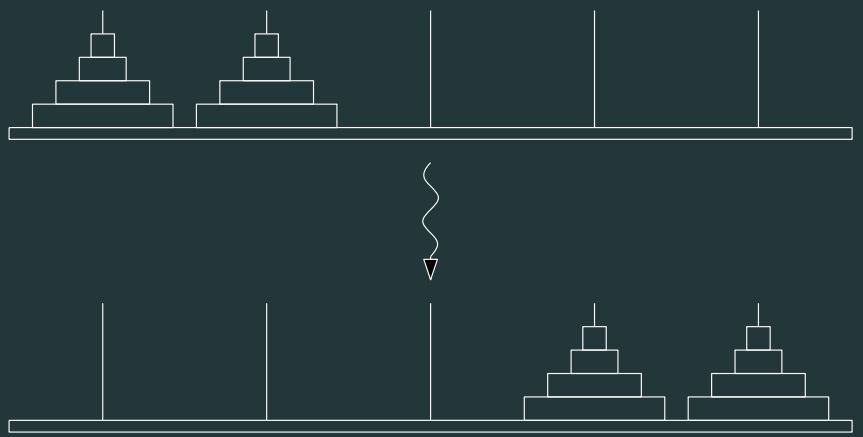
Čas a detekce selhání v distribuovaných systémech

B4B36PDV – Paralelní a distribuované výpočty

Osnova

- Opakování z minulého cvičení
- Čas a uspořádání událostí v distribuovaných systémech
- Detekce selhání v distribuovaných systémech
- Zadání 6. domácí úlohy

Odevzdání semestrální práce se blíží!

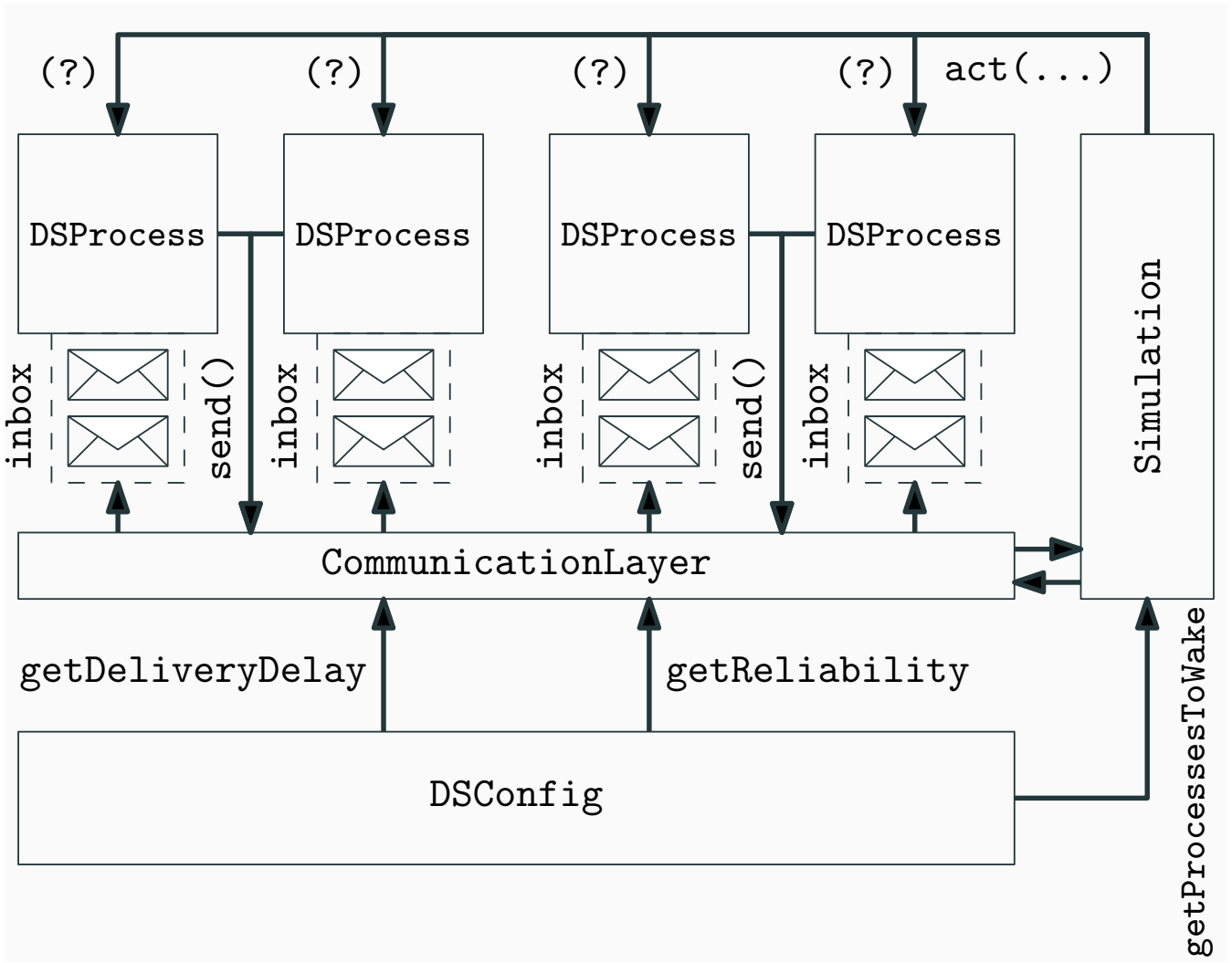


Čtvrtek 2.5. 23:59 CET

Opakování z minulého cvičení

http://goo.gl/a6BEMb

DSand framework



Jakými všemi způsoby může být následující kód proveden?

```
// CONFIG:
        getProcessesToWake() { return {"1", "2", "3"}; }
// PROCESS:
        int time = 0;
        int nid = Integer.parseInt(id);
        public void act() {
                time++;
                if(nid == time && nid != 3)
                        send("3", new DummyMessage();
                while(!inbox.isEmpty()){
                        Message m = inbox.poll();
                        System.out.println(m.sender);
                }
        }
```

Čas a uspořádání událostí v DS

V centralizovaném systému je čas konzistentní…
(procesy typicky sdílí jediné hodiny)

Sdílené hodiny můžeme snadno využít pro:

- Koordinaci
  („výpočet zahájíme v 11:47:23“)
- Uspořádání kroků výpočtu
  (logování, uspořádání procesů při přístupu do kritické sekce atd.)
- … a jiné

Jak jsme na tom s fyzickým časem v případě DS?

Uvažujte například, že si chcete domluvit čas telefonátu s kamarádem, který je na druhém konci světa...

Jak jsme na tom s fyzickým časem v případě DS?

Uvažujte například, že si chcete domluvit čas telefonátu s kamarádem, který je na druhém konci světa...

Každý z Vás má své vlastní hodinky...

Jak jsme na tom s fyzickým časem v případě DS?

Uvažujte například, že si chcete domluvit čas telefonátu s kamarádem, který je na druhém konci světa...

Každý z Vás má své vlastní hodinky...

- Hodinky mohou ukazovat rozdílný čas (*clock skew*)
  (jiná časová zóna, zapomenutá změna letního/zimního času, aj.)

Jak jsme na tom s fyzickým časem v případě DS?

Uvažujte například, že si chcete domluvit čas telefonátu s kamarádem, který je na druhém konci světa...

Každý z Vás má své vlastní hodinky...

- Hodinky mohou ukazovat rozdílný čas (*clock skew*)
  (jiná časová zóna, zapomenutá změna letního/zimního času, aj.)
- Nastavení stejného času před kamarádovým odletem nás nezachrání
  (hodinky kamaráda se mohou například opožďovat – *clock drift*)

Co s tím?

Co s tím?

- Pokusíme se čas zobrazovaný na hodinkách sladit (*synchronizovat*)
  (např. pošleme kamarádovi každých 15 minut SMS zprávu s naším časem)

Co s tím?

- Pokusíme se čas zobrazovaný na hodinkách sladit (*synchronizovat*)
  (např. pošleme kamarádovi každých 15 minut SMS zprávu s naším časem)

To ale nebude moc přesné... :-(

Co s tím?

- Pokusíme se čas zobrazovaný na hodinkách sladit (*synchronizovat*)
  (např. pošleme kamarádovi každých 15 minut SMS zprávu s naším časem)

  To ale nebude moc přesné… :-(

- Termín hovoru si domluvíme ad-hoc
  („zhruba v 11:47 ti napíšu, že už mám čas – ty mi zavolej zpět“)

Co s tím?

- Pokusíme se čas zobrazovaný na hodinkách sladit (*synchronizovat*)
  (např. pošleme kamarádovi každých 15 minut SMS zprávu s naším časem)

  To ale nebude moc přesné… :-(

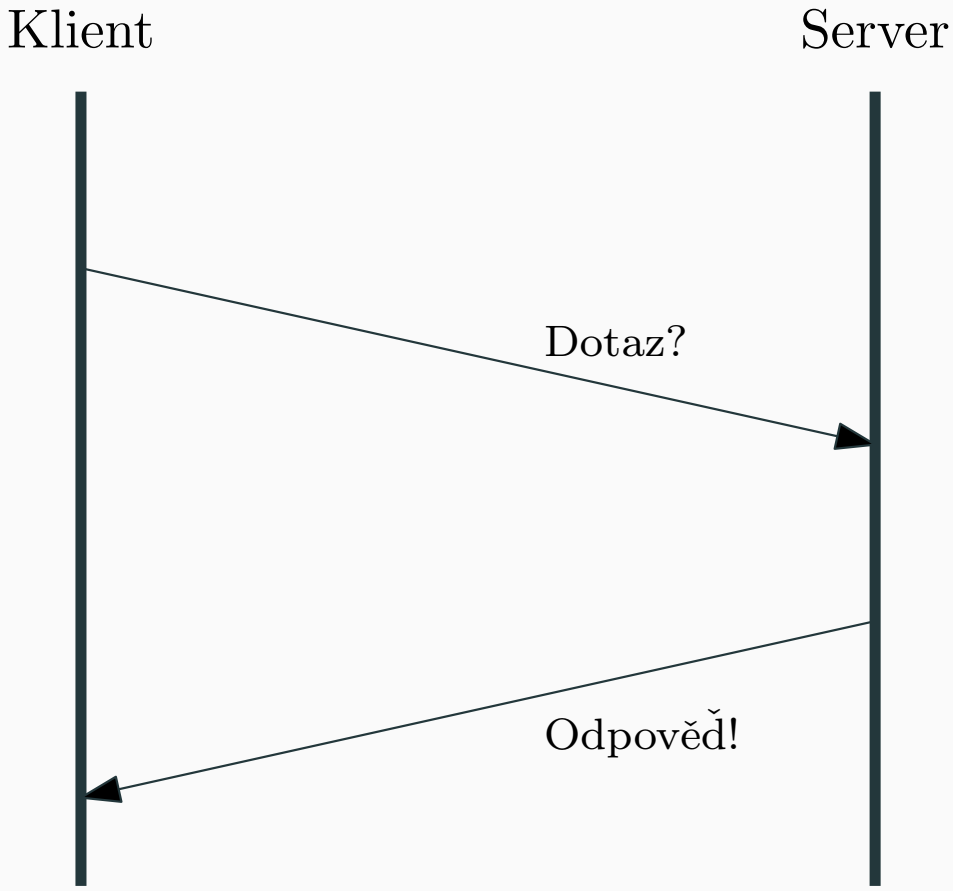
- Termín hovoru si domluvíme ad-hoc
  („zhruba v 11:47 ti napíšu, že už mám čas – ty mi zavolej zpět“)

  Kauzalita!

  Přijetí SMS → Zahájení hovoru

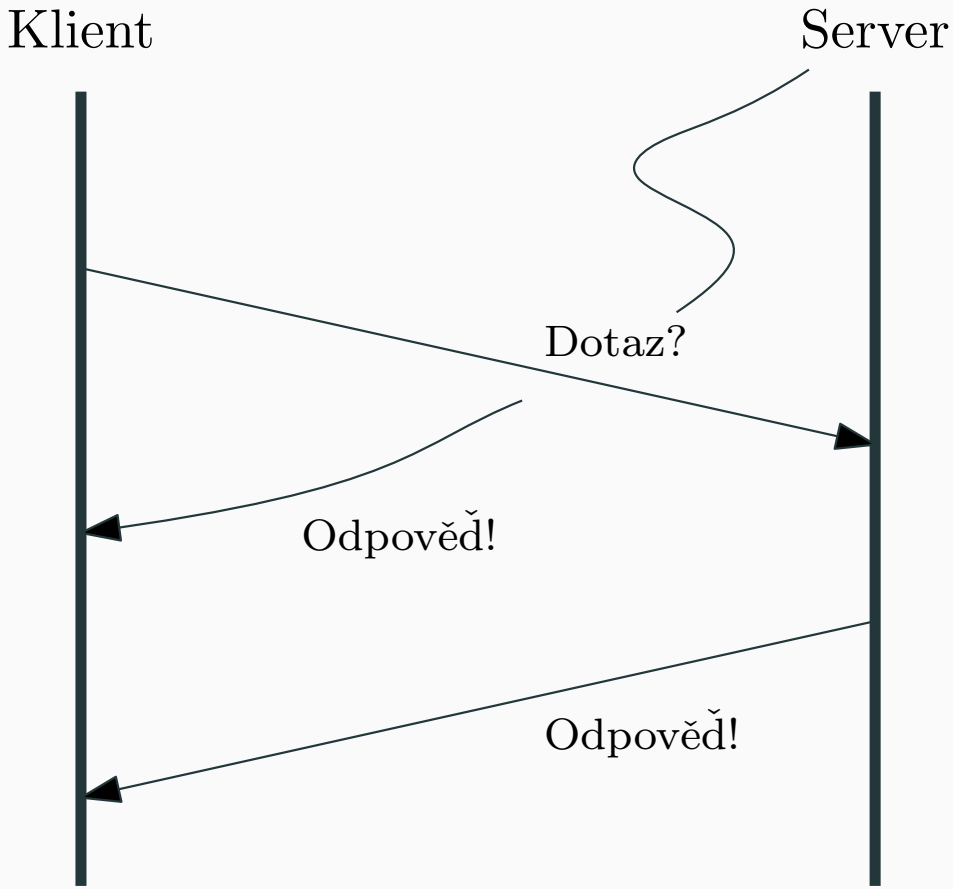
Vztah příčiny a následku je v DS klíčový!

(například, odpověď na dotaz následuje až po položení dotazu)



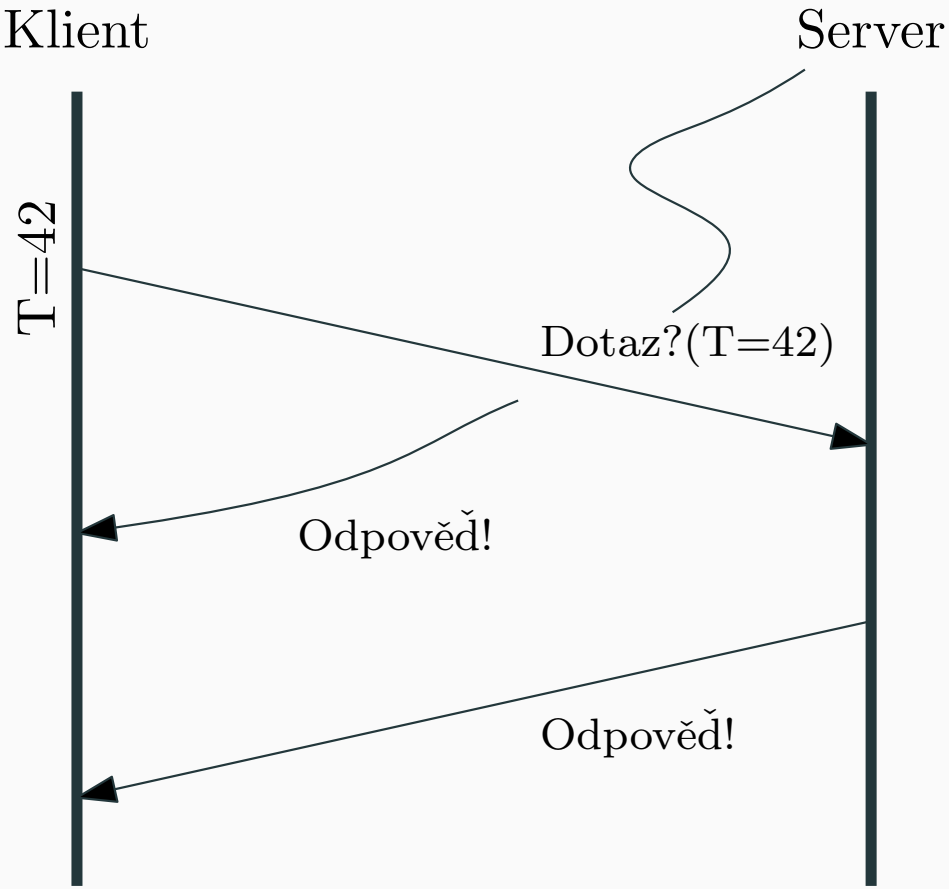
Vztah příčiny a následku je v DS klíčový!

(například, odpověď na dotaz následuje až po položení dotazu)



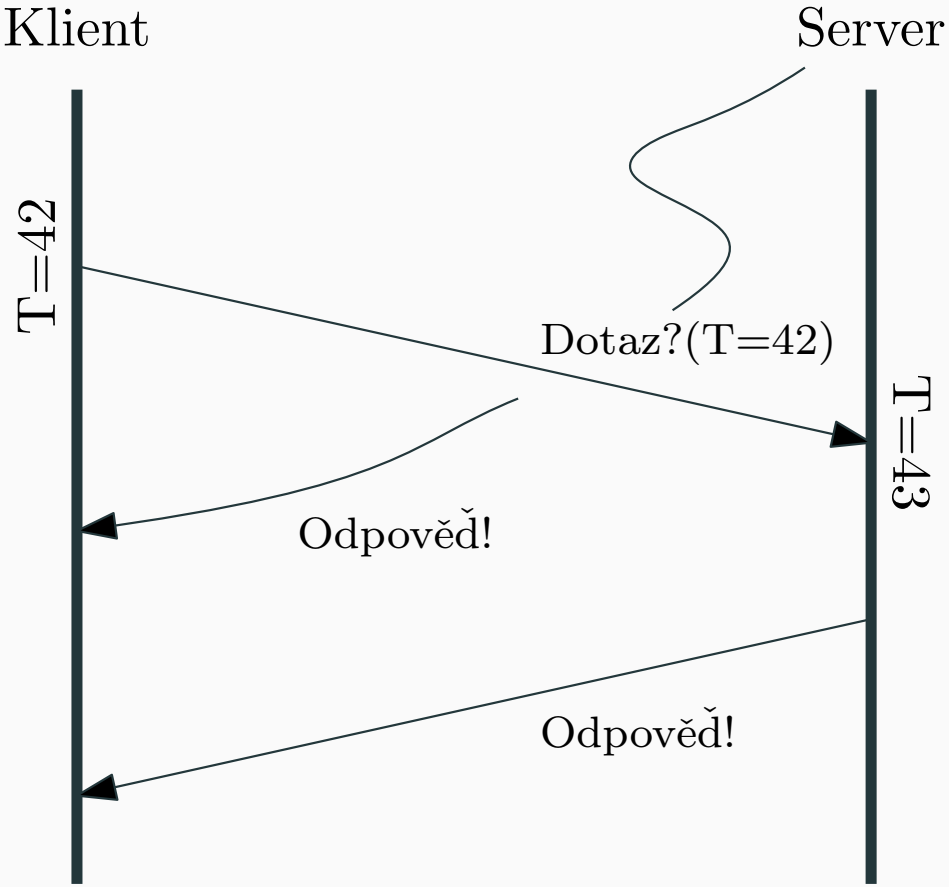
Vztah příčiny a následku je v DS klíčový!

(například, odpověď na dotaz následuje až po položení dotazu)



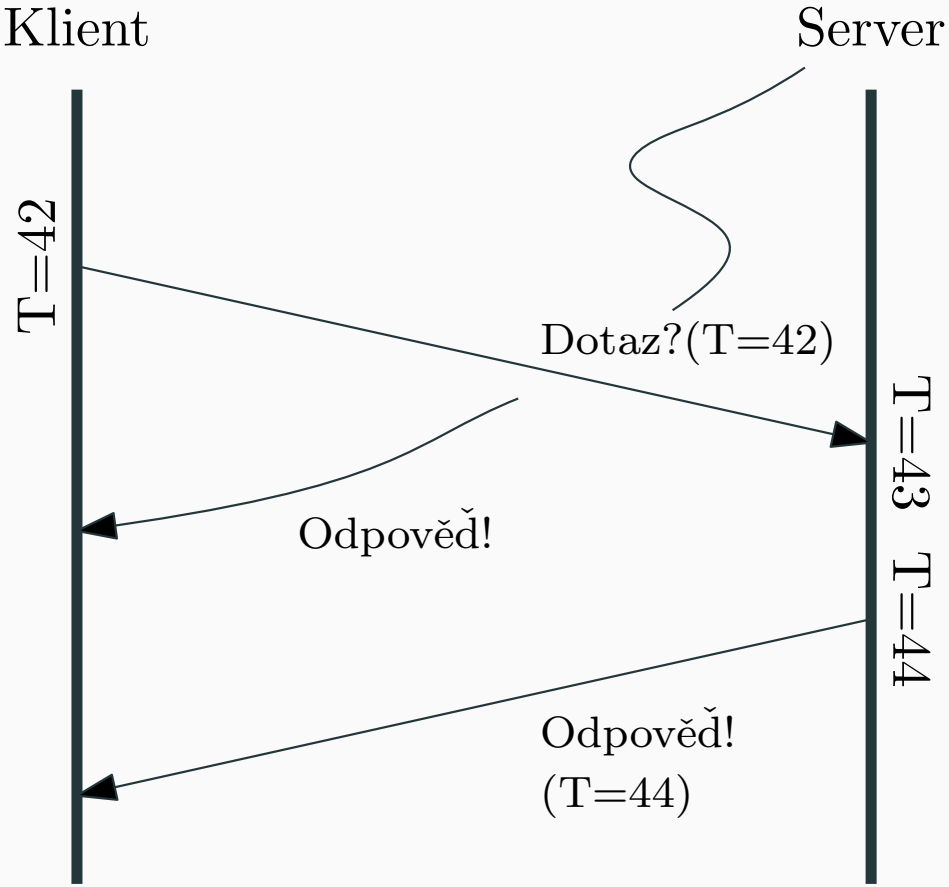
Vztah příčiny a následku je v DS klíčový!

(například, odpověď na dotaz následuje až po položení dotazu)



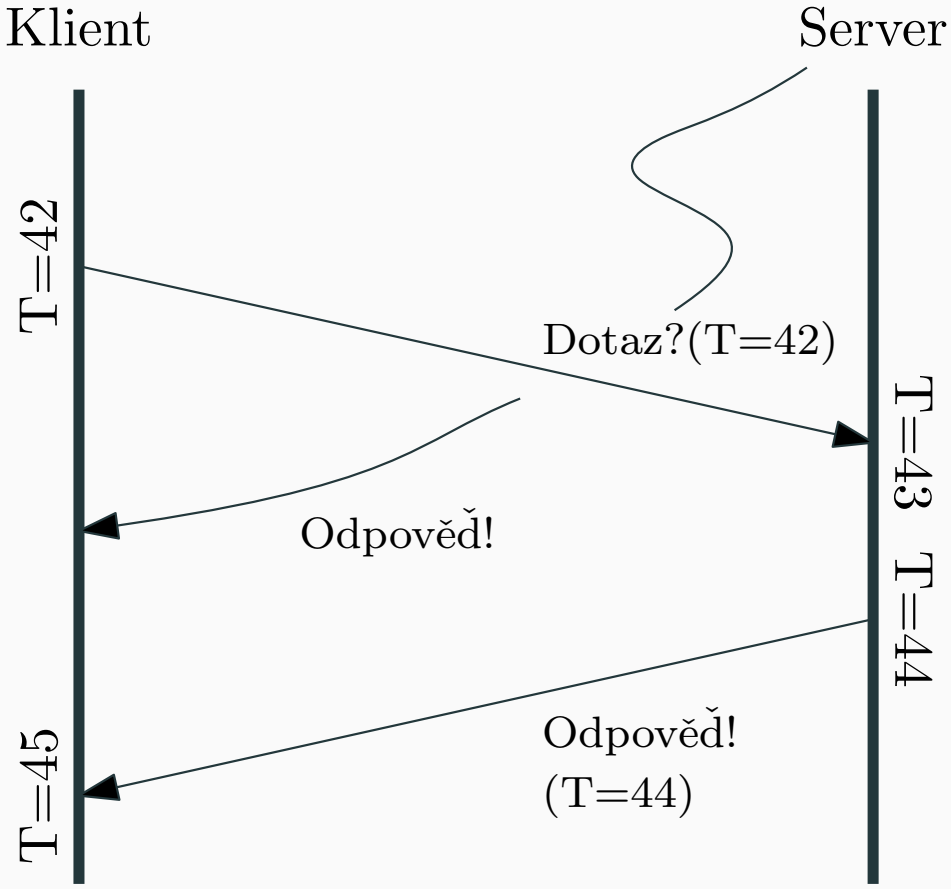
Vztah příčiny a následku je v DS klíčový!

(například, odpověď na dotaz následuje až po položení dotazu)



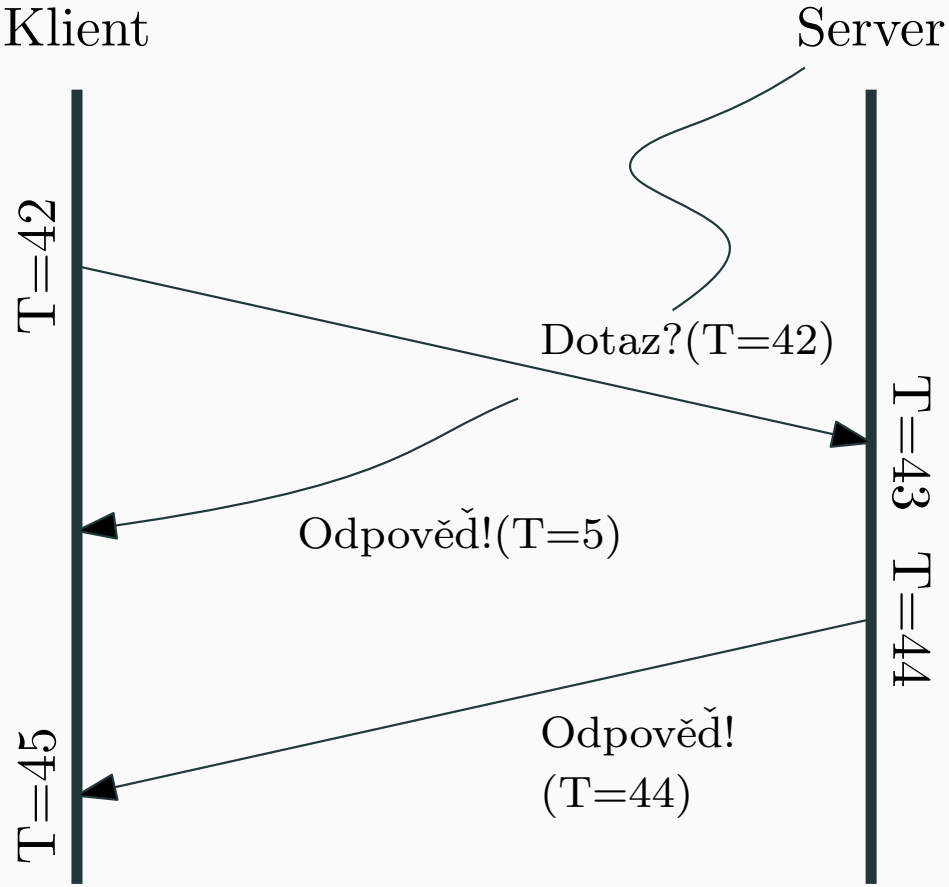
Vztah příčiny a následku je v DS klíčový!

(například, odpověď na dotaz následuje až po položení dotazu)



Vztah příčiny a následku je v DS klíčový!

(například, odpověď na dotaz následuje až po položení dotazu)



Logický čas

Právě jsme v našem DS zavedli logický čas! :-)

(konkrétně Lamportovy skalární hodiny)

Právě jsme v našem DS zavedli logický čas! :-)

(konkrétně Lamportovy skalární hodiny)

Logický čas splňuje pouze kauzalitu!

- Každé události e přiřadíme časovou značku $T(e)$
- Pokud je událost e příčinou události e', pak platí $T(e) < T(e')$
  (Ne nutně ale naopak!)

Lamportův algoritmus

Lamportův algoritmus

1. Každý proces má svoje lokální logické hodiny
   ```
   int logicalTime = 0
   ```

Lamportův algoritmus

1. Každý proces má svoje lokální logické hodiny
   ```
   int logicalTime = 0
   ```
2. Před každou významnou událostí (obzvlášť posláním zprávy!) si proces lokální čas posune
   ```
   ++logicalTime
   ```

Lamportův algoritmus

1. Každý proces má svoje lokální logické hodiny
   `int logicalTime = 0`
2. Před každou významnou událostí (obzvlášť posláním zprávy!) si proces lokální čas posune
   `++logicalTime`
3. Každé zprávě přiřadíme časovou značku $\texttt{msg.}T = \texttt{logicalTime}$
   (Tím říkáme přijímajícímu procesu, ať si upraví svůj čas!)

Lamportův algoritmus

1. Každý proces má svoje lokální logické hodiny
   `int logicalTime = 0`
2. Před každou významnou událostí (obzvlášť posláním zprávy!) si proces lokální čas posune
   `++logicalTime`
3. Každé zprávě přiřadíme časovou značku $\texttt{msg.}T = \texttt{logicalTime}$
   (Tím říkáme přijímajícímu procesu, ať si upraví svůj čas!)
4. Přijetí zprávy je následkem jejího odeslání – pak musí platit $T(e) < T(e')$
   Po přijetí zprávy `msg` si proto musíme zaktualizovat svůj `logicalTime`:

$$\texttt{logicalTime} = 1 + \max\{\texttt{logicalTime}, \texttt{msg.}T\}$$

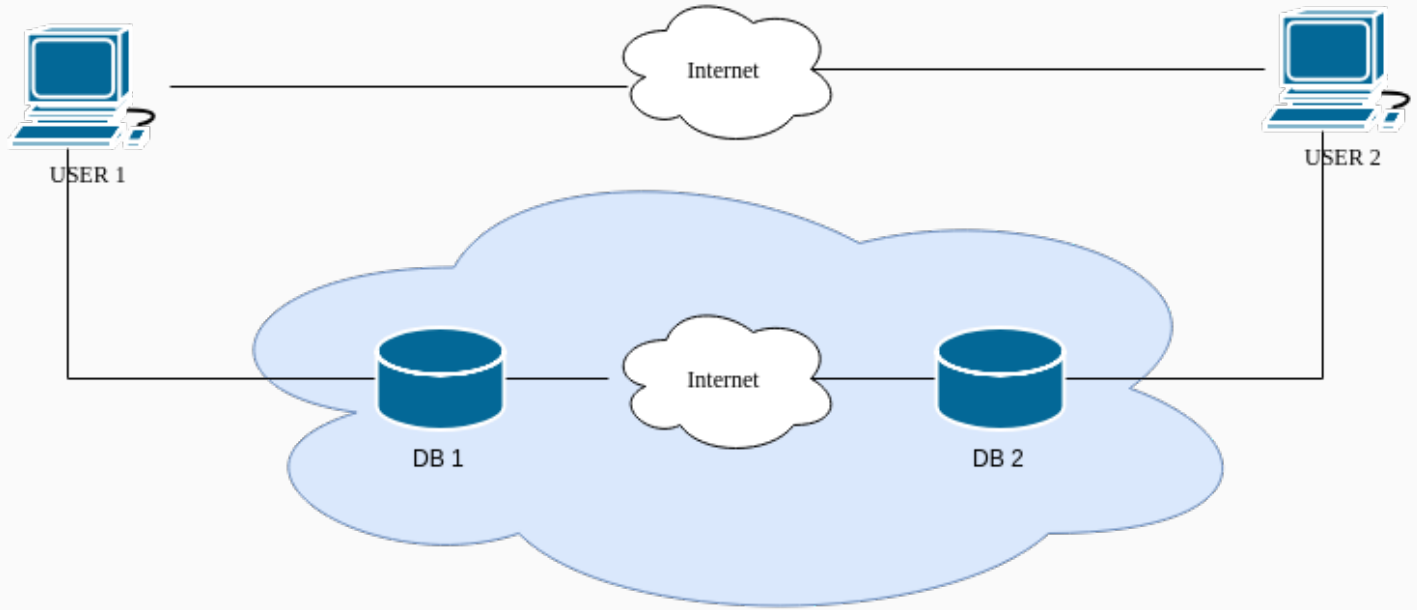
Lamportův algoritmus

1. Každý proces má svoje lokální logické hodiny
   `int logicalTime = 0`
2. Před každou významnou událostí (obzvlášť posláním zprávy!) si proces lokální čas posune
   `++logicalTime`
3. Každé zprávě přiřadíme časovou značku $\texttt{msg.}T = \texttt{logicalTime}$
   (Tím říkáme přijímajícímu procesu, ať si upraví svůj čas!)
4. Přijetí zprávy je následkem jejího odeslání – pak musí platit $T(e) < T(e')$
   Po přijetí zprávy `msg` si proto musíme zaktualizovat svůj `logicalTime`:

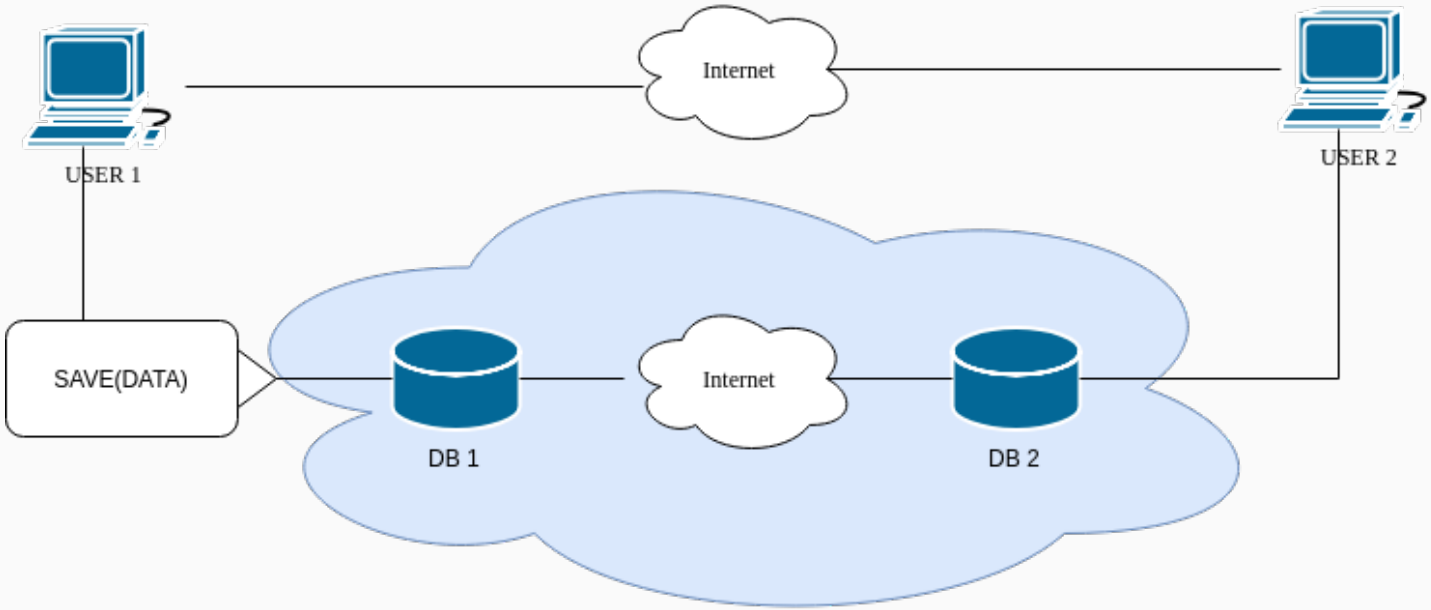
$$\texttt{logicalTime} = 1 + \max\{\texttt{logicalTime}, \texttt{msg.}T\}$$

⚠ Skalární hodiny jsou stavebním kamenem mnoha algoritmů v DS!

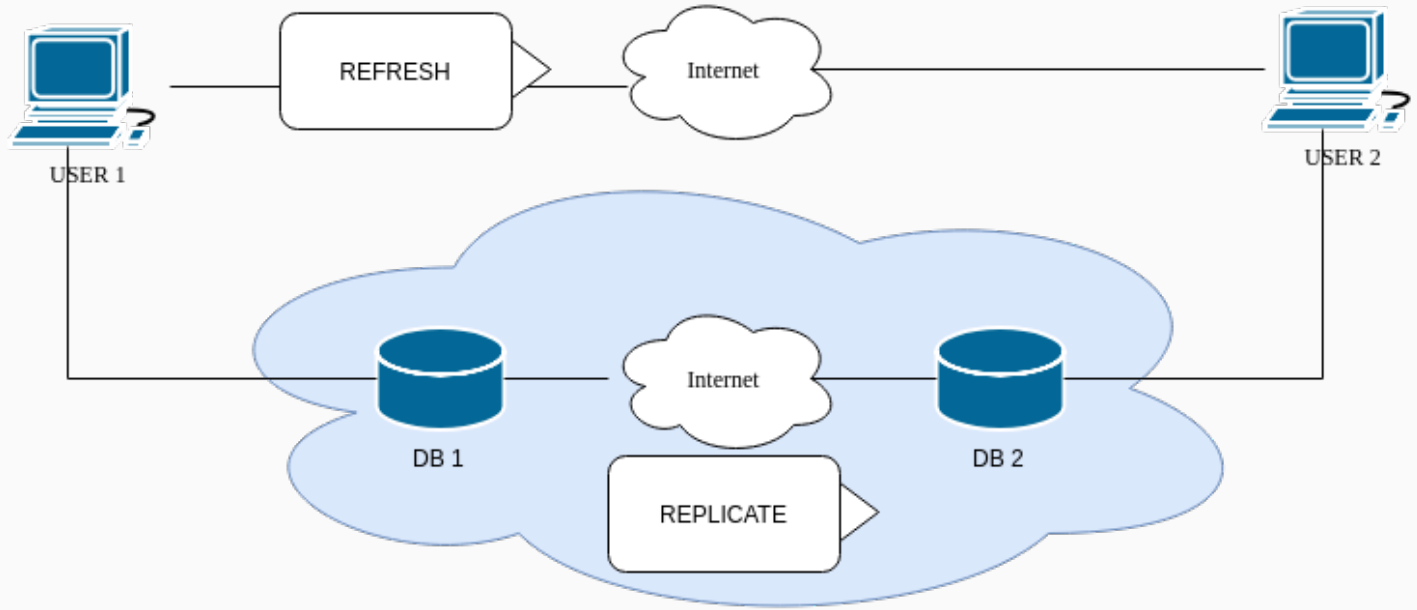
PDV Cloud



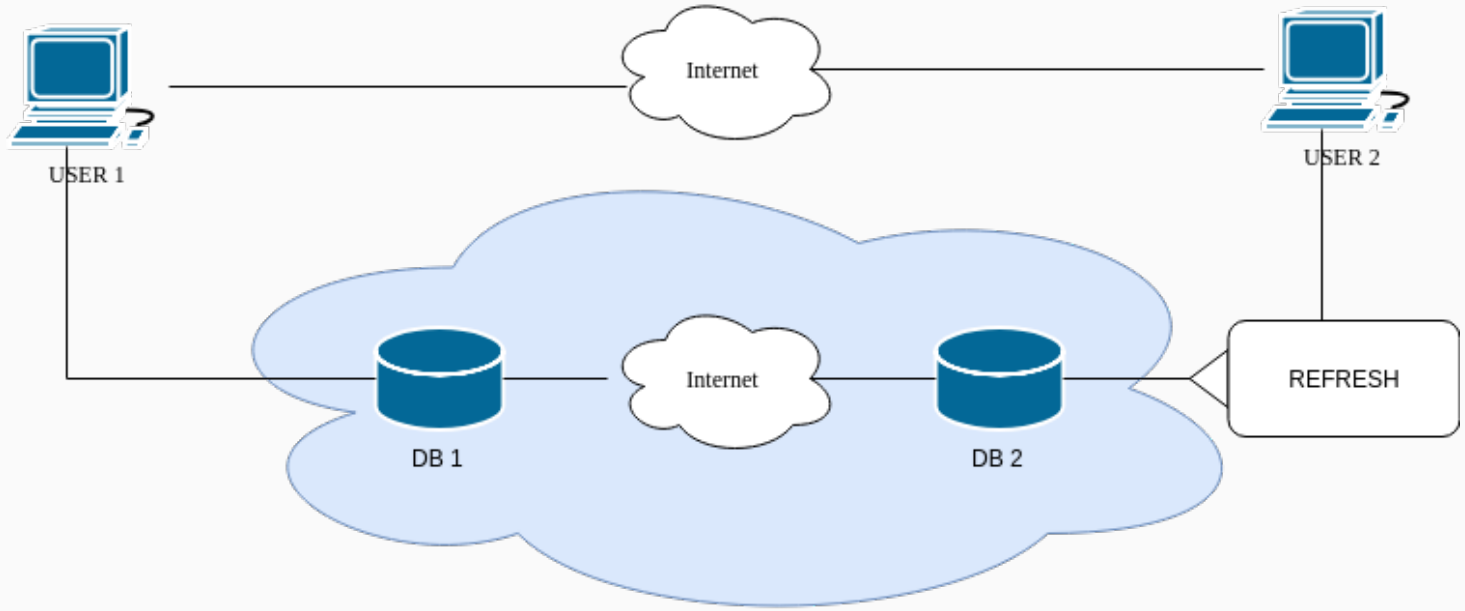
PDV Cloud



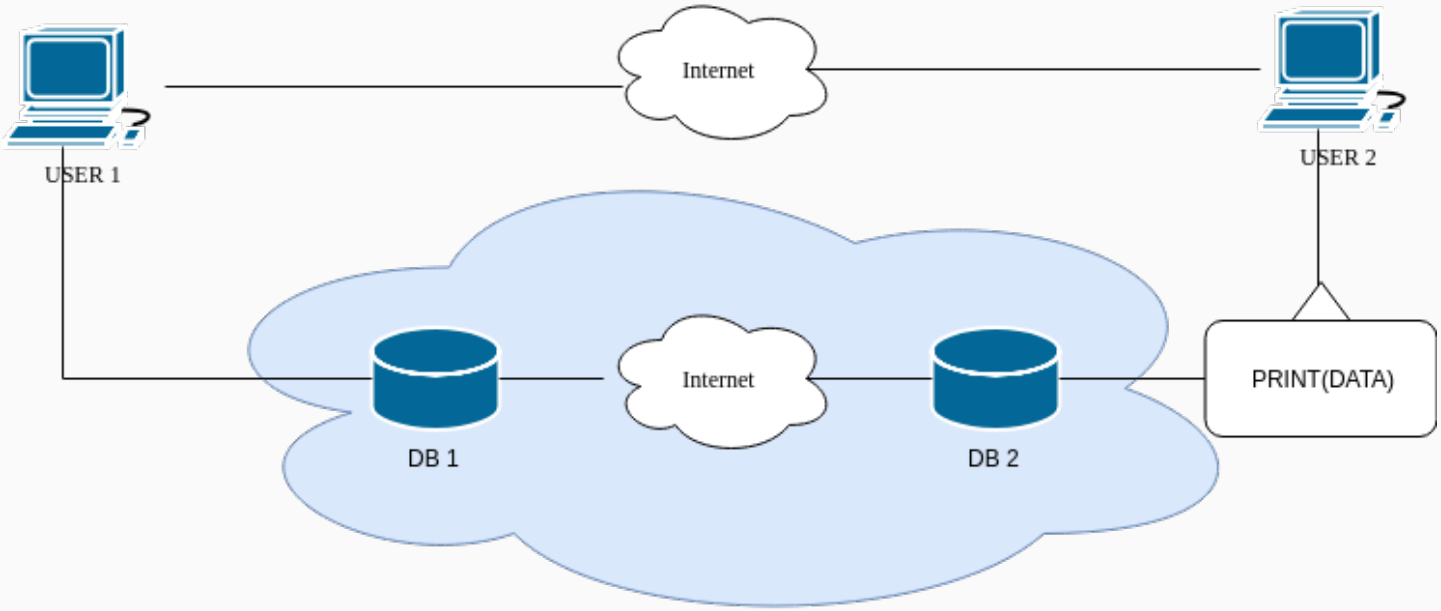
PDV Cloud

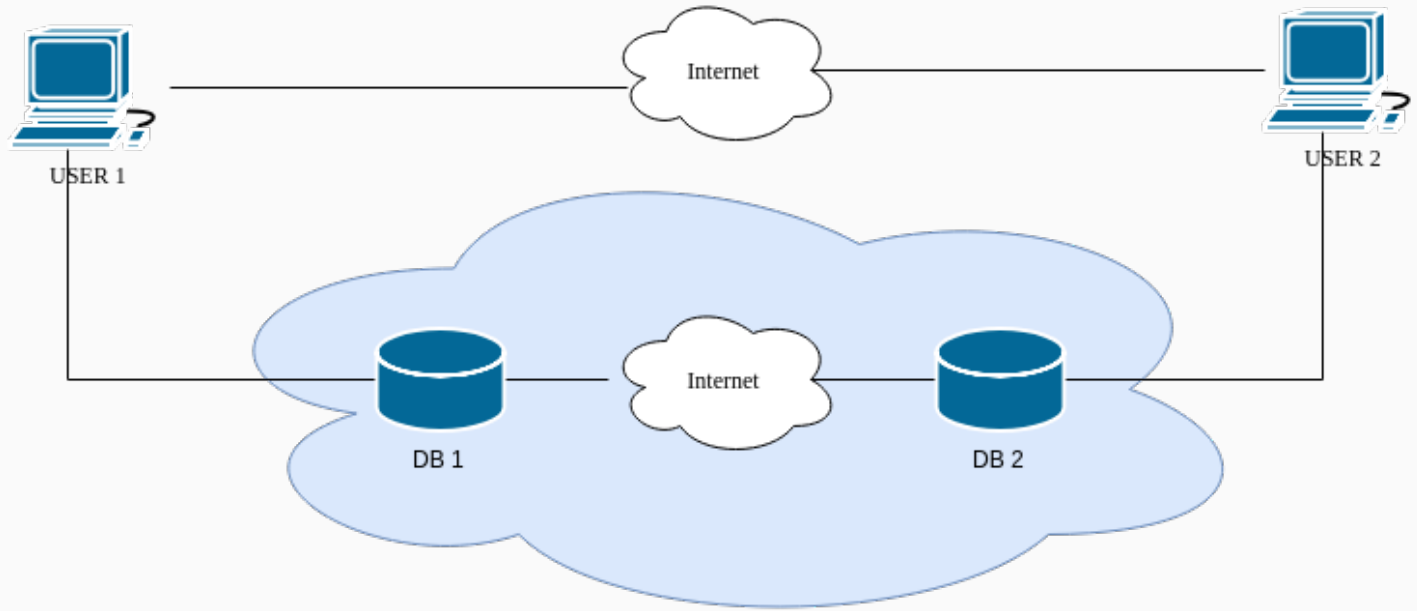


PDV Cloud



PDV Cloud





K čemu bychom zde mohli chtít používat logické hodiny?

Doprogramujte Lamportovy logické hodiny

Doimplementujte logiku Lamportových logických hodin ve třídě `ScalarClock.java`. Následně spusťte scénář `ScalarClockRun.java`.

Doprogramujte Lamportovy logické hodiny

Doimplementujte logiku Lamportových logických hodin ve třídě `ScalarClock.java`. Následně spusťte scénář `ScalarClockRun.java`.

Co je v našem systému špatně?

Doprogramujte Lamportovy logické hodiny

Doimplementujte logiku Lamportových logických hodin ve třídě `ScalarClock.java`. Následně spusťte scénář `ScalarClockRun.java`.

Co je v našem systému špatně?

⚠ Replikace může být pomalá. Druhý klient tak může číst stará data!.

Doprogramujte Lamportovy logické hodiny

Doimplementujte logiku Lamportových logických hodin ve třídě `ScalarClock.java`. Následně spusťte scénář `ScalarClockRun.java`.

Co je v našem systému špatně?

⚠ **Replikace může být pomalá. Druhý klient tak může číst stará data!**.

Jsme to schopní detekovat skalárními hodinami?

Doimplementujte metodu `isCausalityForProcessViolated`

Pak zkuste spustit scénář `ScalarDSConfigBombarding`

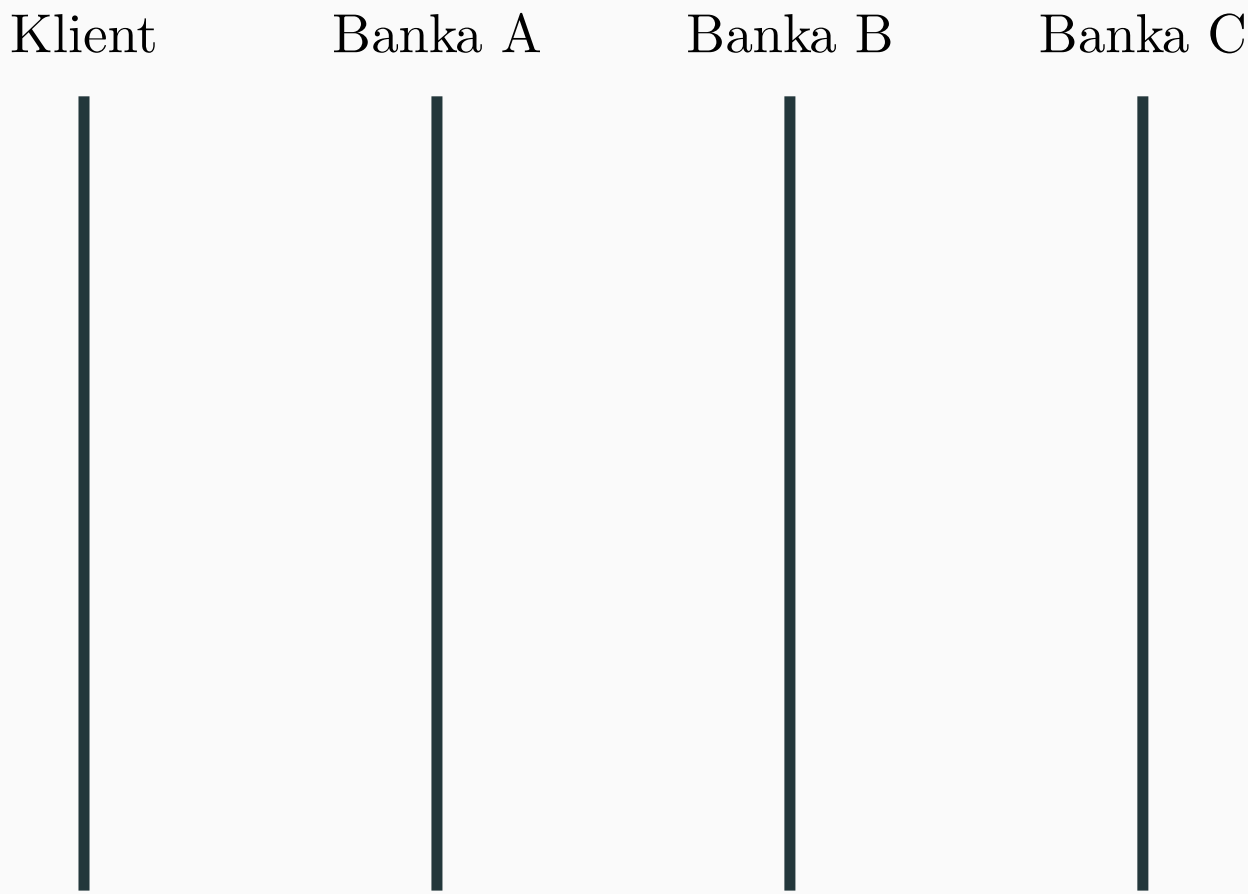
Chceme provést následující dvě operace v daném pořadí:

1. Převést všechny peníze z účtu v bance A na účet v bance B (`transfer_all(A, B)`)
2. Převést všechny peníze z účtu v bance B na účet v bance C (`transfer_all(B, C)`)

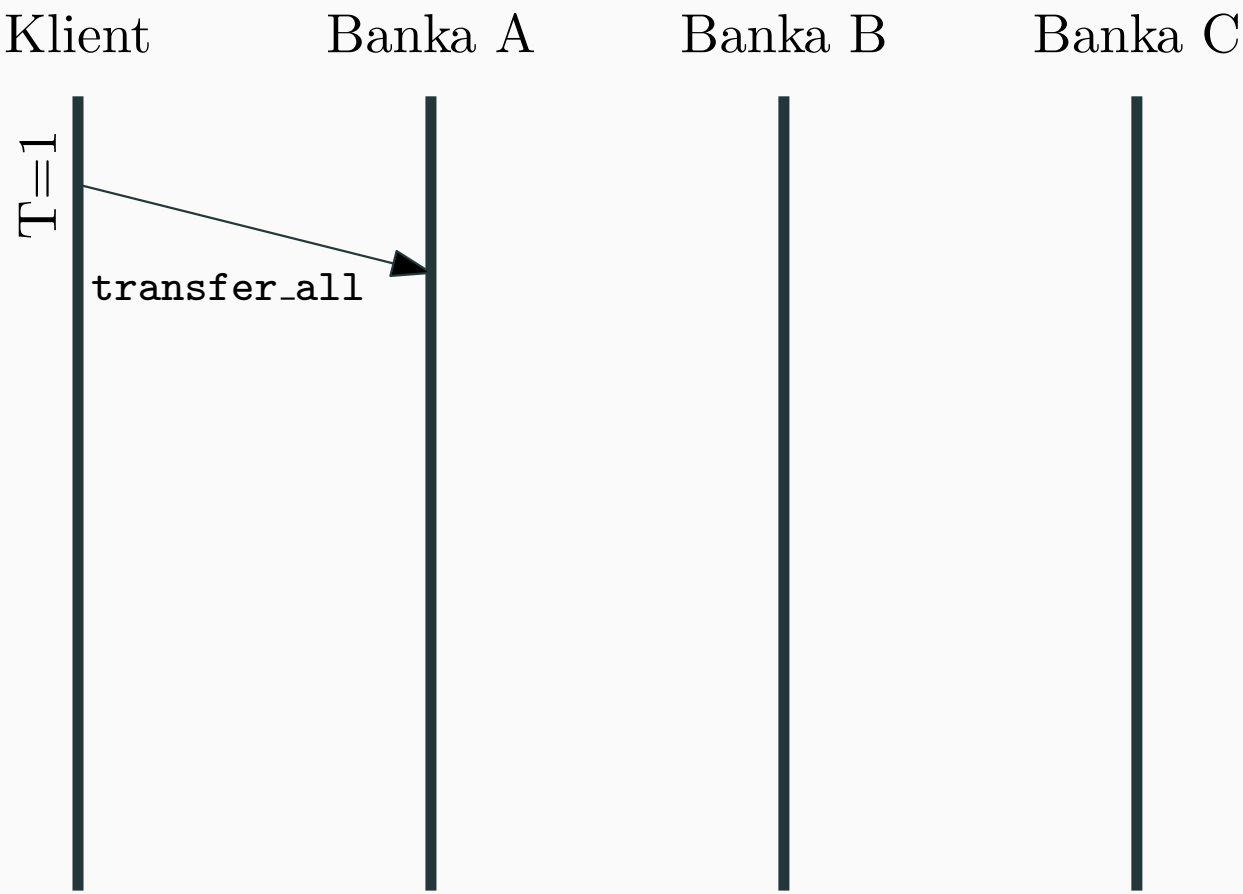
```
void transfer_all(int & from, int & to) {
        to += from;
        from = 0;
}

transfer_all(A, B);
transfer_all(B, C);
```

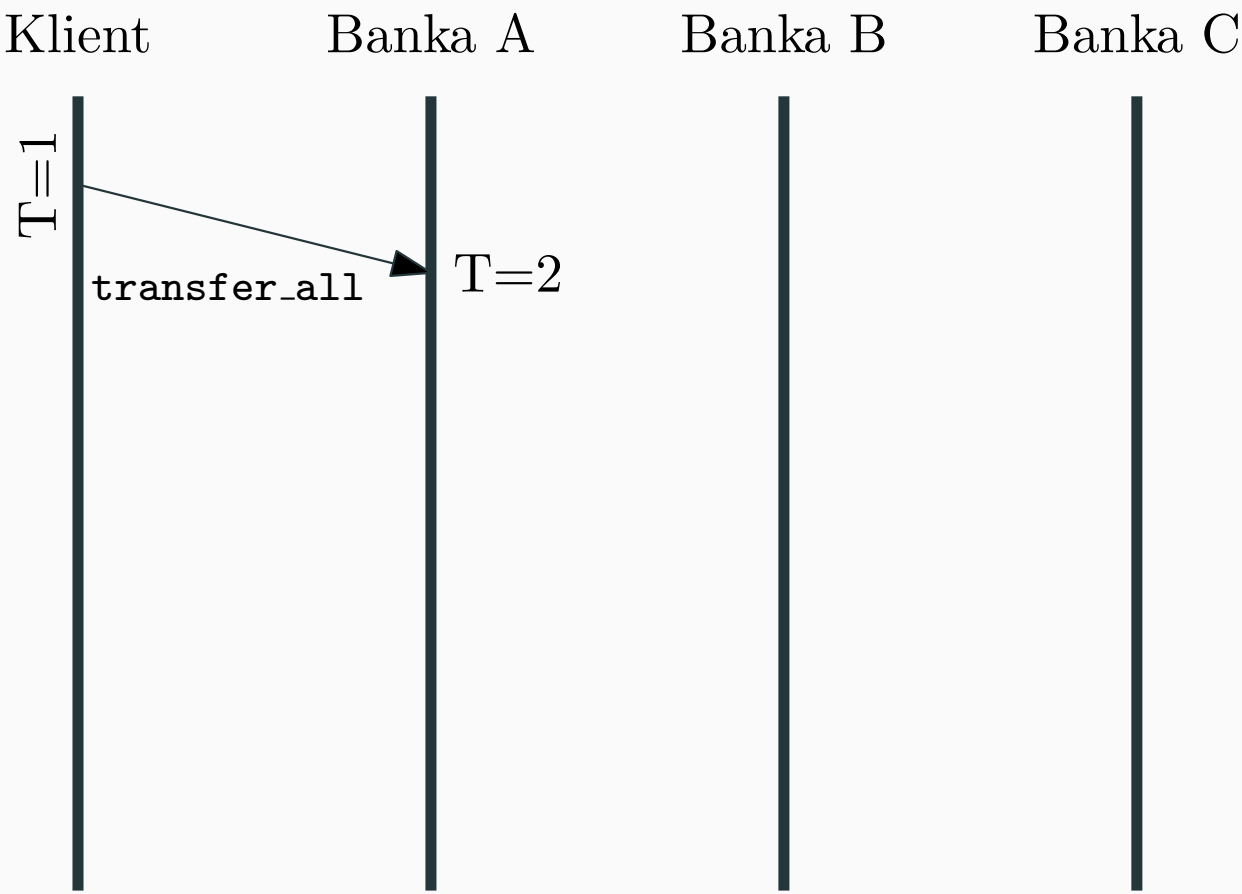
Jak to provést v distribuovaném systému?



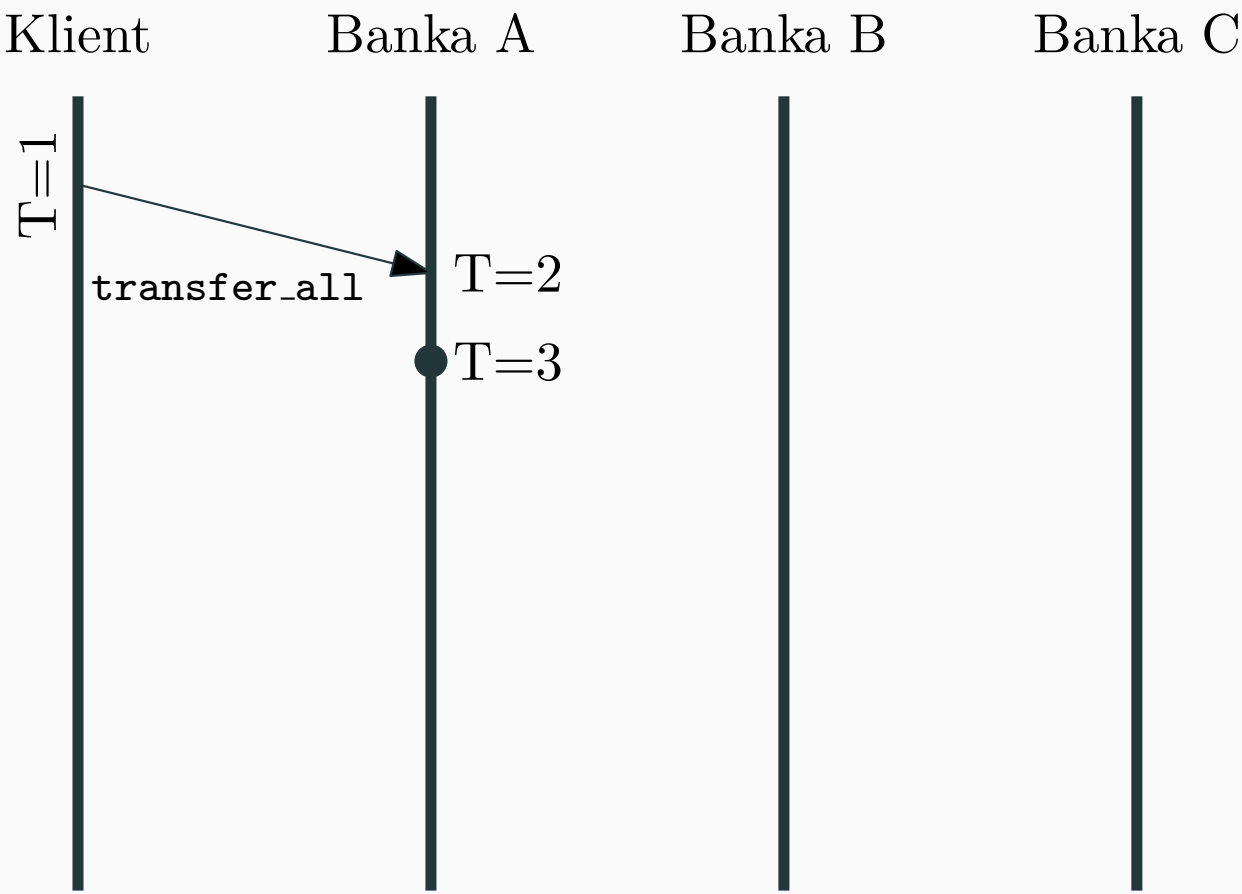
Jak to provést v distribuovaném systému?



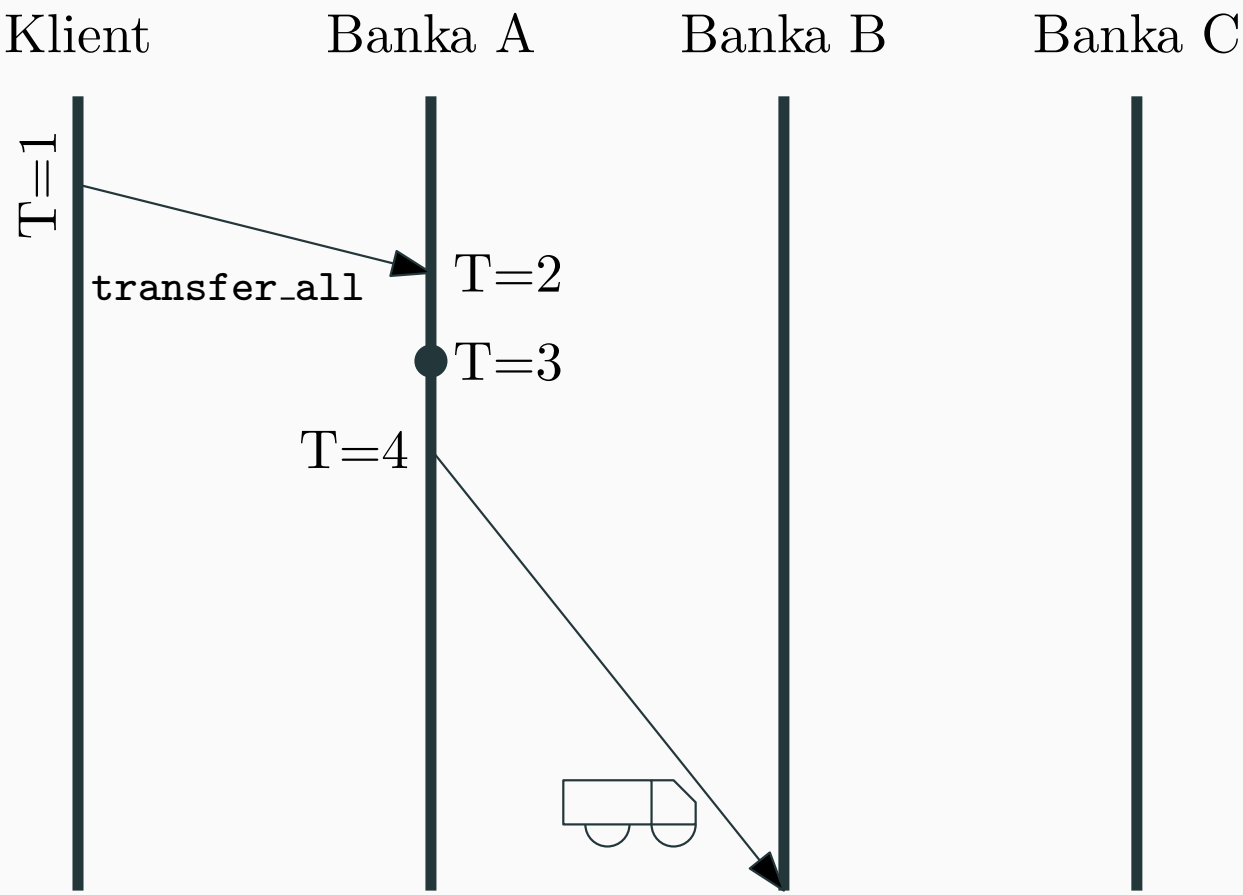
Jak to provést v distribuovaném systému?



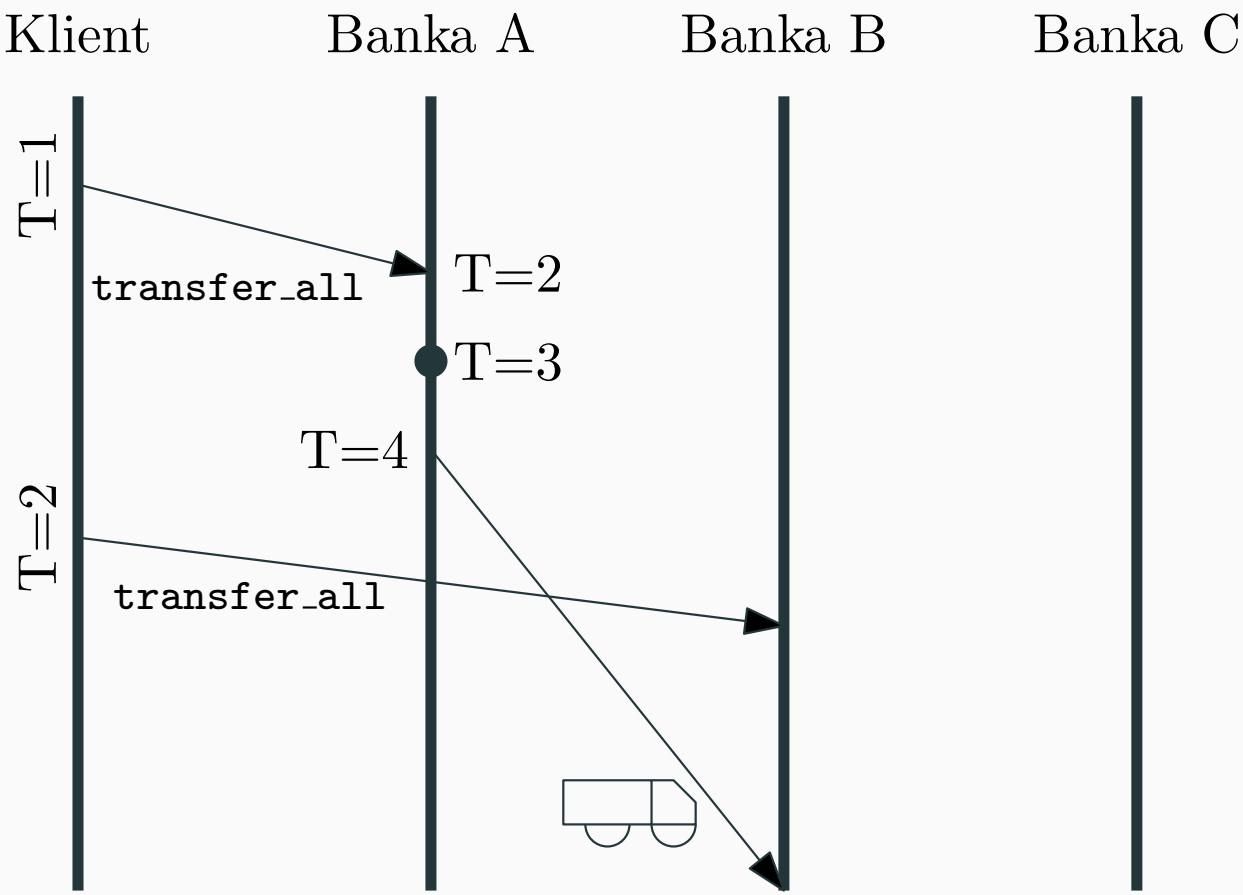
Jak to provést v distribuovaném systému?



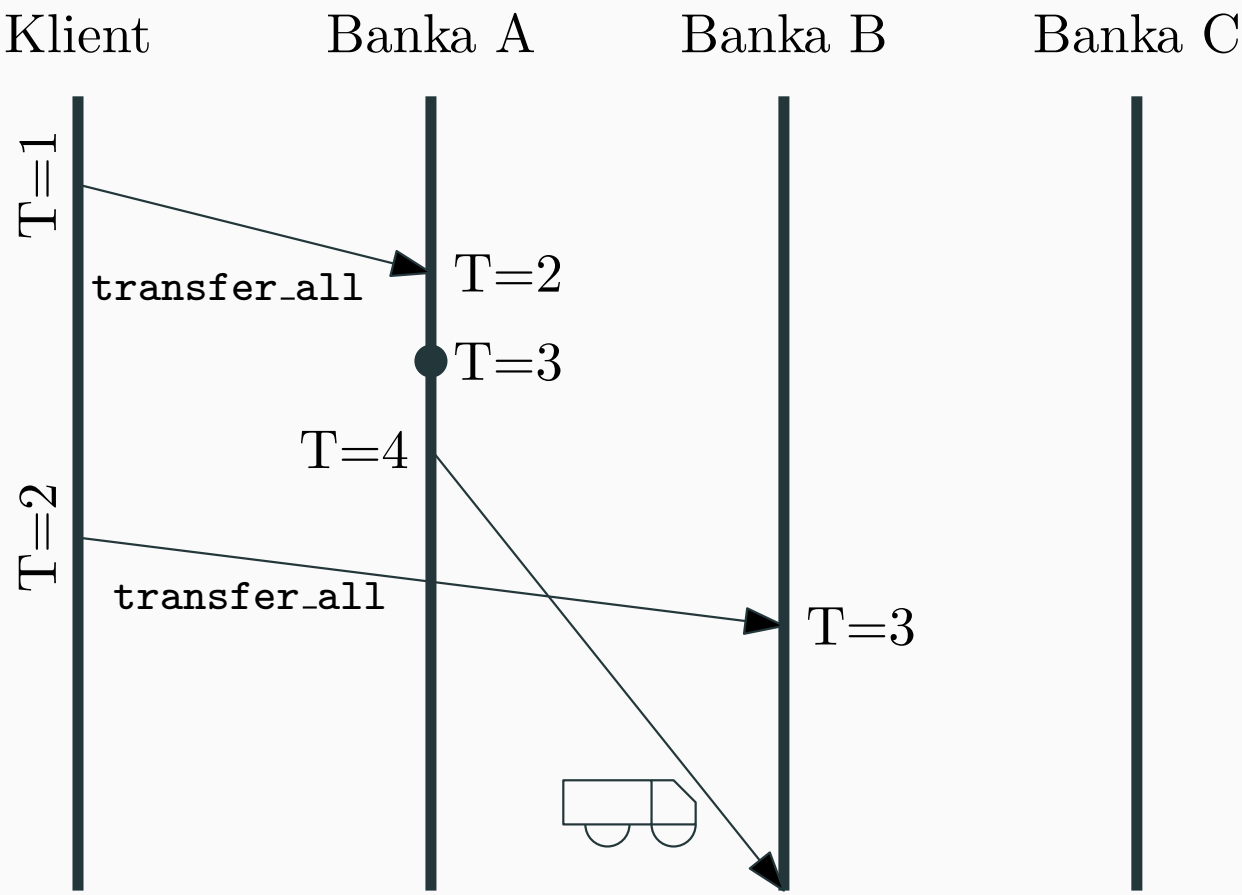
Jak to provést v distribuovaném systému?



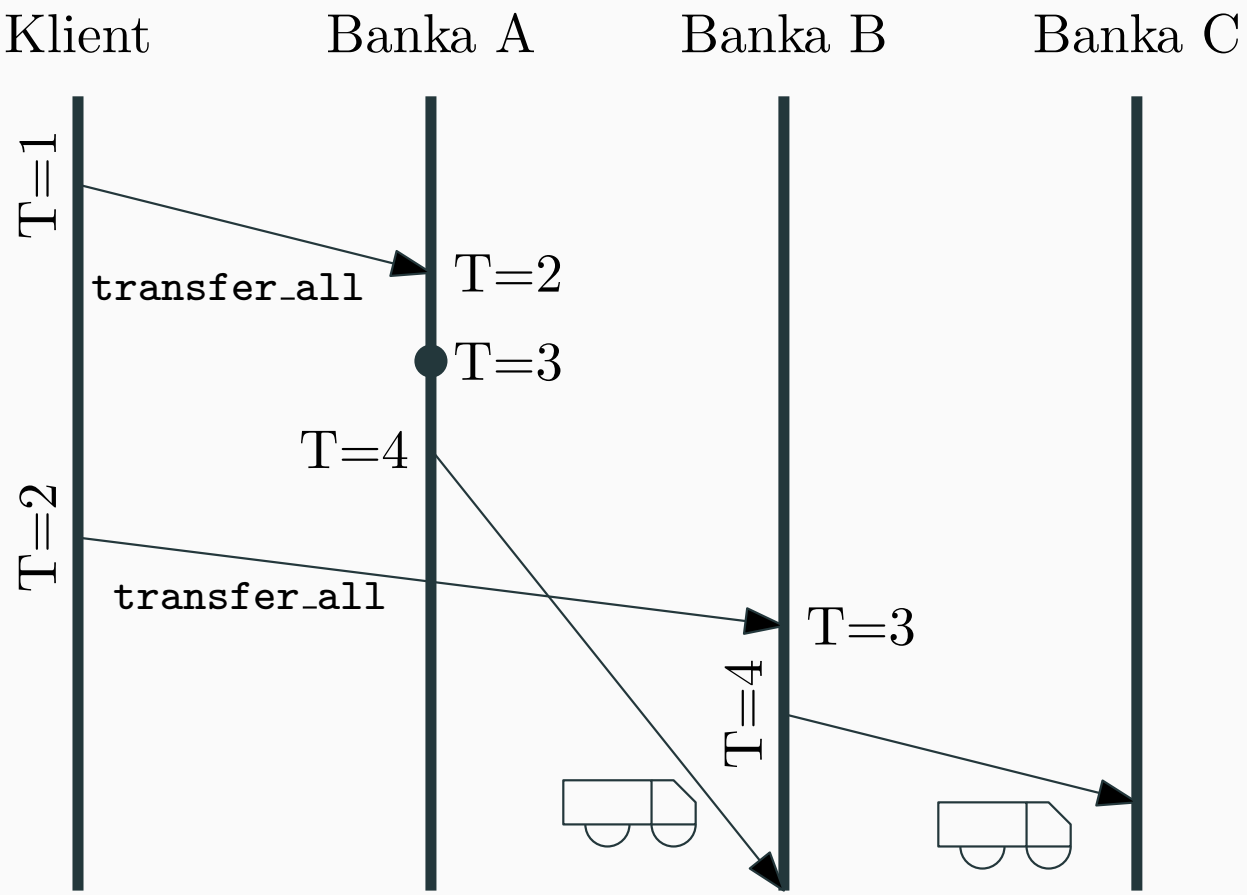
Jak to provést v distribuovaném systému?



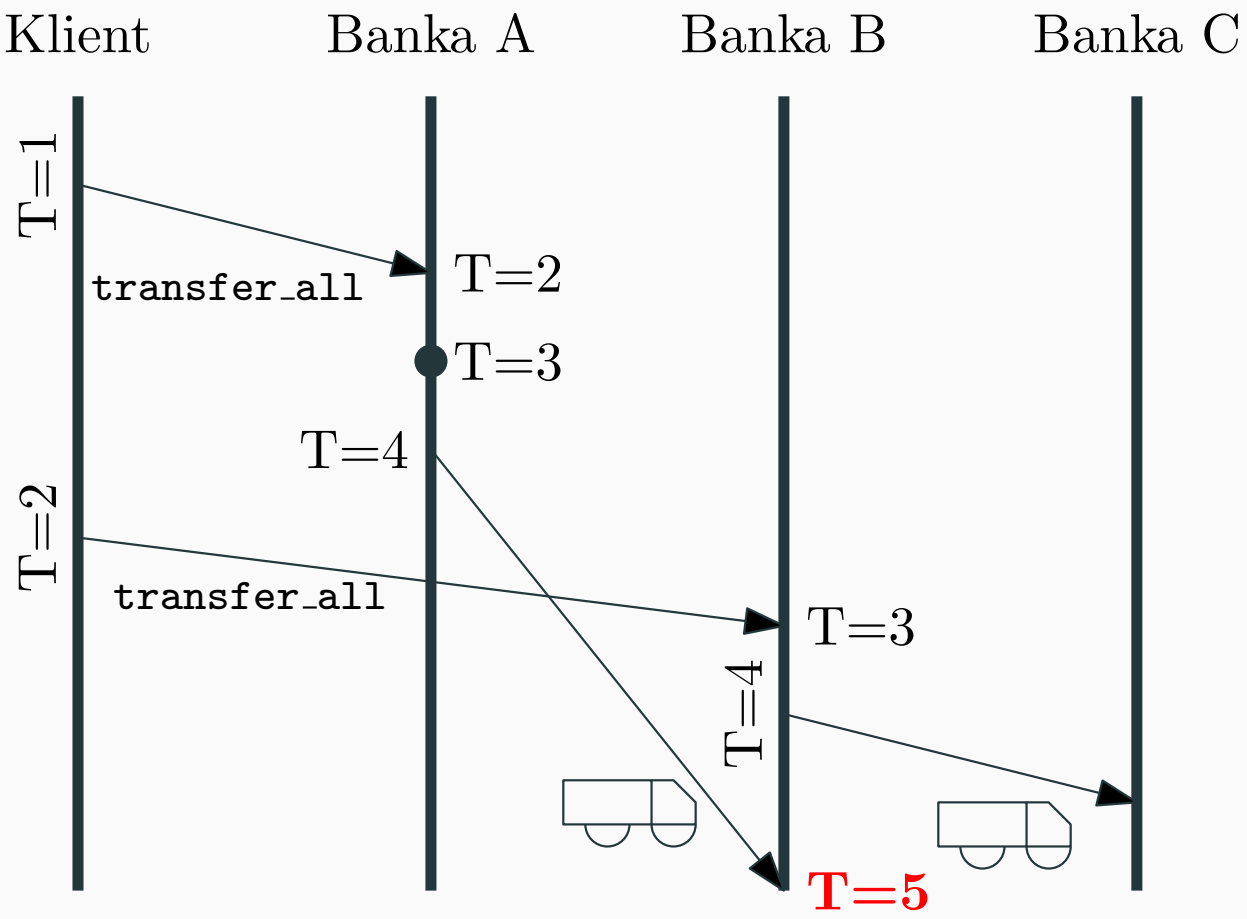
Jak to provést v distribuovaném systému?



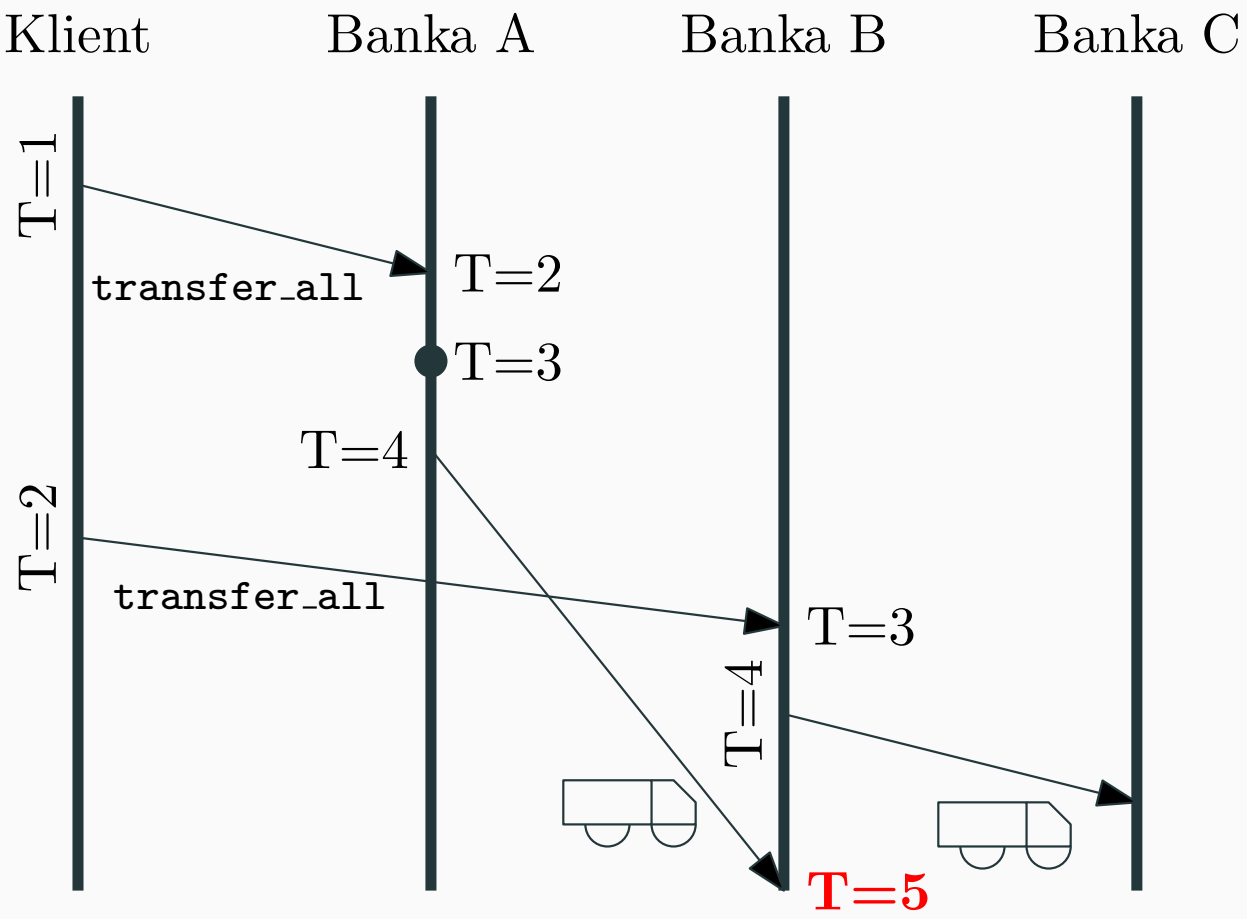
Jak to provést v distribuovaném systému?



Jak to provést v distribuovaném systému?



Jak to provést v distribuovaném systému?



Skalární hodiny agregují všechny události do jediného čísla :-(

Vektorové hodiny

Vektorové hodiny

1. Místo jednoho čísla si držíme vektor časů jednotlivých agentů
   ```
   int[] vectorTime = new int[NUM_AGENTS]
   ```

Vektorové hodiny

1. Místo jednoho čísla si držíme vektor časů jednotlivých agentů
   `int[] vectorTime = new int[NUM_AGENTS]`
2. Před každou významnou událostí (obzvlášť posláním zprávy!) si proces *i* lokální čas posune… **Ale jen svoji komponentu!**
   `++vectorTime[i]`

Vektorové hodiny

1. Místo jednoho čísla si držíme vektor časů jednotlivých agentů
   `int[] vectorTime = new int[NUM_AGENTS]`
2. Před každou významnou událostí (obzvlášť posláním zprávy!) si proces *i* lokální čas posune... **Ale jen svoji komponentu!**
   `++vectorTime[i]`
3. Každé zprávě přiřadíme časovou značku `msg.`T $=$ `vectorTime`

Vektorové hodiny

1. Místo jednoho čísla si držíme vektor časů jednotlivých agentů
   `int[] vectorTime = new int[NUM_AGENTS]`
2. Před každou významnou událostí (obzvlášť posláním zprávy!) si proces i lokální čas posune... **Ale jen svoji komponentu!**
   `++vectorTime[i]`
3. Každé zprávě přiřadíme časovou značku $\texttt{msg.}T = \texttt{vectorTime}$
4. Po přijetí zprávy `msg` procesem i si proces i aktualizuje svůj `logicalTime`:

$$\texttt{vectorTime[j]} = \begin{cases} 1 + \max\{\texttt{vectorTime[j]}, \texttt{msg.}T\texttt{[j]}\} & \text{if } i = j \\ \max\{\texttt{vectorTime[j]}, \texttt{msg.}T\texttt{[j]}\} & \text{jinak} \end{cases}$$

Vektorové hodiny

1. Místo jednoho čísla si držíme vektor časů jednotlivých agentů
   `int[] vectorTime = new int[NUM_AGENTS]`
2. Před každou významnou událostí (obzvlášť posláním zprávy!) si proces i lokální čas posune... **Ale jen svoji komponentu!**
   `++vectorTime[i]`
3. Každé zprávě přiřadíme časovou značku $\texttt{msg.}T = \texttt{vectorTime}$
4. Po přijetí zprávy `msg` procesem i si proces i aktualizuje svůj `logicalTime`:

$$\texttt{vectorTime[j]} = \begin{cases} 1 + \max\{\texttt{vectorTime[j]}, \texttt{msg.}T\texttt{[j]}\} & \text{if } i = j \\ \max\{\texttt{vectorTime[j]}, \texttt{msg.}T\texttt{[j]}\} & \text{jinak} \end{cases}$$

⚠ Vždy posunujeme jen svoji složku časového vektoru!

Doprogramujte vektorové logické hodiny

Doimplementujte logiku vektorových logických hodin ve třídě `VectorClock.java`. Následně spusťte scénář `VectorClockRun.java`.

Doprogramujte vektorové logické hodiny

Doimplementujte logiku vektorových logických hodin ve třídě `VectorClock.java`. Následně spusťte scénář `VectorClockRun.java`.

⚠ Jak využít vektorové logické hodiny k detekci souběžných událostí?.

Jak protokol upravit, aby v nedocházelo k porušení kauzality?

Jak protokol upravit, aby v nedocházelo k porušení kauzality?

Možností je mnoho, například:

Jak protokol upravit, aby v nedocházelo k porušení kauzality?

Možností je mnoho, například:

- Před odesláním REFRESH zprávy si počkat na potvrzení od databáze
  (Odeslání REFRESH zprávy je kauzálním následkem úspěšné replikace)

Jak protokol upravit, aby v nedocházelo k porušení kauzality?

Možností je mnoho, například:

- Před odesláním REFRESH zprávy si počkat na potvrzení od databáze
  (Odeslání REFRESH zprávy je kauzálním následkem úspěšné replikace)
- Pozdržet vyhodnocení dotazu do doby, než replikace proběhne
  (Druhému uživateli můžeme poslat, že má požadovat data zapsaná nejdříve v daném logickém čase)

Jak protokol upravit, aby v nedocházelo k porušení kauzality?

Možností je mnoho, například:

- Před odesláním REFRESH zprávy si počkat na potvrzení od databáze
  (Odeslání REFRESH zprávy je kauzálním následkem úspěšné replikace)
- Pozdržet vyhodnocení dotazu do doby, než replikace proběhne
  (Druhému uživateli můžeme poslat, že má požadovat data zapsaná nejdříve v daném logickém čase)

⚠ **Obecně chceme, aby události e_1, e_2, které mají proběhnout po sobě (tj. například čtení až po replikaci) byly ve vztahu kauzální závislosti.**

Detekce selhání v DS


I have a feeling that something went wrong…
zzz…
by cicakkia '07

Vzpomeňte si na BFS

Co když spící/mrtvá hlava leží na nejkratší cestě?

Detekce selhání

Když nám „umře“ důležitý proces, musíme být schopní se se situací vypořádat. (Jinak nám celý DS zhavaruje)

Detekce selhání

Když nám „umře“ důležitý proces, musíme být schopní se se situací vypořádat. (Jinak nám celý DS zhavaruje)

⚠ Ale to musíme nejdřív zjistit, že „umřel“!

Detekce selhání

Když nám „umře“ důležitý proces, musíme být schopní se se situací vypořádat. (Jinak nám celý DS zhavaruje)

⚠ Ale to musíme nejdřív zjistit, že „umřel“!

Jak na to?

Algoritmy na principu *heart-beats*

- Hearbeats jsou odesílány periodicky (každých T „vteřin“)
- Nedostane-li proces heartbeat od procesu p_j po dobu $T + \tau$ „vteřin“, považuje p_j za mrtvý

Algoritmy na principu *heart-beats*

- Hearbeats jsou odesílány periodicky (každých T „vteřin“)
- Nedostane-li proces heartbeat od procesu p_j po dobu $T + \tau$ „vteřin“, považuje p_j za mrtvý

- Centralizovaný heartbeat
- Kruhový heartbeat
- Všichni-všem (all-to-all) heartbeating

Algoritmy na principu *heart-beats*

- Hearbeats jsou odesílány periodicky (každých T „vteřin“)
- Nedostane-li proces heartbeat od procesu p_j po dobu $T + \tau$ „vteřin“, považuje p_j za mrtvý

- Centralizovaný heartbeat
- Kruhový heartbeat
- Všichni-všem (all-to-all) heartbeating

Doprogramujte detekování selhání procesu na základě (all-to-all) heartbeating

Doimplementujte logiku detekování selhání procesu na základě (all-to-all) heartbeating v `DetectorProcess.java`. Následně spusťte scénář `MainFD.java`, ve kterém máte zajištěno, že selže právě jeden proces.

Vlastnosti detektorů

Vlastnosti detektorů

- **Úplnost**: každé selhání je časem detekováno aspoň jedním funkčním procesem

Vlastnosti detektorů

- Úplnost: každé selhání je časem detekováno aspoň jedním funkčním procesem
- Přesnost: nedochází k označení funkčního procesu za havarovaný

Vlastnosti detektorů

- Úplnost: každé selhání je časem detekováno aspoň jedním funkčním procesem
- Přesnost: nedochází k označení funkčního procesu za havarovaný
- Rychlost: čas do okamžiku, kdy první proces detekuje selhání

Vlastnosti detektorů

- Úplnost: každé selhání je časem detekováno aspoň jedním funkčním procesem
- Přesnost: nedochází k označení funkčního procesu za havarovaný
- Rychlost: čas do okamžiku, kdy první proces detekuje selhání
- Škálovatelnost: ani při velkém počtu agentů nedojde k zahlcení systému

Vlastnosti detektorů

- **Úplnost**: každé selhání je časem detekováno aspoň jedním funkčním procesem
- **Přesnost**: nedochází k označení funkčního procesu za havarovaný
- **Rychlost**: čas do okamžiku, kdy první proces detekuje selhání
- **Škálovatelnost**: ani při velkém počtu agentů nedojde k zahlcení systému

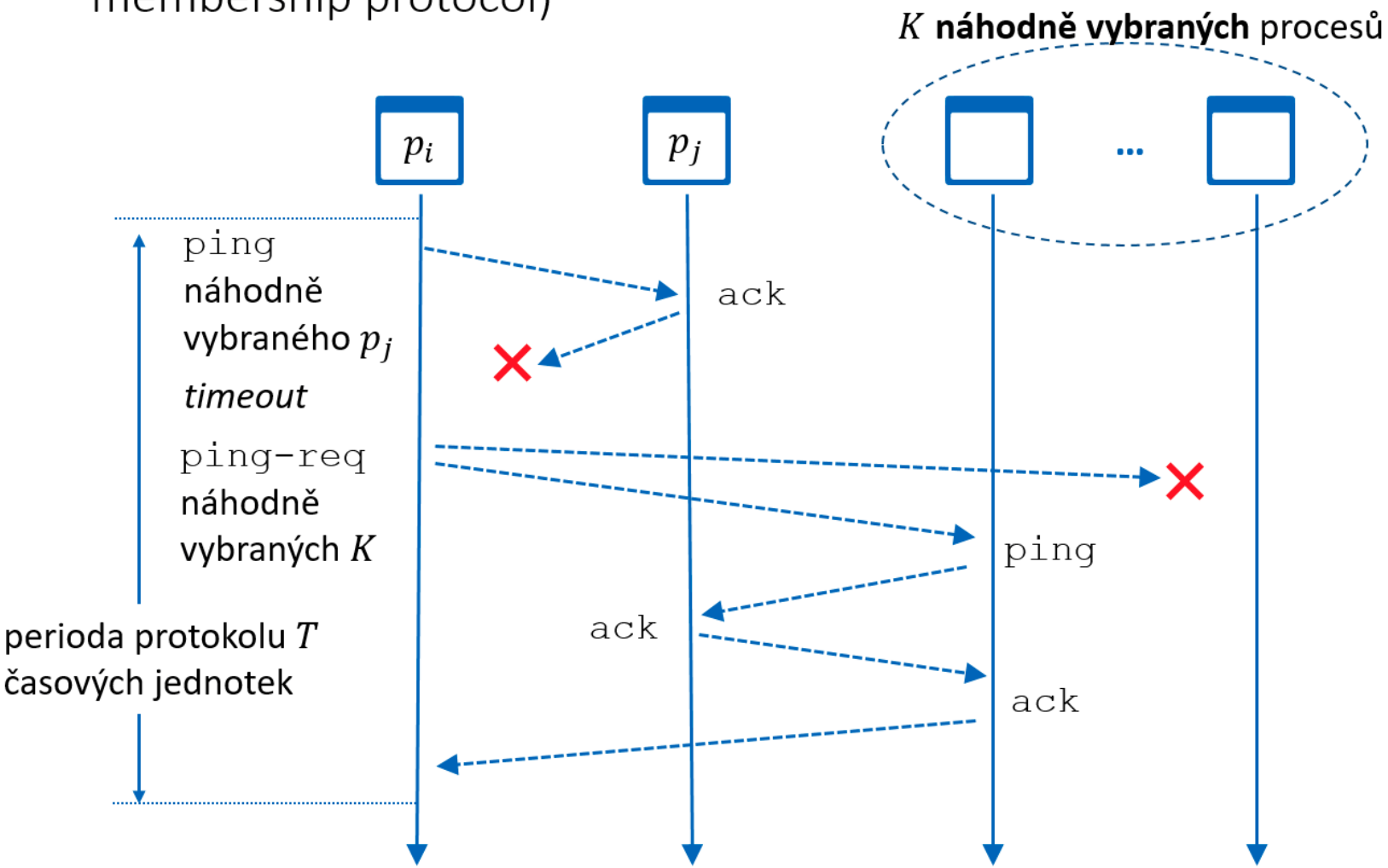
All-to-all heartbeating:

:-(Přesnost

:-(Škálovatelnost

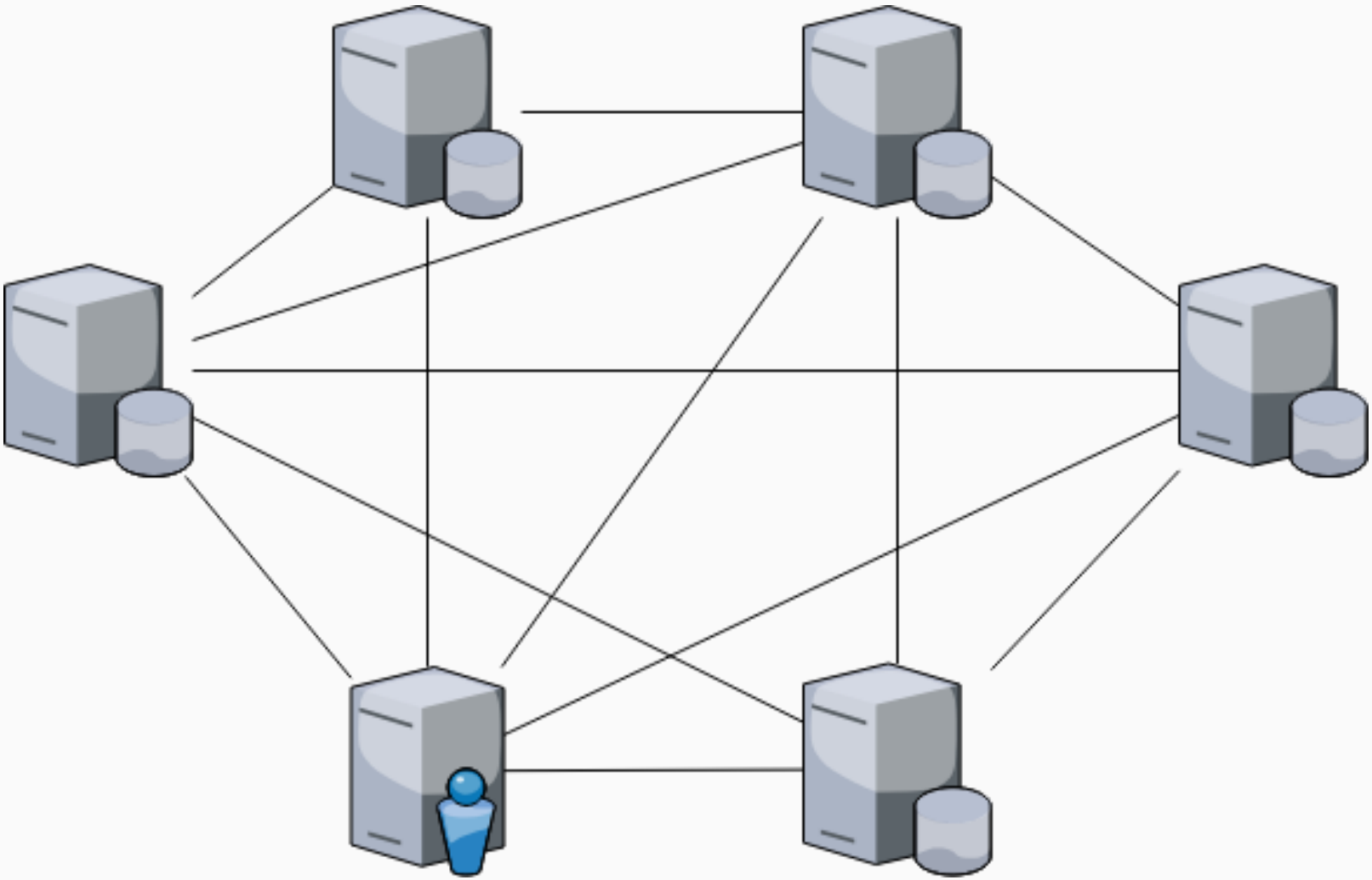
SWIM Failure Detector

(Scalable weakly consistent infection-style proces group membership protocol)



Zadání domácí úlohy

Distribuovaná databáze na komoditních serverech



Distribuovaná databáze na komoditních serverech

Naimplementujte SWIM detekci selhání a zajistěte, že

1. zbytečně nevytěžuje síť;
2. detekuje všechny “mrtvé” procesy s rozumnou rychlostí; a
3. je dostatečně přesné.

Zpracování musí být **distribuované**, procesy si nesahají vzájemně do paměti!

Díky za pozornost!

Budeme rádi za Vaši zpětnou vazbu! →

http://bit.ly/2IFPLQ3