak časem se stává
  starší a starší a má stále větší šanci si na uvolnění zámku počkat.

Prevence uváznutí transakcí na bázi TS, Wound-Wait Scheme

- Wound-Wait Scheme zajišťuje,
  že starší transakce nikdy nečeká na mladší transakci
- Schéma je založené na preemptivní technice
  - ✓ je-li žádající T_i starší než okamžitý vlastník prostředku – T_j ,
    T_i odebere T_j zámek a T_j se v historii vrátí, bude zopakovat žádost
    o jeho přidělení, se stejným časovým razítkem (*roll-back*, *undo*)
  - ✓ je-li žádající T_i mladší než okamžitý vlastník prostředku – T_j,
    T_i čeká na uvolnění zámku a získá ho po té
    jakmile jej dosavadní držitel zámku T_j uvolní
- návratů bývá méně, čekání více
- obě schémata wait-die i wound-wait restartují transakce
  s původním rozhodovacím TS, což zabraňuje stárnutí

Prevence uváznutí transakcí na bázi TS, Wound-Wait Scheme

Příklad:

- Transakce T_1, T_2 a T_3 mají TS s hodnotami 5, 10, resp. 15
- Jestliže T_1 ($TS_1 = 5$) požaduje zámek
  držený transakcí T_2 ($TS_2 = 10$),
  přebere T_1 zámek od T_2,
  T_2 je ,,zraněná''transakcí T_1, je vrácená, ...
- Jestliže T_3 ($TS_3 = 15$) požaduje zámek
  držený transakcí T_2 ($TS_2 = 10$),
  transakce T_3 bude čekat na jeho uvolnění

Klasifikace poruch

- poruchy transakcí
  - ✓ logické chyby

    v řešení T nelze pokračovat v důsledku nějakých vnitřních chybových podmínek aplikace/transakce

    – chybný vstup, nenalezení dat, přetok, . . .

    – Vesměs neobnovitelná činnost, havárie / krach celé aplikace
  - ✓ běh aplikace nemá smysl obnovovat
  - ✓ systémem detekovatelné chyby

    chybu detekuje podpůrný systém (TPM, OS, DBS, . . .), aktivní T krachuje důsledkem např. uváznutí, uplynutím časového limitu, . . .).

    – Zkrachovaná transakce může být následně spuštěná znovu např. protokolem TPM řešícím uváznutí, nebo klientem z důvodu uplynutí časového limitu
  - ✓ neobnovuje se běh aplikace, opakovaně se spouští zkrachovalá T

Klasifikace poruch

- Porucha disku
  - ✓ padnutí hlav disku na povrch, chyba paměťového média, porucha vyřazující diskový mechanismus, ...
  - ✓ předpoklad – porucha je detekovatelná (kontrolní součty, ...)
  - ✓ Vyšší spolehlivost lze dosáhnout aplikací některé z metod RAID
  - ✓ Po obnově z periodicky prováděných záložních kopií dat a kontrolních bodů aplikace lze aplikaci od kontrolního bodu spustit znovu.
- (Dočasný) výpadek systému
  - ✓ výpadek energie, hw porucha, sw porucha systému, ...
  - ✓ předpoklady – obsah energeticky nezávislých pamětí se výpadkem systému nepoškozuje, běh vlastního systému lze posléze obnovit
  - ✓ neporušenost dat lze ověřovat prováděním integritních kontrol
  - ✓ Transakce nedokončená při výpadku systému krachuje a při obnově činnosti systému ji *Recovery Manager* (TPM) spustí znovu.

Algoritmy obnovy

- Jedná se o techniky zajišťující v prostředí s výpadky
  konzistenci báze dat,
  atomicitu transakcí
  vč. trvalosti výsledků provedených transakcí
- Algoritmy obnovy mají 2 části
  - ✓ Akce probíhající během normálního řešení transakcí
    cíl – zajistit dostatek informací nutných pro obnovu po výpadku
    cena – snížení výkonu pro aplikace, zvýšení nároků na paměť
  - ✓ obnovovací akce probíhající po výpadku,
    cíl – obnova obsahu báze dat do stavu, který zaručuje (viz výše):
    – konzistenci báze dat,
    – atomicitu transakcí a
    – trvalost výsledků transakcí

Obnovitelnost transakčního zpracování, správce obnovy

- Obnovitelnost po výpadku / poruše řeší součást TPM – správce obnovy, *recovery manager*
- správce obnovy má úkoly
  - ✓ uchovávat informace o modifikacích hodnot objektů prováděných transakcemi v permanentní paměti (vytváří obnovovací soubor, *recovery file*)
  - ✓ po opětovném spuštění po výpadku obnovovat původní hodnoty objektů v serveru (původní = výsledek provedených transakcí)
  - ✓ průběžně reorganizovat obnovovací soubor s cílem dosáhnout co nejvyšší výkon obnovy
  - ✓ udržovat dostatečný paměťový prostor na disku pro obnovovací soubor
- Pokud se požaduje obnovitelnost i při poruše permanentní paměti, je potřeba použít zrcadlení disků (RAID) apod.

Metody údržby obnovovacího souboru

- Vedení deníku, *log*
- Obnovovací proces musí být idempotentní operací
  - ✓ lze jej opakovat vícekrát se stejným výsledkem

Vedení deníku

- Změny v databázi spravované serverem se během provádění všech transakcí zaznamenávají do deníku (*log*)
- Deníkové záznamy se vytváří apriorním zaznamenáváním – zaznamenáváním předem, *write-ahead logging*,
  - ✓ TPM je vytváří před vlastním provedením operací nad daty
  - ✓ zapisují se na disk přímo, bez vyrovnávání toku na úrovni OS (TPM vyvolá bezprostřední provedení služby OS *output*)
  - ✓ Deník musí TPM periodicky čistit, dále již nepotřebné deníkové záznamy vymazává.
- Pořadí záznamů v deníku odráží historii běhu transakčního zpracování
- Deník obsahuje aktuální, poslední obraz hodnot objektů + historii transakcí, které tento obraz vyprodukovaly

Typy záznamů v obnovovacím souboru typu deník

- Záznam o startu transakce, $< T_i \ start >$
- Záznam o korekci položky/objektu, $< T_i, X_j, V_1, V_2 >$
  - ✓ T_i identifikátor transakce, která provedla *write*
  - ✓ X_j identifikátor položky/objektu
  - ✓ V_1 původní hodnota položky/objektu
  - ✓ V_2 nová hodnota položky/objektu
  - ✓ záznam umožňuje provést *undo* změn učiněných zkrachovalou T
  - ✓ záznam umožňuje provést *redo* změn učiněných provedenou T, jejíž výsledky se před výpadkem nezapsaly trvale na disk
    - T modifikuje DB $\equiv$ provádí *write* do bufferu v RAM, který OS následně, někdy, službou *output* vypisuje na disk
- Záznam o provedení transakce, $< T_i \ commit >$
- Záznam o zkrachování transakce, $< T_i \ abort >$

Vedení deníku

- Správce obnovy je aktivován kdykoliv transakce
  - ✓ je spouštěná – správce zaznamenává v obnovovacím souboru existenci transakce, $< T_i \; start >$
  - ✓ zapisuje do DB – správce aktualizuje obnovovací soubor záznamem o korekci objektu, $< T_i, X_j, V_1, V_2 >$
  - ✓ je provedená / zkrachuje – správce aktualizuje v obnovovacím souboru stav transakce záznamem o provedení či krachování,

    $< T_i \; commit >$, $< T_i \; abort >$
- Deník není aplikační součástí báze dat, deník (obnovovací soubor) si uchovává v permanentní paměti TPM
- Při obnově po výpadku se ruší účinky každé transakce, která nemá v deníku záznam o provedení – $< T_i \; commit >$, všechny podle deníku neprovedené transakce krachují

Vedení deníku

- TPM v určitých intervalech vytváří v deníku kopie stavu celé DB, tzv. kontrolní body, *checkpoints*
- Historii transakcí provedených do kontrolního bodu lze z deníku vymazat
- Při obnově báze dat po výpadku TPM pak zrekonstruuje stav DB před výpadkem podle informací uvedených v posledním kontrolním bodu v deníku
- Zápis do obnovovacího souboru se řeší atomicky, zápisy do obnovovacího souboru jsou přímé, nevyrovnávané zápisy, pokud server vypadne, je vadný pouze poslední zápis
- Obnovovací soubor je udržovaný jako sekvenční soubor, sekvenční zápisy na disk lze řešit poměrně rychle

Algoritmus obnovy – bázové vlastnosti

- Protože deník obsahuje jak staré tak i nové hodnoty objektů, lze tudíž řešit jak *undo*, tak i *redo* transakce
- Algoritmy obnovy obvykle vyžadují, aby objekt modifikovaný jednou transakcí nemohla modifikovat jiná transakce, dokud se prvá transakce neprovede nebo nezkrachuje – musí se použít alespoň striktní 2-fázové zamykání apod. – exkluzivní zámky se uvolňují až při *commit*/*abort* transakce
- Možnost obnovitelnosti se dosahuje za cenu omezení souběžnosti, :–(

Obnova podle deníku, Log-based Recovery

- Záznam o korekci se do deníku na disku zapisuje před modifikací objektu v DB a díky tomu lze v DB udělat
  - ✓ modifikaci objektu kdykoliv později, kdy to je optimální
  - ✓ *undo* modifikací udělaných zkrachovanými / nedokončenými transakcemi
  - ✓ *redo* modifikací udělaných provedenými transakcemi, které se do výpadku nestačily vypsat do DB na disku

Obnova podle deníku, Log-based Recovery

- Jakmile se transakce se stane hotová do deníku na disku se zapíše záznam o provedení transakce $< T_i \; commit >$
  - ✓ v tomto okamžiku jsou v deníku už všechny dosavadní deníkové záznamy dané transakce
  - ✓ provedenou transakci lze tudíž kdykoli (po výpadku) zopakovat (*redo*) z hlediska finálních hodnot objektů
- jestliže dojde k výpadku a v deníku záznam $< T_i \; commit >$ není, účinky transakce se při obnově činnosti ruší, vrací se zpět na původní hodnoty modifikovaných objektů
  - ✓ Transakce se považuje za zkrachovalou, provádí se *undo* podle deníku

Obnova podle deníku, Log-based Recovery, zápory

- celkově dochází ke snižování výkonu
  - ✓ zápis dat se provádí $\mathbf{2}\times$ ($\mathbf{1}\times$ do deníku a $\mathbf{1}\times$do DB),
  - ✓ do deníku se zapisují záznamy o startech, provedení a krachování transakcí
- zvyšuje se paměťová složitost
  - ✓ deník (log) vyžaduje pro své uložení další paměť na disku

Použité obnovovací procedury

- používají deník k nalezení objektů modifikovaných T_i a starých a nových hodnot těchto objektů
- $redo(T_i)$
  - ✓ nastavuje objekty modifikované T_i na nové hodnoty
  - ✓ pořadí nastavování musí být shodné s pořadím modifikací
  - ✓ deník se obvykle prohledává systematicky od začátku a nastavuje se nová hodnota pro každý modifikovaný objekt

Použité obnovovací procedury

- $undo(T_i)$
  - ✓ nastavuje objekty modifikované T_i na staré, iniciální hodnoty
  - ✓ operace $undo(T_i)$ se provádí právě jedenkrát v případech, že se v deníku nenalezne ani *commit* ani *abort* T_i a řeší se návrat během normálního zpracování nebo při obnově po výpadku.
  - ✓ pořadí provádění $undo(T_i)$ – zpětně od posledního záznamu v deníku pro T_i

Kontrolní body, Checkpoints

- deníky transakcí mohou být dlouhé, obnova pak trvá dlouho
- mohou se zbytečně dělat *redo* transakcí,
  které již mají udělané výstupy (*outputs*) do báze dat
- periodicky vytvářené kontrolní body dobu obnovy zkrátí
- během vytváření kontrolního bodu nesmějí transakce dělat
  korekce dat, nesmí zapisovat záznamy do deníku apod.
- při periodickém vytváření kontrolních bodů se provádí:
  - ✓ výstup všech deníkových záznamů umístěných v daném okamžiku v RAM z RAM (energeticky závislé paměti) do permanentní paměti
  - ✓ výstup všech modifikovaných bloků vyr. pam. na disk (perm. paměť)
  - ✓ výstup záznamu $< checkpoint\ L >$ do deníku (*log*) udržovaného v energeticky nezávislé paměti
  - ✓ L je seznam aktivních transakcí v době tvorby kontrolního bodu

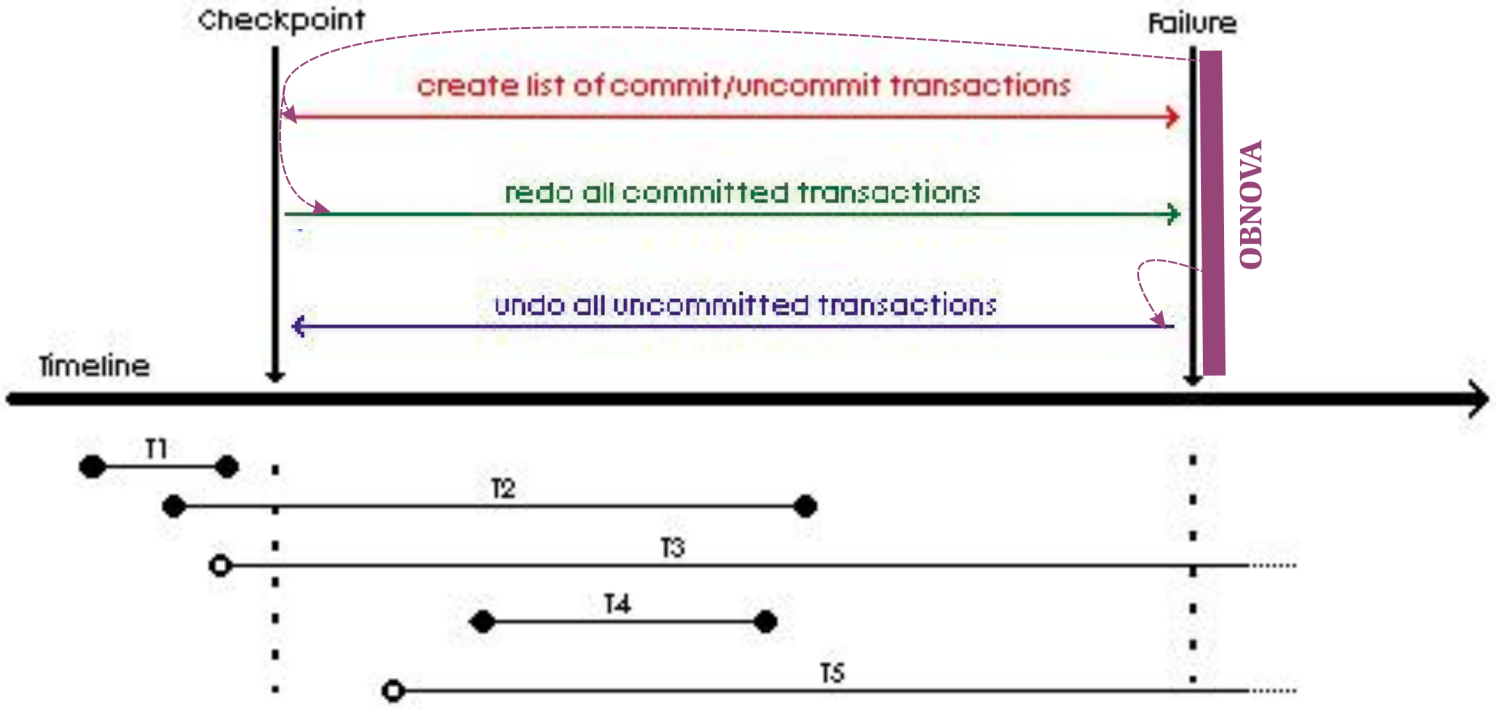
Kontrolní body, Checkpoints

- T_i provedené před zápisem $< checkpoint\ L >$ se neobnovují pomocí *redo*, jejich výsledek je součástí dat kontrolního bodu
- Obnova po výpadku se týká pouze transakcí
  - aktivních v okamžiku vytváření kontrolního bodu a
  - všech transakcí následujících po těchto transakcích

  ✓ Pro transakce nemající v deníku $< commit >$ se provede *undo*

  ✓ Pro transakce mající v deníku $< commit >$ se provede *redo*

Příklad obnovy z kontrolního bodu



- ✓ výsledek T_1 je obsažený v kontrolním bodu, nespouští se žádná akce
- ✓ T_2 a T_4 jsou v době výpadku již provedené,
  T_3 a T_5 nejsou v době výpadku provedené,
- ✓ připraví se seznam *undo* a provede se $redo\ T_4$ a pak $redo\ T_2$
- ✓ provede se $undo\ T_5$ a pak $undo\ T_3$