KATEDRA
INFORMATIKY
UNIVERZITA PALACKÉHO V OLOMOUCI

KMI/PDS - Paralelní a distribuované systémy

Poznámky z výuky *(2025 – 2026)*
Verze z *13. ledna 2026*

Vojtěch Netrh
vojtanetrh@gmail.com

Obsah

[1] Obvykle bezdrátová síť bez centrálního bodu a s proměnlivou topologií

1 Obecný úvod

Dřív tenhle kurz učil RNDr. Martin Trnečka, Ph.D., takže se dá čerpat i z jeho materiálů na webu. Jednak tam jsou prezentace, případně pak i cvičení a nějaký alespoň úvodní kód v Pythonu.

Jednoduchá knihovna pro simulaci distribuovaných systému od Tomáše Mikuly https://github.com/mikulatomas/distsim

Python používám pro ukázky kódu jelikož se nejvíc blíží pseudokódu a zároveň je u něj možné mít hezky zvýrazněnou syntaxi (konkrétní díky balíčku `codly`).

Seznam bodů z jednotlivých úkolů je ve sdílené tabulce.

1.1 Zkouškové otázky

1. Algoritmy pro kritickou sekci.
2. Základní synchronizační primitiva a jejich použití.
3. Prostředky pro synchronizaci vláken.
4. Prostředky pro synchronizaci procesů.
5. Koordinace času v DS.
6. Vzájemné vyloučení v DS.
7. Volba lídra v DS.
8. Shoda v DS.
9. Tolerance chyby v DS.
10. Globální stav v DS a distribuovaný commit.
11. Replikace a konzistence v DS.
12. Blockchain.

1.2 Články ke zkoušce

[Na webu jsou na ně odkazy, tady je jen seznam o jaké jde.]

1. MapReduce: Simplified Data Processing on Large Clusters
2. ZooKeeper: Wait-free coordination for Internet-scale systems
3. In Search of an Understandable Consensus Algorithm (Extended Version)
4. Practical Byzantine Fault Tolerance
5. Bitcoin: A Peer-to-Peer Electronic Cash System

1.3 Pár pojmů na začátek

- Sekvenční
- Paralelní
- Distribuované

- **Cluster** … více strojů v síti (typicky blízko sebe); schované za API
- **Superpočítač** … cluster s velmi rychlým propojením
- **Grid computing** … více strojů v síti; volnější než cluster
- **Cloud** … transparentní přístup HW (IaaS), SW (SaaS) i platformě (PaaS)

1.4 Flynnova taxonomie

- **S** … single
- **M** … multiple
- **I** … instruction
- **D** … data

V průběhu času různé varianty a také u různých věcí – SISD (von Neuman), SIMD (vektorové instrukce, grafiky), MISD (neuronové sítě) a MIMD (dnes už vše). A vlastně ani to MIMD už dnes nestačí.

Existují dále různé nástavby, případně jemnější dělení – SMP, DSM, SPMD, …

1.5 Kdy a proč to dělat?

Na úvod jedno důležité pravidlo: **pokud to není třeba, tak neparalelizovat!** Dnes ale skoro vždy je nutné. Přináší výhody či problémy v řadě oblastí – výkon, spolehlivost, bezpečnost, dostupnost…

1.5.1 Škálování

1. Up (vertikální) – lepší CPU, rychlejší disk, více paměti, …
2. Out (horizontální) – více strojů

1.6 Zrychlení výpočtů

Existuje několik možných pohledů. Ke každému z nich je možno udělat nějaký hezký graf, případně reprezentovat i matematickým zápisem. Obecně jsou pro tyto zákony nějaké limity, na které naráží (fyzikální, podle velikosti problému či specifických částí problému).

1. Moorův zákon – každý 1,5 roku se výkon procesorů zdvojnásobí
2. Amdahlův zákon – použijeme více procesorů pro řešení stejného problému
3. Gustafson-Barsisův zákon – když máme více zdrojů, můžeme problém řešit detailněji

2 Paralelní systémy

Tahle část by měla vydat cca na 3-4 přednášky.

Mohou nastat takové 3 základní problémy:

1. chyba souběhu,
2. uváznutí (deadlock a livelock),
3. vyhladovění.

Paralelní program **Definice 1**

Konečná množina sekvenčních procesů, kde každý proces sekvenčně vykonává atomické operace.

Těmi atomickýcmi operacemi se přechází mezi různými stavy (hodnoty proměnných, aktuální instrukce).

2.1 Synchronizace

Vpodstatě jediný úkol **synchronizovat** = vyřešit omezení na možné scénáře/plánky, které jsou korektní. K tomu slouží několik možných nástrojů – atomické operace, složené atomické

operace[2] , synchronizační primitiva (semafor, zámek, bariéra, …). Scénáře se dají zobecňovat a podle daného vzoru vyřešit (není potřeba vymýšlet nic nového).

Atomická proměnná **Definice 2**

Atomická proměnná je taková, kterou lze atomicky upravit.

Volatilní proměnná **Definice 3**

Volatilní proměnná je taková, která má vždy poslední hodnotu (tzn. změna není jen v cache).

2.2 Vlastnosti programu

Klasické debugování je nevhodné. Projít všechny možné scénáře je totiž nereálne, kvůli jejich velikému množství. Pomůže nám tedy (modální a temporální) logika, prekondice, postkondice a Hoareho logika (viz co už jsme řešili v PP4).

Máme 2 základní typy vlastností paralelního programu – živost (tvrzení platné pro alespoň 1 stav výpočtu) a bezpečnost (tvrzení platné pro všechny stavy výpočtu).

Důležitá vlastnost pro prostředí/plánovač je **férovost** (to už taky známe z PP4). Plánovač je férový pokud jsou všechny scénáře férové (čili pokud se proces chystá vykonat operaci, musí se ve scénáři objevit). Rozlišujeme 2 typy férovosti.

slabá férovost = akce, která vždy může nastat, někdy nastane (bez dalšího omezení)
silná férovost = akce, která někdy může nastat, někdy nastane (je tam nějaké omezení)

2.3 Kritická sekce

Kritická reference **Poznámka 4**

Výskyt proměnné je kritická reference pokud do ní zapisuje jeden proces a čte ji jiný proces.[3]

Problém kritické sekce, je část programu kde program pracuje se sdílenými zdroji a musíme vyřešit, aby nedocházelo k problémům.

Příkaz `await` **Poznámka 5**

Příkaz **`await`** je (pasivní) čekání na splnění podmínky (neboli přerušení).

Požadavky pro korektní vyřešení kritické sekce jsou:

1. **vzájemné vyloučení** – max 1 proces v kritické sekci
2. **absence uváznutí** – jestliže se nějaké procesy snaží současně vstoupit do kritické sekce, pak jeden z nich musí někdy uspět
3. **absence vyhladovění** – pokud se proces snaží vstoupit do kritické sekce, jednou musí uspět

Vyřešil ho E. W. Dijkstra. Je to n procesů, které ve smyčce vykonávají posloupnost akcí rozdělenou na kritickou a nekritickou sekci. Synchronizací v podstatě zajistíme korektnost.

[2] atomicity a její konkrétní definice se může lišit na základě konkrétního systému/programovacím jazyku.
[3] Jeden příkaz může obsahovat více kritických referencí

Dobré podívat se na návrhy řešení v prezentaci 02. Především ty co jsou nevhodné.

Pro řešení kticiké sekce se dají použít obecné algoritmy uvedené dále.

2.3.1 Dekkerův algoritmus

Hlídá právo na vstup (**`turn`**) a žádá o vstup (**`want`**) s možností vzdát se.

Sdílená paměť

wantA, wantB ← false
turn ← A

A	**B**
opakuj:	opakuj:
A_1: nekritická sekce	B_1: nekritická sekce
A_2: wantA ← true	B_2: wantB ← true
A_3: while wantB	B_3: while wantA
A_4: if turn = B	B_4: if turn = A
A_5: wantA ← false	B_5: wantB ← false
A_6: await turn = A	B_6: await turn = B
A_7: wantA ← true	B_7: wantB ← true
A_8: kritická sekce	B_8: kritická sekce
A_9: turn ← B	B_9: turn ← A
A_{10}: wantA ← false	B_{10}: wantB ← false

Obrázek 1: Dekkerův algoritmus.

2.3.2 Petersonův algoritmus (Tie-breaker)

Vychází z Dekkerova algoritmu, ale cyklus s **`await`** nahrazen **`await`** se složenou podmínkou.

Sdílená paměť

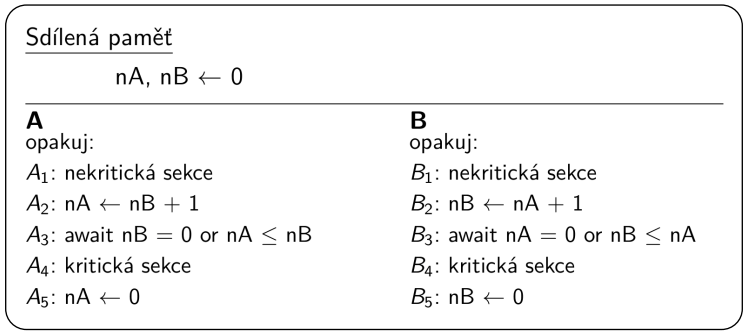
wantA, wantB ← false
turn ← A

A	**B**
opakuj:	opakuj:
A_1: nekritická sekce	B_1: nekritická sekce
A_2: wantA ← true	B_2: wantB ← true
A_3: turn ← A	B_3: turn ← B
A_4:	B_4:
await not(wantB and turn=B)	await not(wantA and turn=A)
A_5: kritická sekce	B_5: kritická sekce
A_6: wantA ← false	B_6: wantB ← false

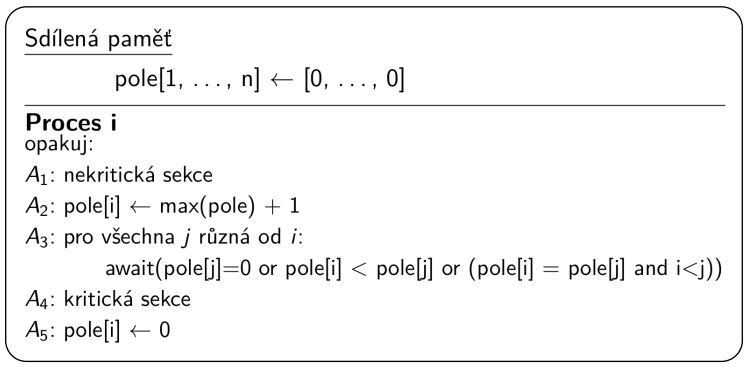
Obrázek 2: Petersonův algoritmus.

2.3.3 Bakery algoritmus

Simulace chování v pekárně, kde se čeká na lístky na chleba. Velmi hezké řešení, ale pomalé.



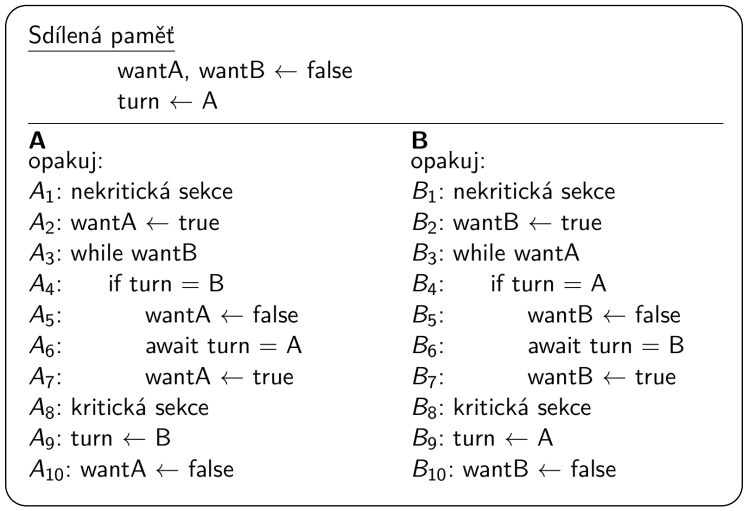
Obrázek 3: Bakery algoritmus (základní).



Obrázek 4: Bakery algoritmus pro n procesů.

2.3.4 Lamportův Bakery algoritmus

Oproti předchozímu se vždy ví o případných kolizích, kde rozhoduje **`id`** procesu. Skutečně lze použít (např. implementace vzájemného vyloučení tam kde HW nenabízí synchronizační primitiva).



Obrázek 5: Lamportův Bakery algoritmus pro n procesů.

Formalizace toho kdy lidé čekají ve frontě na chleba s očíslovanými lístečky. Buď můj lísteček má nejmenší číslo nebo tam jsem jediný. Velmi elegantní řešení. Nutné každé vlákno nějak jednoznačně (ideálně nějaké ID) pro případné porovnání těch vláken (poslední podmínka v tom algoritmu).

2.3.5 Další algoritmy

Existují i další algoritmy, které se dále používají, ale jsou složitější. Algoritmy se reálně neimplementují, ale používají se již implementované nebo synchronizační primitiva.

2.4 Synchronizační primitiva

2.4.1 Složené atomické akce

Jsou to „nižší“ operace než synchronizační primitiva typu `lock` nebo `semaphore`. Existuje jich několik různých (my jich probereme 5), dělají zhruba to samé. Je k nim možné napsat co dělají (nejlíp formou pseudokódu). Důležité si uvědomit, že se to provede **celé bez přerušení**.

1. `test-and-set`
2. `swap`
3. `fetch-and-add` – použili bychom čítač a frontu kdo je na řadě
4. `compare-and-swap`
5. `load-link` / `store-conditional`

```cpp
bool lock = false
while(testAndSet(&lock)){
  // empty cycle
}

// --- KRITICKÁ SEKCE ---
lock = false
```

```cpp
int lock = 0; // 0 = volno, 1 = obsazeno

while (!CompareAndSwap(&lock, 0, 1)) {
    // Čekáme, dokud lock není 0
}

// --- KRITICKÁ SEKCE ---
lock = 0;
```

Kritická sekce za pomocí `compare-and-swap`.

2.4.2 Zámek (lock, mutex)

Nejjednodušší synchronizační primitivum. Může vzniknout problém s přehnanou synchronizací.

```python
nekritická sekce
lock.lock()
kritická sekce
lock.unlock()
```

2.4.3 Semafor

Má stavy 0 až n. Poskytuje 2 operace: *čekat (dekrementace)* P a *signalizace (inkrementace)* V. Funguje jako chráněný čítač za pomocí zámku. Pro implementaci není vyžadováno aktivní čekání. Férovost vyřešíme tak, že čekající procesy jsou ve frontě.

Jako nejjednodušší binární semafor můžeme brát zámek.

```python
semaphore = 1   # sdílená paměť

nekritická sekce
semaphore.wait()
kritická sekce
semaphore.signal()
```

2.4.4 Monitor

Nejvíce strukturovaný synchronizační nástroj. V dost jazycích se nevyskytuje (má ho `Java` a `C#`, třeba `Python` ne). Kód kritické sekce se schová do monitoru (velmi jednoduché řešení).

```python
m = monitor():
  operation criticalOperation():
    kritická sekce

nekritická sekce
m.criticalOperation()
```

2.4.5 Podmíněná proměnná

`waitC` vždy zablokuje a čekám až mě někdo zavolá

2.4.6 Bariéra

Místo v programu, kde se musí sejít více procesů a až poté jdou společně dále. Více možností jak to udělat (lineární, stromová, motýlová).

2.4.7 Threadpool

Nějaká skupina (pool) vláken, která se přiřazují na vyžádání. Když to vlákno je volné, tak se vyčistí a předá se mu nový úkol. Obvykle je totiž méně vláken než úkolů.

2.5 Synchronizační problémy

Typické problémy vyskytující skrz různé praktická aplikace, primárně v operačních systémech. Problémy i jejich řešení je obecné, takže se dá napasovat na různé konkrétní příklady a není nutné nic vymýšlet znovu.

2.5.1 Producent a konzument

Často se vyskytující problém, ve kterém máme 2 typy procesů – *producenta* (produkuje data) a *konzumenta* (sbírá a zpracovává data). Obvykle je od obou typů více instancí. **Komunikace probíhá přes buffer**, který má danou velikost n (teoreticky by mohl být nekonečný a některé věci by byly jednodušší, ale praxe taková není). Pro každou položku v bufferu platí, že je **vyprodukována a zkonzumována právě jednou**.

Příklady jsou: event loop, stream, zpracování výsledků, které jsou produkovány průběžně. V praxi třeba tzv. *event driven systems*, kde program musí na něco reagovat (třeba stisk klávesy uživatele).

Pro podmínky řešení platí některé zásady. Pokud někdo manipuluje (dodává nebo čte) data z bufferu, tka buffer není v konzistentním stavu $\Rightarrow$ **buffer musí být ve vzájemném vyloučení**. Pokud je buffer prázdný a konzument chce data přečíst, musí počkat (obdobně u producenta).

```python
# producent
event = waitForEvent()
mutex.wait()
  buffer.add(event)
  items.signal()
mutex.signal()
```

```python
# konzument
items.wait()
mutex.wait()
  event = buffer.get()
mutex.signal()
event.process()
```

Výpis 1: Řešení producent a konzument s nekonečným bufferem

```python
# inicializace
mutex = Semaphore(1)
items = Semaphore(0)
spaces = Semaphore(buffer.size())
```

```python
# producent
event = waitForEvent()

spaces.wait()
mutex.wait()
  buffer.add(event)
mutex.signal()
items.signal()
```

```python
# konzument
items.wait()
mutex.wait()
  event = buffer.get()
mutex.signal()
spaces.signal()

event.process()
```

Výpis 2: Řešení producent a konzument s bufferem o velikosti n

2.5.2 Čtenáři a písaři

Opět jsou 2 typy procesů – *písaři* (zapisují do nějakého místa) a *čtenáři* (z onoho místa čtou). Písaři musí psát ve vzájemném vyloučení, ale čtenáři mohou číst zároveň a každý písař je ve vzájemném vyloučení se všemi čtenáři.

Příklady: přístup k databázi, souboru nebo datové struktuře.

```python
# inicializace
readers = 0                 # počet čtenářů v místnosti
mutex = Semaphore(1)
roomEmpty = Semaphore (1)   # 1 pokud v místnosti nikdo není, 0 jinak
```

```python
# písaři
roomEmpty.wait()
  # KRITICKÁ SEKCE pro písaře
roomEmpty.signal()
```

```python
#čtenáři
mutex.wait()
  readers += 1
  if readers == 1:
    roomEmpty.wait() # kontroluje a zamyká jen první
mutex.signal()

# KRITICKÁ SEKCE pro čtenáře

mutex.wait()
```

```
    readers -= 1
    if readers == 0:
      roomEmpty.signal() # poslední co odchází odemkne
  mutex.signal()
```

Analogií pro speciální pravidla pro prvního a posledního čtenáře může být skupina lidí co vchází do místnosti, kde je tma a první rozsvěcí a poslední zhasne.

Vylepšenou variantou je problém **férových čtenářů a písařů**. Pokud písař chce psát a pořád nějaký čtenář čte, tak mohou písaři vyhladovět ⇒ chceme aby měli písaři přednost. To přidá podmínku, že pokud chce nějaký písař psát, nesmí žádný nový čtenář začít číst (ostatní čtenáři své čtení dokončí normálně). Viz kód dále.

Python
```
class Lightswitch:
  def __init__(self):
    self.counter = 0
  self.mutex = Semaphore(1)

  def lock(self, semaphore):
    self.mutex.wait()
      self.counter += 1
      if self.counter == 1:
        semaphore.wait()
    self.mutex.signal()

  def unlock(self, semaphore):
    self.mutex.wait()
      self.counter -= 1
      if self.counter == 0:
        semaphore.signal()
    self.mutex.signal()
```

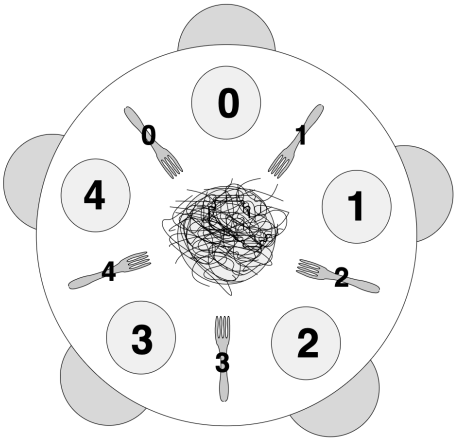
Python
```
# inicialiazce
readSwitch = Lightswitch()
writeSwitch = Lightswitch()
noReaders = Semaphore(1)
noWriters = Semaphore(1)
```

Python
```
# písaři
noReaders.wait()
  readSwitch.lock(noWriters)
noReaders.signal()
  # KRITICKÁ SEKCE
readSwitch.unlock(noWriters)
```

Python
```
#čtenáři
writeSwitch.lock(noReaders)
  noWriters.wait()
    # KRITICKÁ SEKCE
  noWriters.signal()
writeSwitch.unlock(noReaders)
```

2.5.3 Večeřící filozofové

Procesy v tomto problému požadují více sdílených zdrojů ⇒ **konkurují si**. Je zde 1 typ procesů, který má 2 operace – jí (vyžaduje zdroje) a přemýšlí (nevyžaduje zdroj). Sedí u kulatého stolu a mezi každou dvojicí leží 1 vidlička a každý potřebuje k jídlu ony 2 vidličky vedle něj. Vidličku smí vždy držet jen 1 filozof, vždy může někdo jíst (nesmí dojít k deadlocku), nikdo nesmí vyhladovět (vyhladovění) a v jediném okamžiku musí být umožněno jíst více filozofům (lepší efektivita).



Obrázek 6: Večeřící filozofové vizualizace

```python
# základní chování filozofů
while True:
  think()
  get_forks()
  eat()
  put_forks()
```

Existuje několik korektních verzí řešení:

1. Použití číšníka (*footman*), který eliminuje počet filozofů, kteří mohou jíst (pokud budou jen 4 nemůže dojít k deadlocku)
2. Další řešení je mít alespoň 1 praváka a 1 leváka, poté nemůže dojít k deadlocku
3. Řešení podle Tanenbauma přiřazuje filozofům stavy (*eating*, *thinking*, *hungry*) a používá semafory, které indikují zda filozof může začít jíst. Toto řešení není ideální neboť může dojít k vyhladovění.

2.5.4 Morissův algoritmus

Něco jako *non-starve mutex solution* (zámek o který když vlákno požádá musí ho v konečném čase dostat) z knížky Little Book of Semaphores od Allen B. Downey z kapitoly 4.

```python
mutex.wait()
  room1 += 1
mutex.signal()

t1.wait()
  room2 += 1
  mutex.wait()
  room1 -= 1

```

```
    if room1 == 0:
      mutex.signal()
      t2.signal()
    else:
      mutex.signal()
    t1.signal()

  t2.wait()
    room2 -= 1
    # critical section
    if room2 == 0:
      t1.signal()
    else:
      t2.signal()
```

2.5.5 Problém kuřáků

V tomto problému máme 4 vlákna – 1 agent a 3 kuřáci.

Popis problému: kuřáci dokola opakují čekáni na suroviny (tabák, papírky, sirky), smotání cigarety a kouření. Předpokládáme, že agent má neomezeně všech surovin a každý kuřák má neomezeně od jedné suroviny. Agent opakovaně náhodně vybírá 2 ingredience a dává je k dispozici kuřákům (podle toho které vybere zvolí kuřáka, kterému ingredience stačí k ubalení cigarety).

Analogie k praktickému problému je taková, že **agent představuje operační systém, který alokuje zdroje a kuřáci jsou programy, které zdroje chtějí a potřebují**.

Obecně se prezentuje více různých verzí, které mají různě náročné řešení (případně jsou neřešitelné). Tady uvažujeme verzi, kde platí, že **nemůžeme modifikovat kód (nastavení) agenta**, ostatní omezení nejsou.

Python
```
# agent
agentSem = Semaphore(1)
tobaco = Semaphore(0)
paper = Semaphore(0)
match = Semaphore(0)
```

Agent se v podstatě skládá ze 3 konkurentních vláken – agent A až agent C.

Python
```
# agent A
agentSem.wait()
tobaco.signal()
paper.signal()
```

Python
```
# agent B
agentSem.wait()
paper.signal()
match.signal()
```

Python
```
# agent C
agentSem.wait()
tobaco.signal()
match.signal()
```

Obvyklá řešení, která nás napadají zde nefungují. Nejčastěji se objeví problém deadlocku. Řešením od Parnase je použití 3 vláken navíc zvaných *pushers*. Ti odpovídají na signály od agentů a drží si přehled dostupných surovin, tak aby mohli zavolat vhodného kuřáka.

Python
```
# inicializace
isTobacco, isPaper, isMatch = False
tobaccoSem = Semaphore(0)
```

```
paperSem = Semaphore(0)
matchSem = Semaphore(0)
```

```python
# pusher A
tobacco.wait()
mutex.wait()
  if isPaper:
    isPaper = False
    matchSem.signal()
  elif isMatch:
    isMatch = False
    paperSem.signal()
  else:
    isTobacco = True
mutex.signal()
```

Probudí se kdykoliv se objeví tabák. Pokud bude `isPaper` pravdivý, tak *pusher B* už musel být zavolat a může tedy signalizovat kuřáka, který má sirky. Pokud se to samé stane s `isMatch`, zavolá se kuřák s papírky. Nakonec pokud *pusher A* běžel první, vše ostatní bude `false` a on tedy signalizuje, že tabák je dostupný, ale nikoho nemůže zavolat.

Pro ostatní 2 pushery je kód obdobný.

```python
# kuřák, který má tabák
tobaccoSem.wait()
makeCigarette()
agentSem.signal()
smoke()
```

Mohou nastat další zobecněné varianty tohoto problému. Třeba pokud upravíme agenta tím, že eliminujeme požadavek, že agent musí čekat po vybrání suroviny. V tomto případě může být na stole několik instancí jedné ingredience.

2.6 Návrh paralelního systému

Lze rozdělit do několika kroků. Obvykle před vytvářením paralelního systému máme ten systém sekvenční, pokud ne, tak prvně uděláme sekvenční řešení (je intuitivnější a jednodušší). Dalším krokem je dekompozice (rozklad) na menší části. Dá se dělat podle úkolů (*task decomposition*) – jaké úkoly máme, jejich závislost (pomocí grafu) a nebo podle dat (*data decomposition*) – jak rozdělit data pro konkurentní zpracování, potřebujeme podporu u HW, typické pro SIMD a má dobrou horizontální škálovatelnost.

Pipeline. Vzor pro dekompozici podle úkolů. Úkol jde rozdělit na více po sobě jdoucích úkolů. Více úkolů poběží najednou na jiný částech dat. Např. montážní linka na auta, proces *extract--transform-load*.

Paralelizace cyklů (loop-level). Vzor pro dekompozici podle dat. Cyklus provádějící daný úkol (nezávisle) pro iterovaná data. Každou iterace můžeme provést paralelně se všemi dalšími. Jde provést jednoduše třeba s `threadPool` a někdy provádí sám kompilátor.

Mapování. Vzor pro dekompozici podle dat. V kolekci dat na každém prvku provádíme stejnou operaci (něco takového jsme brali v Lispu a Paradigmatech programování). Opět každý můžeme provést paralelně. Podobné jak s cykly o odstavec více, ale přístup je funkcionální.

Fork/Join. Vzor pro dekompozici podle dat. Část zpracování dat se dá udělat paralelně na částech dat a následně potřebujeme zpracovat výsledky z těchto části (proto název *fork* a *join*). Např. paralelní merge sort.

Map/Reduce. Vzor pro dekompozici podlet dat. Nejdříve mapování a pak agregace (redukce). Podobné jako *Fork/Join*, ale z funkcionálního pohledu.

0. Sekvenční řešení
1. Dekompozice (podle úkolů nebo dat)
2.

2.6.1 Fosterova metodologie

Vymyslel postup návrhu paralelního programu.

1. Dekompozice – rozdělení na menší části
2. Komunikace – popis a zabezpečení komunikace mezi částmi
3. Aglomerace – shlukování částí do logických skupin
4. Mapování na zdroje – technická realizace

Příkladem může být násobení matic nebo určení četnosti slov v textu. Pro každý z těchto příkladů je v prezentaci 07 rozpracovaný krokový návrh.

3 Distribuované systémy

Tady je zbylých 7-8 přednášek.

Distribuovaný systém můžeme chápat jako speciální případ paralelního systému, jelikož více autonomních systémů spolupracuje na jednom úkolu. Mohou být na jediném stroji, případně v síti WAN i LAN, eventuálně i více distribuovaných systémů tvářících se jako jeden (můžou se řetězit). Oproti paralelním systémům mají několik rozdílů:

- absence sdílené paměti,
- komunikace (jak, rychlost, latence, spolehlivost),
- synchronizace času.

Příklady mohou být: aplikace komunikující s DB, IoT, BlockChain, Internet, …

3.1 Architektura distribuovaných systémů

Rozdíl mezi SW architekturou a systémovou architekturou. Plno možností jak bude co udělaná a co se musí vyřešit.

Klient-Server. Klienti požadují od serverů služby (ti je nabízí). Asi nejvíc běžné. Pro někoho může být uzel A klient pro někoho server (záleží na úhlu pohledu). Obvykle po síti. Nevýhodou je centralizace (kvůli serveru). Pro komunikaci klienta se serverem se používá API. Komunikace formou *request-response* a probíhá asynchronně. Např. webové služby, databázové servery, e-mailové servery.

Peer-to-Peer. Založena na myšlence, že všechny uzly si jsou rovny a znají své sousedy. Tím pádem je systém plně decentralizovaný. Stává se tedy snadno škálovatelný a dynamický (uzly mizí a přibývají dle potřeby). Např. blockchain, BitTorrent.

Vrstvená-Architektura. Systém rozdělen do logických vrstev (obvykle jsou 3 – uživatelské rozhraní, logika aplikace, datová vrstva). Obvykle jsou nezávislé a komunikují mezi sebou. Vrstvy komunikují jen se sousedními. Např. větší webové aplikace (UI jako JavaScript, back-end třeba C# a DB v MySQL).

Service oriented architecture (SOA). Snaha mít komponenty jako služby a znovu je používat. Výhodou je vysoká modularita a dobrá škálovatelnost.

Microservices. Něco jako neprovázaná SOA. Jednotlivé microservices by mělo být možné nasazovat nezávisle. Členění bývá uděláno podle logiky aplikace. Dnes velmi populární.

Event-driven architecture. Centrálním prvkem systému jsou události. Jednotlivé komponenty je produkují a následně konzumují. Jsou na sobě plně nezávislé. Např. chytrá domácnost, IoT, Kafka.

Hybridní přístup. Kombinuje více různých přístupů. Např. BitTorrent, který má prominentní uzly sloužící jako „servery“.

3.2 Komunikace v distribuovaném systému

V distribuovaných systémech **nemáme sdílenou paměť** ⇒ musíme **komunikovat přes posílání zpráv**. U zpráv máme různé úrovně abstrakce, různé protokoly. Jsou to klasické sítě jako jsme probrali dříve (KMI/POS1 a KMI/POS2). V realitě je to **vrstvený model** sítě jako ISO/OSI.

Budeme řešit věci, které jsou především v aplikační vrstvě (především z ISO/OSI tam je to jednodušší). V této vrstvě funguje např. DNS, autentizační/autorizační protokoly, komunikační protokoly RPC a AMQP, ... Fyzický přenos za náš řeší OS.

3.2.1 Midleware protokoly

Systém pro zpracování zpráv. Jsou 2 možnosti – perzistentní systém (drží zpráv dokud není doručena) a tranzistentní (dočasný) systém. Odesílatel může být asynchronní (jen pošle a pracuje dál) nebo různě moc synchronní (potvrzení převzetí, potvrzení přijetí příjemcem, …). Přístupy odesílatele a middleware můžeme kombinovat.

3.2.2 Remote Procedure Call (RPC)

Z pohledu procesů není vidět, že bychom nějak komunikovali přes síť (funguje prostě tak, že se volají přímo lokální metody). O co jednodušší je myšlenka, tak provedení je náročné (různé adresní prostory, předání argumentů/výsledků, jiné konvence, stroj může vypadnout ze sítě, …). RPC musí být podporováno programovacím jazykem (u všech běžných to je).

Provedení z pohledu klienta. Klient netuším, že volá metodu ze vzdáleného objektu, pro něj jde o lokální volání `remoteObject.methods(arg)`. Middleware

Provedení z pohledu volaného

Může nastat řada problémů. Předávání parametrů – marshalling(unmarshalling). Výsledek jako jednoduché typy je úplně bez prolbémů, horší jsou referenční typy (nejhůře ty složité).

RPC funguje v několika variantách (asynchronní, jednosměrné, multicast). Obecně ale RPC tlačí, aby zpracování bylo synchronní (čili počkám si na zpracování a návrat výsledku).

3.2.3 Další možnosti komunikace

- **MOC** – obecné posílání zpráv (message oriented communication). Příkladem můžou být roury (pipes). Jednosměrné spojení výstup-vstup. V Pythonu jsou obousměrné. Sockety jsou abstrakcí portu. Různé operace pro klienta (`connnect`, `send`, `close`, …) a pro server (`bind`, `listen`, `accept`, …). Sockety jsou ještě nad HTTP protokolem. Message queue jsou fronty zpráv. Jsou perzistentní. Např. RabbitMQ. Message broker je centrálním místem pro výměnu zpráv. Má poměrně rozsáhlé možnosti. Např. ApacheKafka.
- **MOM** – message oriented middleware
- **MPI** – message passing interface
- **IPC** – inter-process communication

Několik různých modelů komunikace: *request-reply*, *publish-subscribe*, *pipeline*.

Vysvětlení multicastu a broadcastu. Overlay jako struktura nad sítí (tree nebo mesh). Multicast má více typů – flooding (uzel přepošle všem kromě zdroje, pozor na duplikaci) a edpidemické protokoly/gossip (snaha informovat okolí náhodným výběrem).

- `gRPC` framework
- `protobuf` repository
- RPC in Python

3.3 Koordinace a čas v distribuovaných systémech

Koordinace DS probíhá na základě času (typicky timestamp). Na různých uzlech by mohl být různý čas. V DS nemůže být jednotný čas kvůli různé době cestování mezi různými uzly (jsou

totiž různé typy spojení). Obecně chybí globální hodiny. Je rozdíl mezi skutečným časem a logickým časem.

Omezení času **Poznámka 6**

Čas se nikdy nevrací zpět! Rozbily by se tím timestamp na onom uzlu.

3.3.1 Skutečný čas

Nelze plně synchronizovat (tudíž se snažíme synchronizací jen snížit rozdíl). Použití naivního algoritmu (viz níže) nefunguje, protože: latence sítě, mohlo by dojít k posunu dozadu, není společný referenční časový bod.

1. Klient K požaduje UTC čas od serveru S
2. K požádá o čas S.
3. S odpoví svým aktuálním časem t_s.
4. K nastaví čas dle t_s.

3.3.2 Cristianův algoritmus

Máme klienta K a server S s časem UTC. Server S nemusí být nutně na nejvyšším autoritativním levelu, jen je na vyšší úrovni než klient (jinak to nedává smysl).

1. K požáda o čas S a uloží si čas požadavku t_1
2. S odpoví aktálním časem t_s
3. K si uloží čas přijetí t_2
4. výsledný čas K: $t_s + \frac{t_2 - t_1}{2}$

Čas z tohoto může být celkem nepřesný. Můžeme tento algoritmus zopakovat x-krát a z toho udělat průměr nebo medián pro zisk lepšího výsledku.

3.3.3 Network Time Protocol (NTP)

V praxi dost často používané. Základně podobné Cristianově algoritmu, ale lépe odhadujeme zpoždění zpráv a na základě toho dostaneme přesnější čas. Centrální autoritou je UTC.

1. K požádá o čas S v čase t_1
2. S přijme požadavek v čase t_2
3. S pošle odpověď s (aktuálním) časem t_3
4. K ji přijme v čase t_4
5. proběhne odhad offsetu[4] K vůči S: $\Theta = t_3 + \frac{(t_2 - t_1) + (t_4 - t_3)}{2} - t_4$
6. odhad zpoždění zpráv: $\delta = \frac{(t_4 - t_1) - (t_3 - t_2)}{2}$
7. K eventuálně upraví rychlost času pro srovnání

Výpočet Θ a δ se provádí vícekrát (pro lepší přesnost). Pokud by K byl napřed oproti S (záporné Θ), tak se čas na K zpomalí. Pro nastavení času na K se použije minimální δ.

3.3.4 Reference Broadcast Synchronization (RBS)

Distribuované řešení používané v bezdrátových/senzorových sítích (IoT). Cílem zde není mít skutečný přesný čas, ale shodnou se na stejném v rámci sítě. Předpokládá se stejná rychlost přenosu mezi všemi zařízeními. Uzly si pouze ukládají offset, neupravují svůj čas.

1. Uzel u pošle broadcast zprávu m (referenční impulz), která to celé začne

[4] O kolik jsou lokální hodiny K rozdílné oproti času serveru S. Může nabývat i záporných hodnot.

2. Uzly p a q uloží po obdržení časy přijetí t_p a t_q
3. Uzly p a q si vzájemně pošlou zprávu s časem přijetí m (tedy t_p a t_q)
4. Uzly p a q z dodaného času vypočítají offset proti druhému uzlu
5. Časem výsledky mohou být neaktuální kvůli driftu (průměry, lineární regrese)

3.3.5 Logicky čas

Skutečný čas přináší celkem dost problémů a může nám stačit „méně“. Logický čas je něco jako správné uspořádání operací (operace `a` nastala po operaci `b`). Takovou operací může být to, že *odeslání zprávy musí proběhnout před jejím zpracováním* (tzv. **koncept předcházení**).

3.3.6 Koncept předcházení ($a \to b$)

Byl představen Lamportem v roce 1978. Jedná se o tranzitivní relaci (nikoliv úplnou), takže pokud platí $a \to b$ (čteme jako a *předchází* b) a $b \to c$, pak musí platit $a \to c$. Pokud jsou 2 události v odlišných procesech, které nesdílí zprávy pak $x \to y$ není pravda, ale ani $y \to x$ není pravda.

Tyto pravidla jsou dobře vidět na vizualizacích.

- Článek - Lamport, logický čas
- Článek - Ricart, Agrawala, vzájemné vyloučení
- Apache zookeper
- Blogy o čase v DS
- Blogy o čase v DS vol. 2

3.3.7 Lamportův algoritmus

Lokální hodiny jsou vnímány jako *čítač* (za každé provedení operace `+1`). Čítač se zvedá při:

- vykonání lokální operace,
- před odesláním zprávy,
- při přijetí zprávy s vyšší hodnotou čítače (tam se bere vyšší hodnota z hodnoty přijaté a hodnoty mojí zvýšené o 1).

3.3.8 Vektorové hodiny

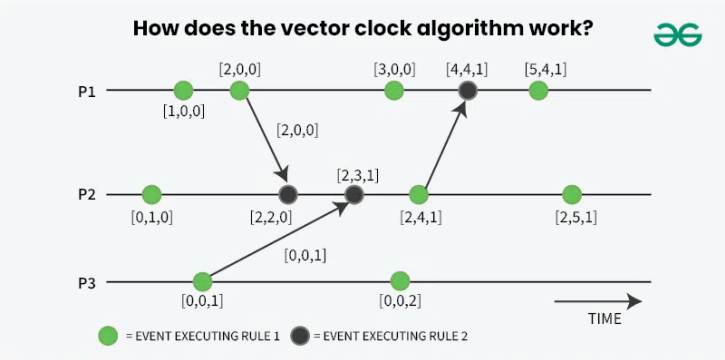
Každé události a můžeme přiřadit časovou hodnotu $C(a)$. Potom platí $a \to b$, tak $C(a) < C(b)$. Naopak NEplatí, že pokud $C(a) < C(b)$, tak $a \to b$ (nepopisují kauzalitu).

Vektor hodin $\vec{C}_i$ je v každém procesu. $\vec{C}_i[i]$ je lokální logický čas procesu i. $\vec{C}_i[j] = k$ znamená, že proces i ví, že proces j vykonal k operací. Pokud hodiny není možné porovnat jedná se o konkurentní operace. Proces i se dozví o změně hodin procesu j pouze pokud od něj dostane zprávu (kde jsou jeho hodiny).

Platí následující uspořádání:

$$\overrightarrow{C_1} \leq \overrightarrow{C_2} \text{ pokud } \forall i : \overrightarrow{C_1}[i] \leq \overrightarrow{C_2}[i]$$

$$\overrightarrow{C_1} < \overrightarrow{C_2} \text{ pokud } \overrightarrow{C_1} \leq \overrightarrow{C_2} \text{ a } \overrightarrow{C_1} \neq \overrightarrow{C_2}$$



Obrázek 7: Znázornění vektorových hodin.

3.4 Vzájemné vyloučení v DS

Vzhledem k tomu, že v DS musí procesy spolupracovat na dokončení úkolu nastává často situace, kdy musí přistupovat ke stejnému zdroji. Pro vyloučení nekonzistence je nutné zajistit exkluzivní přístup k tomuto zdroji. Obecně se používají dva typy přístupu *přístup založený na tokenu* a *přístup založený na oprávnění* a řešení mohou být centralizovaná, decentralizovaná nebo distribuovaná.

3.4.1 Centralizované řešení

Nejjednodušším řešení, které člověka napadne je určení jednoho uzlu (rozhodčí), který bude rozhodovat o tom, kdo má výhradní přístup ke sdílenému zdroji. Funguje na principu fronty a posílání zpráv `REQUEST` a `RELEASE`.

Nevýhody: selhání rozhodčího (ostatní nemají jak poznat), přetížení rozhodčího (bottleneck).

3.4.2 Distribuované řešení

Řešení je inspirováno Lamportovými logickými hodinami. Řešení vyžaduje úplné seřazení událostí v systému (u jakékoliv dvojice událostí musí být jasné, která z nich se stala první). Potřebuje $2(N-1)$ zpráv (kde N je počet uzlů) a využívá zprávy `REQUEST` a `ACK`.

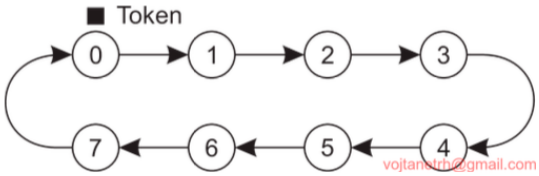
1. Proces i chce výhradní přístup
2. Zašle `REQUEST`(i, t) ostatním (t je časové razítko zprávy)
3. Proces j po přijetí odpoví `ACK` (nechce přístup nebo $t < C_j$), nebo přidá i do fronty žádostí.
4. Pokud čekající i dostane $(N-1)$ `ACK`, získává přístup.
5. Při ukončení práce pošle i ACK všem ve své frontě.

Nevýhody: možných N bodů selhání (opakující se zprávy).

3.4.3 Řešení tokenem

Nad sítí se vytvoří virtuální kruh v němž si jednotlivé uzly předávají token (ten je právě jeden). Platí výhradní práce se zdrojem = držení tokenu. Když proces obdrží token zjistí zda-li chce pracovat se dílením zdrojem, pokud ano pracuje (po dokončení odešle token dále), pokud ne odesílá token dále ihned.

Tento přístup má vlastní problémy – ztráta tokenu (při výpadku uzlu nebo nedoručení zprávy), nutnost udržovat informaci o kruhu kvůli výpadku následovníka.



Obrázek 8: Vzájemné vyloučení pomocí tokenu.

3.4.4 Decentralizované řešení

Přsístup je založen na hlasování. Každý sdílený zdroj má N replik, které vždy mají svého koordinátora, který slouží k výhradnímu přístupu. Pokud chce uzel přistupovat ke sdílenému zdroji potřebuje většinu hlasů $m > \frac{N}{2}$ od koordinátorů. Pokud byl přístup zamítnut bude uzel žádat znovu po náhodném čase. Problém nastává při výpadku koordinátora a jeho resetování, může totiž nekorektně pracovat s tím jaký hlas a komu už udělil.

3.4.5 Zookeper

Řešení používané v reálném světe pro tyto typy problémů. Jedná se o celý (centralizovaný) koordinační systém. Poskytuje vzájemné vyloučení, volbu lídra, monitoring, zotavení z chyb.

3.5 Lídr a jeho volba

Lídr je nějak významný uzel. Má několik typických úkolů: komunikace s uživatelem, koordinace, vzájemné vyloučení, atd. Na tom kdo je leader se musí shodnout **všechny** uzly. Lídří se v průběhu času mohou měnit. Často je toto chování od systému vyžadováno navenek.

Předpokladů pro leadera je několik:

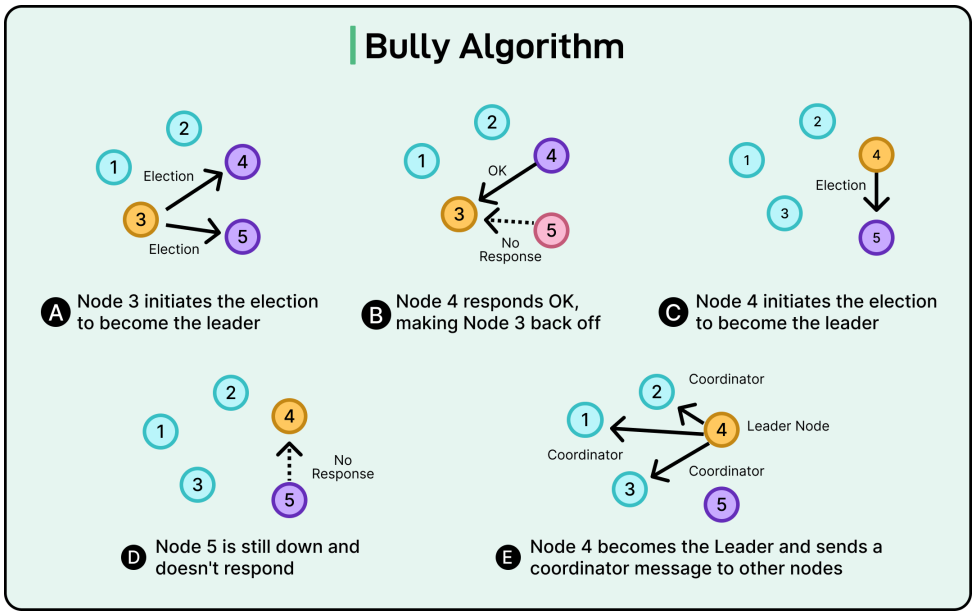
- všechny uzly mají `ID` (předeším pro volbu, preferujeme větší)
- všichni o všech ví
- uzly mohou vypadnout (u běžného uzlu nevadí, u leadera musí být nějak vyřešeno)

3.5.1 Bully algoritmus

Pracuje na principu „silnější vyhrává“. Kandidující uzly jsou:

- ty které zjistí, že leader nereaguje
- nově příchozí uzel
- silnější může převzít kandidaturu.

Kandidátů může být logicky v nějaký čas více. V nejhorším případě $O(n^2)$ zpráv. Pokud dojde k rozdělení sítě algoritmus selže.



Obrázek 9: Bully algoritmus pro volbu leadera.

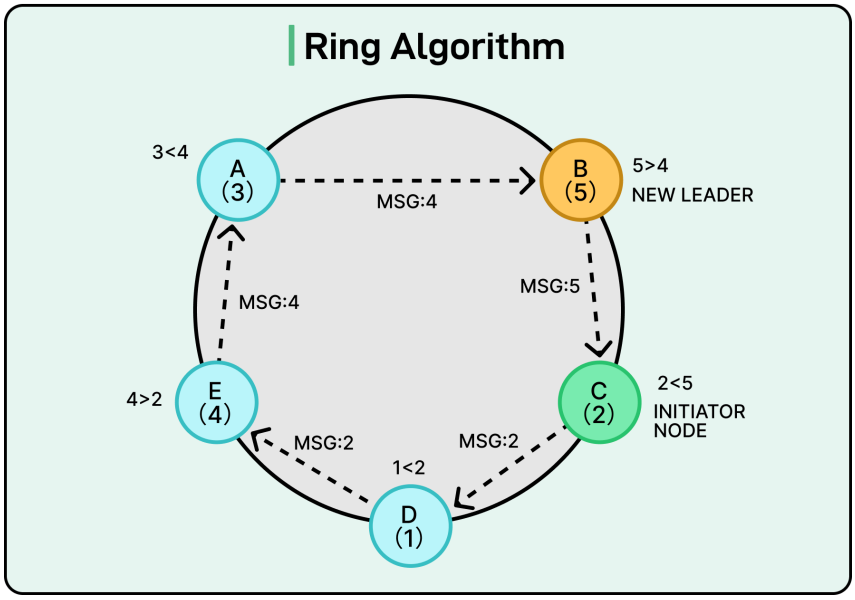
1. Uzel začne posláním zprávy **`election(id)`** všem uzlům, které mají vyšší **`id`** než on sám
2. Uzel, který obdržel zprávu buď neraguje nebo přebírá kandidaturu (má vyšší **`id`**) a odpoví **`OK`**
3. Postupně se opakuje a jediný, kdo zůstane se stává leaderem (nejvyšší **`id`**), což musí všem oznámit (zpráva **`COORDINATOR`**).

3.5.2 Ring algoritmus

Je to tzv. logický kruh (overlay). Všichni musí znát následovníky i další uzly. Volby zahájí uzel (uzly), které zjistí že leader nereaguje, takže postupně kandidují všichni běžící. Vždy $O(n)$ zpráv.

Začíná se u uzlů, které zjistí že leader nereaguje a ti vyhlašují volby, které probíhají následovně:

1. Uzel posílá zprávu **`election(id)`** svému následovníkovi (pokud nefunguje, tak ho přeskočí)
2. Uzel který obdrží zprávu, tak má 2 možnosti – pokud zprává neobsahuje jeho **`id`**, tak ho přidá a pošle dál (tím se dostává do voleb) nebo pokud obsahuje, tak pošle zprávu **`leader`** se stejným seznamem (už ten seznam u něj musel jednou být)
3. Až to takto oběhne kolečko (vlastně 2x), tak se leaderem stává ten s nejvyšším **`id`** (ten se to sám dozví z toho seznamu).



Obrázek 10: Bully algoritmus pro volbu leadera.

3.5.3 Raft algoritmus

Základem pro další algoritmy. Jde vlastně o takový distribuovaný log, do kterého vidí všichni. Funguje na většinovém kvóru. Umožňuje tolerovat chyby. Na počátku algoritmu jsou všichni následovníci. Pokud dojde k rozdělení sítě, tak v každé části nastanou volby a Raft si s tím dokáže korektně poradit.

Máme 3 stavy uzlů – *lídr, následovník a kandidát.*

Několik stěžejních pojmů – *term, timeout, hearbeat, split-vote.*

Co platí? V každém termu je pouze jeden lídr. Termy jsou číslované od 0. Leader musí v pravidelných intervalech posílat heartbeat, aby se vědělo, že žije. Pokud by se uzly do timeoutu neshodly nastanou nové volby.

Volba lídra

1. Volby vyvolá následovník pokud mu nepřijde heartbeat od lídra – zvýší svůj term, stane se kandidátem, hlasuje pro sebe a požádá všechny ostatní o hlas (poslání zprávy)
2. Pokud uzel obdrží žádost o hlas tak – pokud nehlasoval a nemá novější log (více commitovaných operací) než uzel který poslal žádost, tak hlas potvrdí nebo pokud hlasoval ignoruje
3. Když kandidát dostane většinu hlasů (z celkového počtu uzlů), tak se prohlásí leaderem

Všechno o tomto algortimu od autora (články, knihovny v různých jazycích i vizualizace) je tady na webu.

3.5.4 Algoritmus pro ad-hoc[5] sítě

V tomto případě jsou kandidáti na lídra všichni a je vybrán ten, kdo nejlépe plní požadavky.

1. Ten kdo chce být lídrem posíla `ELECTION` svým sousedům
2. Ten kdo obdrží `ELECTION` pokud nemá rodiče, tak si na jeho místo nastaví odesílatele, přepošle všem ostatním a po potvrzení od nich potvrdí i on
3. Poku už rodiče má, tak jen potvrdí přijetí

[5]Obvykle bezdrátová síť bez centrálního bodu a s proměnlivou topologií

4. Tímto vzniká strom

3.5.5 Další algoritmy

- Zookeper
- Chang-Roberts (upravení Ring algoritmus)
- Gossip protokoly (napodobuje šíření drbů nebo virů v populaci)
- Proof of work nebo Proof of stake (pro velké sítě)

3.6 Chyby v distribuovaných systémech

Chyba je selhání čehokoliv v systému (uzlu, hrany, kanálu, …). Prostě je to jiné než zamýšlené a očekávané chování. To dělá reakci výrazně složitější. Obecně musíme v omezeném množství tolerovat chyby.

Asynchornní systém neumožní chyby detekovat, protože nevíme jestli probíhá výpočet a čeká se nebo uzel vypadl. U částečně synchronního s timeouty to možné je.

Základní 4 vlastnosti:
1. dostupnost (dává odpověď)
2. bezpečnost (chyba nezpůsobí katastrofu)
3. spolehlivost (běží nepřetržitě v daném časovém úseku)
4. udržovatelnost (snadnost opravy v případě chyby).

Občas problémy s rozlišením dostupnosti a spolehlivosti.

Metriky pro měření chyb (z jejich hodnot můžeme vyčíst, které vlastnosti jsou dobře splněny):
- MTTF – meain time to failure (čas po kterém dojde k chybě)
- MTTR – mean tim to repair (čas nutný k opravě)
- MTBF – mean time between failures (čas mezi výpadky)

Chyba **Poznámka 7**

Všechny předchozí definice a odstavce važdují aby byla chyba přesně definována.

3.6.1 Klasifikace chyb

Rozlišujeme podle trvání nebo podle projevu. Pomáhá rozlišit jak jsou chyba závažné, případně jaký způsob je vhodný pro jejich řešení.

1. **přechodná** – objeví se jednou a zmizí
2. **přerušeovaná** – objevuje se a mizí opakovaně
3. **trvalá** – trvá do vyřešení

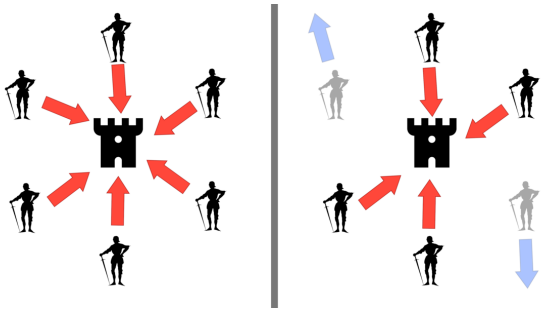
1. **pád** – uzel vypadne, jinak funguje bezproblémů
2. **vynechání** – selhání v poslání/přijetí zprávy
3. **časování** – odpověď mimo daný časový rámec (pozdě)
4. **chyba odpovědi** – odpovídá špatně
5. **náhodná (byzantská) chyba** – odpovídá náhodně a v náhodném čase

3.6.2 Byzantské chyby

Chyba uzlu, kterou ostatní nepoznají (jedna z nejhorších variant). Uzel se chová ke každému jinak. Dobré znázornění na *problému byzantských generálů* (koordinované útoky/ústupy a (ne)poctivost generálů). Jedná se o reálný problém. může jít o bug i útok na síť.

Systém odolá byzantské chybě pokud:

- pošle-li poctivý uzel hodnotu X, systém se shodne na X
- všechny poctivé uzly se shodnou na X



Obrázek 11: Problém byzantských generálů.

3.6.3 Detekování chyb a redundance

Typy chyb seřazené od nejméně závažné po nejvíce závažné.

Fail-stop – spolehlivě detekovatelné, známe stav uzlu

Fail-noisy – eventuálně spolehlivě detekovatelné, chyba je zjevná

Fail-silent – nelze rozlišit pád a vynechání (neznáme stav uzlu)

Fail-safe – o chybě nevíme nic, ale nezpůsobí škodu (uzel se přepne do tzv. *fail-safe módu*)

Fail-arbitrary – o chybě nevím nic a nelze ji ani spolehlivě detekovat

Redundance umožňuje tyto tolerovat chyby. Může ji být několik typů, které jsou odolné vůči rozdílným typům chyb (informační, časová, fyzická redundance).

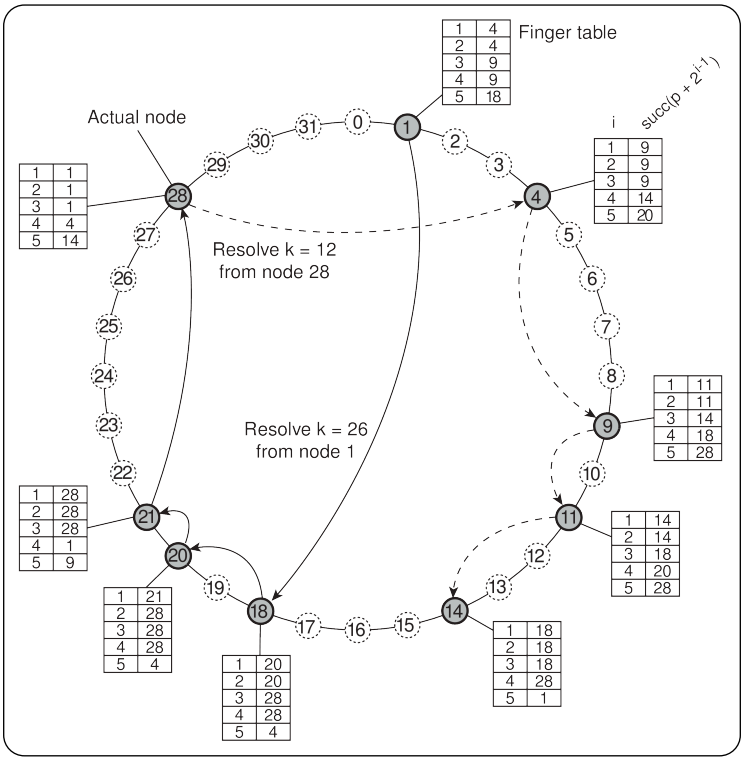
Triple modular redundancy **Definice 8**

Jde o přístup kdy každý uzel má 3 repliky. Pokud se 2 shodnou považuje se výsledek za korektní (čili až 1 může selhat). Může vézt až ke l replikám, kdy systém přežije až k chybných uzlů (potom říkáme k-fault tolerance).

3.7 Chord Systém

Něco jako distribuovaná hashovací tabulka (takže režim klíč-hodnota). Jedná se o peer-to-peer systém, který má kruhovou strukturu (ring). Používají se m bitové klíče (obvykle 128 nebo 160), kdy `id` uzlů jsou také z tohoto prostoru. Dohromady 2^m klíčů.

Klíč k spadá pod uzel s nejmenším `id` splňující `id` $\geq k$. Takovému uzlu říkáme následovník a značíme ho `succ(k)`.



Obrázek 12: Chord systém.

Poznámka 9

Uzly evidující informace i o předchůdcích.

Je nutné vyřešit jak efektivně hledat správný uzel pro klíč k. Naivní přístup říká, že uzel p ví o svém následovníkovi `succ(p + 1)`. V tomto případě pokud p obdrží požadavek na klíč k, tak požadavek přepošle dokud neplatí `pred(p)` $< k \leq p$ a poté správný uzel odešle své `id`, protože jemu náleží klíč k. Tento přístup v průměru potřebuje projít $\frac{1}{2}$ ringu (což je neefektivní – složitost $O(n)$).

Efektivnější řešení je použít zkratky, tzv. *finger tables* FT_{id} (pro uzel s `id`). Jde o vyhledávací tabulku s velikostí max m. Platí $FT_{\text{id}}[i] = \text{succ}(\text{id} + 2^{i-1})$ neboli i-tý záznam v tabulce ukazuje na následníka vzdáleného alespoň o 2^{i-1} pozic. Pro udržení v kruhové struktuře se používá modulární aritmetika. Pro korektní přeposlání klíče k z uzlu p na uzel q využijeme:

$$q = FT_p[j] \leq k < FT_p[j+1]$$

Tento vylepšený přístup vyžaduje $O(\log(N))$ kroků, což je značné zlepšení.

Udržení této tabulky aktuální se provádí pomocí neustále opakované operace. V ní uzel q opakovaně kontaktuje uzel `succ(q + 1)` a vyžádá si `pred(succ(q + 1))`, pokud se tato hodnota rovná q, pak je vše správně. Pokud ne, tak někdo nový vstoupil do ringu $q < p \leq \text{succ}(q+1)$ a je nutné aktualizovat záznamy tak, že $FT_q[1] = p$. Poté si ještě p zkontroluje zda má q jako předchůdce (případně nastaví ho). Taktéž se pravidelně kontroluje zda předchůdce žije, pokud ne je nastaven na `unknown`. Tyto procedury zajišťuje že Chord je pořád s velkou většinou uzlů v konzistentním stavu.

Pokud se chce uzel p přidat do systému je postup jednoduchý. Uzel p kontaktuje příslušný uzel a zažádá si o `succ(p + 1)`. Po odpovědi se p může připojit do kruhu.

Reálné fungování
V reálném světě by mohlo docházet k tomu, že geograficky blízké uzly mají dost odlišná `id`. Systém se proto snaží uzlům blízko sobě geograficky přiřadit blízké hodnoty `id`. Pro následníky i záznamy v tabulce je zavedena redundance – čili nemají jen jednoho (jeden záznam), ale je jich r.

3.8 Shoda v distribuovaném systému

Systém se musí shodnout na řadě vecí (výsledek, další akce, stav, …). Pokud funguje bez chyb je to triviální, s chybami (hodně záléží na typu a závažnosti) se situace komplikuje. Pro shodu potřebujeme: synchronní systém nebo omezený čas na zpráv a pořadí zpráv nebo multicast.

> **Poznámka 10**
>
> Shoda v DS není vždy možná.

3.8.1 Redundance

Pro řešení těchto problémů je zavedena **redundance**, kdy úlohu jednoho uzlu přebírá skupina uzlů. Existují dva základní typy:

1. **primární záloha** – hierarchická skupina, při výpadku primárního (te nvykonává všechny `write` operace) převezme roli záloha
2. **replikovaný zápis** – ve skupině mají všichni stejné role (plochá skupina), nemají kritický bod, je nutná koordinace

> **Odolnost proti chybám** **Definice 11**
>
> Skupina je odolná proti chybám pokud všechny bezchybné procesy vykonávají stejné operace ve stejném pořadí.

3.8.2 Algoritmus Flooding consensus

Základní algoritmus pro shodu pracující pouze s chybami *fail-stop*. Shoda se hledá v kolech, kterých je $f + 1$, kde f je maximální počet uzlů, které mohou selhat (logicky $f < N$, kde N je počet uzlů v DS).

1. V každém odešle každý uzel svůj aktuální seznam návrhu (začíná se s tím, kde je pouze jeho vlastní návrh)
2. Uzel přijaté návrhy v kole sloučí se svými návrhy
3. Každý uzel determinsticky ze seznamu nárvhu vybere volbu podle předem dané funkce
4. Rozhondutí na základě odpovědí:
    1. Pokud nikdo nedostane všechny, odpovědi začíná nové kolo
    2. Pokud všichni dostanou všechny odpovědi, dojde k volbě výsledku a algoritmus skončí
    3. Pokud jen někdo dostane všechny odpovědi, rozešle volbu, kterou čekající převezmou a končí se

3.8.3 Byzantské chyby podruhé

Máme n generálů z nichž m je zrádných. Jak toto vyřešit? Při nákresu řešení s postupně zvyšujícím se počtem generálů (2, 3, 4, 5, …) se dostaneme k tomu, že musí platit $n > 3 \cdot m$, aby se systém shodl. Nevýhodou je exponenciální počet zpráv.

Existují i další možnosti řešení, které mají specifické předpoklady a omezení:

- HoneyBadgerBFT (digitální podpisy)
- Blockchain (cena)
- MiniBFT (specifický HW)
- Kryptografické protokoly (důvěra)

3.8.4 Raft algoritmus

Jedná se o moderní a široce používaný algoritmus (slouží jako náhrada Paxosu).Pracuje s *fail-noisy* modelem. Shoda systému probíhá pomocí distibuovaného logu operací (uzly tedy musí být ve stejném stavu). Skupina uzlů má lídra, ten rozhoduje o commitech do logu a v případě výpadků může být nahrazen. Využívá se většinové kvórum ($n > 2 \cdot m$).

Stavy uzlů: lídr, následovník, kandidát.

Volba lídra byla zmíněna už dříve. Je nutné ještě vyřešit jako funguje commit do distribuovaného logu.

Klient vždy posílá požadavek na lídra (případně na něj může být přesměrován z jiného uzlu). Tedy lídr má vždy přehled o všech operacích (i nevyřízených). Každý požadavek je zapsán do logu ve formě $\langle o, t, k \rangle$, kde t je aktuální term a k je index požadavku o v tabulce lídra. Raft garantuje, že operace které prošly commitem byly provedeny většinou uzlů a výsledek byl navrácen klientovi.

Přijetí nového požadavku a distribuce logu

1. Po obdržení požadavku lídr rozšíří svůj log (délky n) o $\langle o, t, n+1 \rangle$
2. Lídr celý log odešle všem uzlům společně s indexem poslední commitované operace c
3. Každý příjemce si svůj log upraví podle přijatého, potvrdího lídrovi a zkontroluje, že všechny operace až do c včetně byly provedeny (o zatím nemůže být commitováno)
4. Po přijetí potvrzení od většiny uzlů lídr vykoná operaci o a navrátí výsledek
5. Lídr nastaví c na $n + 1$
6. Při dalším požadavku si ostatní uzly zkontrolují ono c

3.8.5 Paxos algoritmus

Autorem je Lamport. Stejně jako Raft využívá *fail-noisy* model. Často používaný, ale komplikovaný, takže byl nahrazen raftem. Stejně jako raft využívá **většinové kvórum**.

Typy procesů: *client, proposer, leader, learner, acceptor.*

Vnitřní dělení uzlů: *proposer, acceptor, learner.*

Předpoklady pro fungování (slabé): DS může být nesynchronní, umožňuje nespolehlivé spoje, poškozené zprávy jsou detekovatelné, operace jsou deterministické, nejsou povoleny náhodné chyby.

Jádro algoritmu funguje ve 2 krocích – fáze prepare a fáze accept. Pro návrh se vždy vybírá náhodné číslo, které se používá k rozhodování v případě více návrhů (větší číslo vyhrává). Návrhy podávají proposers a acceptors pro ně hlasují (jeden uzel může zároveň plnit více rolí).

[Vysvětlení asi v knížce nebo spíš zjednodušeně na internetu]

3.9 Replikace a konzistence

Redundance … více vzájemně nahraditelných uzlů

Replikace … Raft jako replikace stavu

Konzsitence … repliky musí být shodné

3.9.1 Replikace

Pro zvýšení spolehlivosti (pád uzlu, zničení dat) a výkonu. Zahrnuje v sobě redundanci (protože replika vyřeší pád originálu), ale přináší navíc geografickou dsotupnost či rozložení zátěže. Např. kešování, CDN, DNS. Tyto vylepšení sebou ale nesou i negativa, především jak udržet více uzlů (míst) v konzistentním stavu.

3.9.2 Řetězová replikace

Uzly jsou uspořádány do řetězce (hlava --> vnitřní uzly --> ocas). Operace zápisu se provádí na hlavu a operace čtení na ocase. Zápis se poté postupně propraguje od hlavy až na ocas. Tento přístup zajistí **silnou konzistenci** (linearizovatelnost). Musíme řešit správu uzlů a jejich selhání, ve *fail-stop* modelu následující:

- hlava → nahradí se následovníkem
- ocas → nahradí se předchůdcem
- uzel uprostřed → řeší centrální uzel přepojením řetězce
- centrální uzel → nutná replikace řetězce

Koncepčně jde o jiný přístup než předchozí – není zde lídr, ocas reaguje na čtení rovnou, pomalý uzel může tvořit bottleneck.

3.9.3 Konzistence

Konzistence nás zajímá při několika kopiích jednotlivé uzlu či při proložemých `read` a `write` operacích. Existuje několika typů, které kladou různé podmínky pro splnění (silná, eventuální, sekvenční, klauzální, …).

Silná (striktní) konzistence … změny propagovány okamžitě; jako kdyby všichni reagovali stejně

Sekvenční konzistence … výsledný stav odpovídá nějakému sekvenčnímu pořadí operací

Kauzální konzistence … pokud platí $a \to b$, pak všichni vidí nejdříve výsledek a a až potom b; u konkurenčních (nekauzálních) operací to je jedno

Eventuální konzistence … jednou dostaneme aktuální data, ale nikoliv hned (např. kešování); povolení read-write konfliktů; write-write konflikty záléží na implementaci

Kdy je konzistentní systém — **Poznámka 12**

Pokud jednotlivé uzly DS reagují (dostatečně) stejně.

3.9.4 Několik teorému na závěr

CAP teorém — **Teorém 13**

Uvažujme DS s replikací dat. Pozorováním zjistíme, že síťovým problémům a následnému rozpadnutí sítě se nelze vyhnout. Když nastane rozpad můžeme zachovat buď dostupnost (avilability) nebo konzistenci (consistency), nikdy ne obojí.

PACELC teorém **Teorém 14**

Rozšíření CAP torému. Jde o promyšlenou zkratku **P** paritition, **A** avilability, **C** consistency, **E** else, **L** latency, **C** consistency. Pokud dojde k rozdělení **P** sítě musíme vybrat mezi dostupností nebo konzistencí, jinak (běžný provoz) volíme mezi latencí a konzistencí.

Různé varianty prostě podle vhodného rozdělení písmen.

Např. PostgreSQL distr. má PC/EC u NoSQL je to dost rozmanité.

CALM teorém **Teorém 15**

Consistency as Logical Monotonicity. Říká co můžeme mít bez drahé koordinace. Nedíváme se na vlastnsoti systému, ale na vlastnosti programu. **Monotónní systém lze implementovat bez synchronizace při zachování konzistence**. Monotónnost je pokud se data pouze přidávají a vrství na sebe (pokud se něco jednou přídá už se to nemůže změnit).

Monotónní (povolené) operace: přidání do seznamu, sjednocení, ...

Nemonotónní (zakázané) operace: minimum, maximum, množinový rozdíl, ...

3.10 Globální stav v DS

Globální stav celého DS se skládá z lokálních stavů jednotlivých uzlů a kanálů mezi nimi (zachycuje celý systém v jednom okamžiku). **Lokální stav** uzlu (procesu) jsou hodnoty proměnných, alokované zdroj a stav vykonávání programu. Události v systému můžeme rozdělit na 2 skupiny, na ty co se již staly a ty co se ještě nestaly.

Konzistence řezu **Definice 16**

Konzistence řezu C $(a \to b \wedge b \in C) \Rightarrow a \in C$.

Posloupnost konzistentncíh řezů nám vytváří průběh výpočtu v celém DS.

Snapshot **Definice 17**

Globální stav DS v daném okamžiku.

3.10.1 Chandy-Lamport algoritmus

Algoritmus umožňuje vytvořit snapshot (zjistit stav) DS bez nutnosti zastavení DS.

1. Iniciátor uloží svůj stav a pošle zpráv s markerem *snapshot request* a začne zaznamenávat všechny zprávy na dalších kanálech
2. Proces obdrží zprávu s markerem
    1. úplně poprvé: uloží svůj stav, pošle jej iniciátorovi, dále posílá marker s každou zprávou, začne sledovat stavy ostatních kanálů
    2. poprvé na daném kanálu: uloží stav kanálu a pošle jej iniciátorovi a končín na něm monitorování
3. Proces s markerem dostane zprávu bez markeru → musí být z okamžiku před snapshotem, uloží ji do stavu kanálu

Kanály musí být typu FIFO. Kanály musí být jednosměrné.

3.10.2 Dijkstra-Scholten algoritmus

Algoritmus zajišťuje detekci ukončení. Uzel má 2 stavy – *aktvní* nebo *skončil*. Zprávy `SIGNAL` (výzva počítáš/nepočítáš?) a `ACK` (odpověď už nepočítám nebo se hlásím jinam). Algoritmus vytvářří v reálném čase stromovou strukturu. Algoritmus funguje pouze pro **difuzní výpočty**, což jsou výpočty, které začínají u jednoho uzlu.

Budování stromu:

1. Vytvoření hrany rodič – potomek pokud uzel A pošle zprávu B a B byl do té doby pasivní, uzel A se stává rodičem B
2. Každý uzel si udržuje čítač, který říká kolik zpráv poslal sousedům a ještě nebyly potvrzeny
3. Poslání `ACK` uzel pošle rodiči pouze pokud on sám je pasivní (dokončil práci) nebo obdržel potvrzení od všech kterým sám poslal zprávu
4. Výpočet je v celém systému považován za ukončsný pokud se kořen stromu stane pasivní nebo pokud má čitač kořenu hodnotu 0

3.11 Distribuované transakce

Jedná se o transakce zahrnující více uzlů v DS. Obecná změna více úložišť/systémů (např. mazibankovní převod, zápis do více různých DB). Zakládá se na klasických transakcích (nutná speciální primitiva). Držet se ACID. Nejvíce nás bude zajímat **distribuovaný commit**, což je závěrečná fáze distribuované transakce. Takové zmněny se musí provést buď všude nebo nikde (prostě jako klasická transakce). Ke commitům je potřebný koordinátor.

3.11.1 Jednofázový commit

Vlastně se nejedná o nic co by zaručovalo správné provedení. Jde o úplně naivní řešení, kdy koordinátor pošle commit všem uzlům instrukci a ti ji hned provedou. Jenže pokud vypadly nebo k nim zpráva nedorazí dostane se systém do nekonzistentního stavu.

3.11.2 Dvoufázový commit

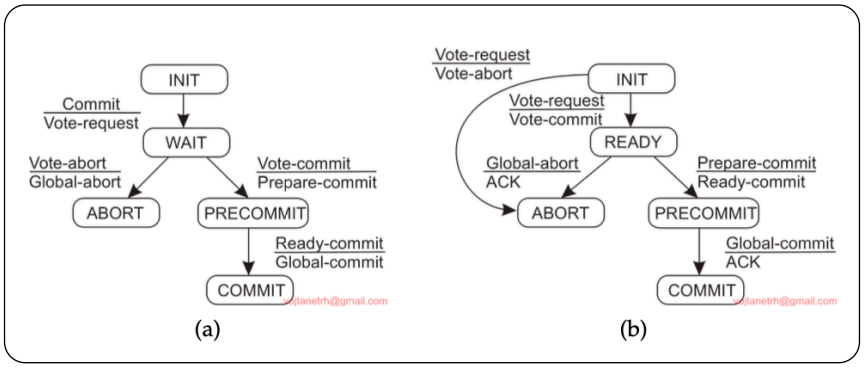
Již se jendá o použitelné rozumné řešení. Jsou 2 typy procesů – *koordinátor* a *následovníci*. Nedokáže reagovat na selhání koordinátora. Nejvíce se používá (např. PostgreSQL `PREPARE TRANSACTION` a `COMMIT/ROLLBACK PREAPRED`).

1. Koordinátor pošle všem `VOTE_REQUEST`
2. Uzel na něj zareaguje pokud je připraven tak `VOTE_COMMIT`, pokud ne `VOTE_ABORT`
3. Kooridnátor při obdržení od všech `VOTE_COMMIT` zašle `GLOBAL_COMMIT` (říká uzlům že mají lokálně provést commit), jinak `GLOBAL_ABORT`

3.11.3 Třífázový commit

Řeší nedokonalosti dvoufázového. Nikdy nelze volit rovnou `COMMIT` nebo `ABORT`. Přibyde tedy nový stav a zpráv (`PRECOMMIT` a `PREPARE_COMMIT`).

1. Koordinátor pošle dotaz jestli uzly mohou provést commit
2. Uzly odpoví `YES` nebo `NO`
3. V případě, že koordinátor neobdrží od všech `YES` proces končí, jinak rozesílá zprávu `PRE_COMMIT`, které zajistí zápis do logu a uzamčení zdrojů, ale operace se ještě neprovádí a odpověď uzlů
4. Pokud od všech opět pozitivní opověď rozesílá se samotná zpráva `GLOBAL_COMMIT`
5. Uzly provádí operace



Obrázek 13: Stavy třífázového commitu pro (a) koordinátora a (b) následovníka.

Pokud koordinátor vypadne uzly se domluvíme mezi sebou, buď čekají na zotavení nebo proces ukončí. Pokud vypadne následovník, tak koordinátor čeká a je vyžadováno zotavení u následovníka.

4 Blockchain

Umožňuje registrování distribuovaných transakcí, proto se také nazývá *Distributed Ledger Technology* (DLT). Blockchain všechny věci (datová struktury, operace) k tomu nutné implementuje. Obecně pracuje s tím, že nepoužívá důvěryhodnou centrální autoritu. Nemusí být vždy veřejné, ale mohou mít nějakou míru centralizace (hybridní) či být privátní.

Základy jsou v mnoha oblasatech: kryptoměny (Bitcoin, Ethereum, Litecoin, …), DNS, IoT, smart contracts.

Už z názvů vyplývá, že se jedná řetězec navzájem provázaných bloků. Každý blok obsahuje: data, timestamp, hash dat (obvykle kryptografický), hash předchozího bloku (taky kryptografický) a eventuálně další specifické věci. První blok se nazývá **genesis block** a nemá předchůdce (takže nemůže mít ani jeho hash) – v tom je odlišný. Hash dat v bloku musí být snadné ověřit (takže něco jako SHA-256).

4.1 Bloky

Mají rekurzoivní vlastnost (každý obsahuje hash předchozího kromě genesis bloku). Díky tomuto není možné předchozí bloky měnit a narušit strukturu. Při změně bloku nebude sedět hash ve všech následujících, takže je to snadno odhalitelné. Každý uzel může mít kopii celého blockchainu, ale stačí mít shodu na posledním bloku. Může existovat více konkurenčních bloků, což vede k větvení a pravděpodobnostní volbě.

Vytvoření nového bloku vyžaduje velké množství práce (model **proof of work**):

1. zájemce posbírá transakce do bloku
2. sestaví základ bloku
3. musí blok doplnit tak, aby vyřešil výpočetně náročnou úlohu (úloha má upravitelno obtížnost na základě aktuálních možností sítě)

Ten kdo to zvládne první získá odměnu za blok. Následné ověření toho, že blok je validní musí být triviální. Problémy se mohou vyskytnout se spotřebou, tvrbou poolů či rychlostí.

Změna bloku **Poznámka 18**

Změna jediného bloku by vyžadovala přepočítání všech dalších, což je vzhledem k nákladům na vytvoření každého **nereálné**.

Problémy přístupu PoW se snaží vyřešit **proof of stake**. Místo vytvoření velkhé množství práce se požadavuje *zástava*.

1. zájemce nabídne zástavu (kryptoměnu daného blockchainu)
2. systém vybere ze zájemců
3. vítěz posbírá transakce a připojí je do sítě

Ostatní validátoři ověří platnost bloku (za to získávájí odměnu). Pokud je blok OK vítěz získá odměnu, pokud nikoliv tak ztratí zástavu a je v síti penalizován.

Vzhledem k tomu, že podvod vyžaduje mít hodně prostředků jednalo by se v podstatě útok na sebe sama. Nevýhodou je centralizace bohatství a moci.

4.1.1 Další typy shod

Delegated PoS – volená malá skupina validátoru, hlas má váhu podle majetku

Proof of Authority – malá důvěryhodná skupina validátorů

Proof of Elapsed Time – nutný důvěryhodný HW a robustní systém práce s časem

Practical Byzantine Fault Tolerance

4.2 Merkle tree

Jedná se o speciální datovou strukturu hašového binárního stromu. Slouží k verifikaci obsahu velkých datových struktur. Používá se v: Bitcoinu, Ethereu, gitu, BitTorrentu, btrfs, …

Sdílená paměť

$nA, nB \leftarrow 0$

A	**B**
opakuj:	opakuj:
A_1: nekritická sekce	B_1: nekritická sekce
A_2: $nA \leftarrow nB + 1$	B_2: $nB \leftarrow nA + 1$
A_3: await $nB = 0$ or $nA \leq nB$	B_3: await $nA = 0$ or $nB \leq nA$
A_4: kritická sekce	B_4: kritická sekce
A_5: $nA \leftarrow 0$	B_5: $nB \leftarrow 0$

Obrázek 14: Merkle tree.

4.3 Problémy blockchainu

Poskládán z 5 vrstev:

1. Infrastruktura (HW)
2. Síťová vrstva – změna uzlů, šíření a ověření informace
3. Datová vrstva – bloky a transakce
4. Shoda –
5. Aplikační nástavby – kryptoměn, decentralizované aplikace, smart contracts

Řeší se konzistence, bezpečnost a především anonymita.

4.4 Hrozby blockchainu

1. Ovládnutí sítě (51% útok)
2. Sybil(a) útok – vydávání se za více účastníků
3. Eclipse útok – oddělení uzlu
4. Útoky na podpůrné vrstvy
5. Útoky na aktuální kryptografii pomocí kvantových počítačů
6. DDoS
7. Sociální inženýrství

4.5 Bitcoin

Autor je Satoshi Nakamoto (synonym) jehož reálná identita je neznámá. Vymyšleno jako alternativa k fiat měnám. Bitcoinů je fixní počet (21 milionů). Na začátku odměna 50 BTC v bloku, ale každých 210 000 bloků ($\approx$ 4 roky) nastává havling (půlení odměn), kdy očekávaný konec je asi v roce 2140.

1 BTC = 10^8 Satoshi

Jako hashovací funkci používá SHA-256 a podpis transakcí zajišťuje ECDSA. Nový blok se vytvoří asi každých 10 minut. Dříve velikost 1 MB, dens asi 4 MB.

4.6 Ethereum

Založil ji Vitalik Buterin v roce 2015. Záměr byl takový, že blockchain může poskytovat více než jen měnu, takže přidal decentralizované aplikace, smart contracts. Narodzíl od BTC nemá limit, ale používá jiné metody pro omezení inflace.

Jako hashovací funkci používá Kecak-256 (modifikovaný SHA-3) a pro podpis transakcí stejně jako Bitcoin ECDSA. Původně využíval PoW, v roce 2022 přechod na PoS, což snížilo spotřebu o 99 %.