KOMUNIKUJÍCÍ SYSTÉMY

Systémy s předáváním zpráv

- Dvě základní primitiva
- Párová komunikace
  - *send(channel, msg)*
  - *receive(channel, msg)*
- Kolektivní komunikace
  - *broadcast (vícenásobný send po více komunikačních kanálech)*
  - *reduce (vícenásobný příjem s provedením operace)*
- **Komunikační kanál**, buffering
  - *Asynchronní kanál*
    - neomezená kapacita
    - operace `send` je neblokující
    - složitější implementace
    - příklad: e-mail
  - *Synchronní kanál*
    - omezená kapacita X nulová kapacita
    - `send` může být blokující
    - počet příjemců
      - *Jeden příjemce - kanál*
      - *více příjemců - mailbox*

VŠESMĚROVÉ VYSÍLÁNÍ

Všesměrové vysílání, n-tity

m: zpráva z množiny zpráv msgs

operace **b'cast(*m*)**, **deliver(*m*)**

Každá zpráva obsahuje následující položky:

– **sender(*m*)**, identita odesilatele

– **seq(*m*)**, sekvenční číslo

- **sender(*m*)** = *p* and **seq(*m*)** = *i značí, že zpráva m* je *i*-tou zprávou zaslanou procesem *p*

Nekorektní procesy

- Nekorektní procesy jsou takové, které mohou po spuštění selhat
- Abstrakce nekorektních procesů:
  - ***Crash & stop*** – po selhání procesu neprovádí žádnou činnost (zastaví se)
  - ***Crash & recovery*** – po nějáké době může dojít k obnovení jejich správné činnosti
  - ***Byzantine*** – byzantské procesy, pro které není chování po selhání definováno

Vlastnosti paralelních programů

- V paralelních a tedy i distribuovaných systémech požadované vlastnosti mohou patřit mezi **životné (liveness)**, nebo **bezpečné (safety)**
- Požadujeme odpověď na požadavek? Chceme, aby dříve nebo později nastala událost odpovědi → **životná podmínka**
- Chceme, aby byly zprávy doručovány v nějakém pořadí? Nemůže nastat chyba v doručení → **bezpečná podmínka**

Spolehlivé všesměrové vysílání

- **Platnost** (**Validity**) – pokud zpráva byla nějakým korektním procesem vysílána, pak dřív nebo později ji každý korektní proces doručí
- **Shoda** (**Agreement**) – pokud zpráva byla doručena korektním procesem, pak dřív nebo později bude doručena každým korektním procesem
- **Integrita** (**No duplication nebo Integrity**) – žádná zpráva není doručena více než jednou
- **Opravdovost** (**No creation**) – pokud process doručil zprávu **m** od procesu **p**, potom tento proces zprávu opravdu odeslal
- Budme pracovat s **korektně** se chovajícími procesy, i s **nekorektními**, které mohou být ukončeny během komunikace (crash-stop procesy)

“Best effort” všesměrové vysílání

b’cast(*m*):

Tag *m* with sender(*m*) and seq(*m*)

send(*m*) to all neighbors including self

deliver(*m*):

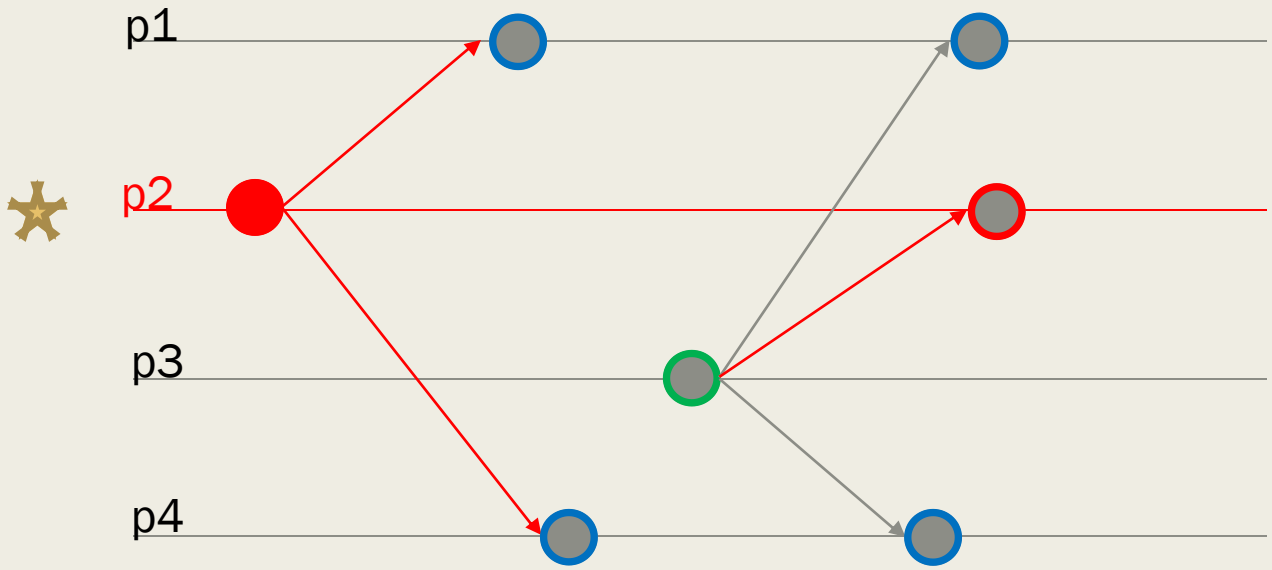
upon receive(*m*) do

deliver(R, *m*)

enddo

„Best Effort” všesměrové vysílání

- Je zde garantována **platnost** ale ne **shoda**
- Kdy může nastat situace, že platí **platnost**, ale neplatí **shoda**? V systému nemusí existovat pouze korektní procesy. Pak **vysílá** jen jeden korektní proces p_3 a od toho **doručí** zprávu všechny korektní procesy p_1 a p_4
- Proces p_2 není korektní, proto **platnost** nenaruší to, že zpráva není doručena
- Zpráva od p_2 byla **doručena** korektními p_1 a p_4, pak pro **shodu** musí být **doručena** i p_3, což neplatí! Nedoručení zprávy od p_3 procesu p_2 ovšem shodu nenaruší



Předpoklady pro komunikaci při všesměrovém vysílání

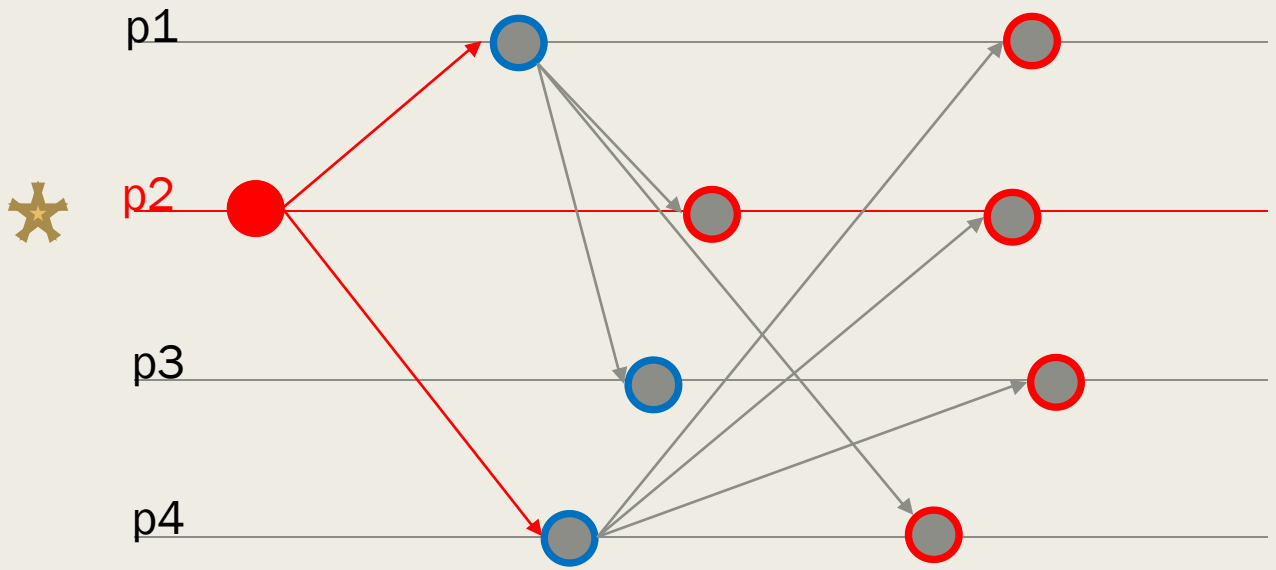
- Komunikace
  - *Známé největší možné zpoždění při doručování zprávy*
  - *Lokální hodiny pro každý z procesů*
  - *Známý nejvyšší časový limit pro vykonání interní akce procesem v jednom kroku.*
- Topologie
  - *Topologie sítě je obecná*

Spolehlivé (reliable) všesměrové vysílání

```
b'cast(R, m):
        Tag m with sender(m) and seq(m)
        send(m) to all neighbors including self

deliver(R, m):
        upon receive(m) do
                if p has not previously executed deliver(R, m) then
                        if sender(m) != p then send(m) to all neighbours
                        deliver(R, m)
                endif
        enddo
```

Spolehlivé všesměrové vysílání

- Procesy přeposílají obdržené zprávy
- Je zde garantována **platnost** i **shoda**



Další vlastnosti všesměrového vysílání

- **1) FIFO uspořádání:** Pokud nějaký proces vysílá zprávu *m* před tím, než vysílá zprávu *n*, pak žádný korektní proces nedoručí zpráv *n* před tím, než doručí zprávu *m*– *“FIFO order” from single process*
- **2) Kauzální (Causal) uspořádání:** Pokud vysílání zprávy *m* kauzálně předchází vysílání zprávy *n*, pak žádný korektní proces nedoručí zpdávu *n* dokud nedoručí zprávu *m*
- **3) Úplné (Total) uspořádání:** Pokud korektní procesy *p* a *q* oba doručí zprávy *m* a *n*, pak *p* doručíelivers *m* before *n* iff *q* delivers *m* before *n*

Přijímají z pohledu příjemců ve stejném pořadí všechny procesy

Třídy všesměrového vysílání

- Reliable = Validity + Agreement + Integrity
- FIFO = Reliable + FIFO Order
- Causal = Reliable + Causal Order
- Atomic = Reliable + Total Order
- FIFO Atomic = Reliable + FIFO Order + Total Order
- Causal Atomic = Reliable + Causal Order + Total Order

FIFO všesměrové vysílání

init

forall *q*

msgbag[*q*] := empty

next[*q*] := 1

b'cast(F, *m*): **b'cast**(R, m)

deliver(F, *m*):

upon **deliver**(R, *m*) do

q := sender(*m*)

msgbag[*q*] := msgbag[*q*] U {*m*}

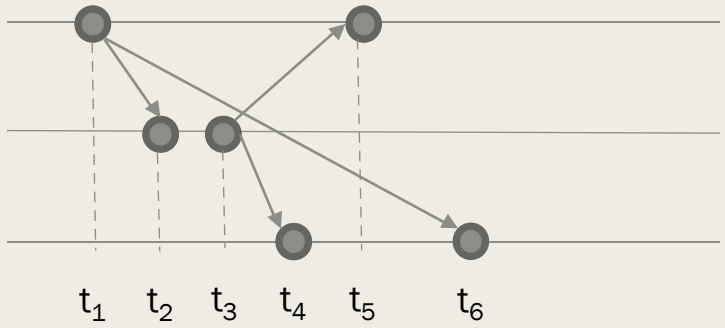
while exists message *n* in msgbag[*q*] with seq(*n*) = next[*q*] do

deliver(F, *n*); next[*q*] := next[*q*] + 1

msgbag[*q*] := msgbag[*q*] - {*n*}

enddo

Kauzální předávání zpráv



t_1: David posílá zprávu Petrovi a Karlovi: „Já a Karel chceme vědět, jak hrála Zbrojovka na Slávii“

t_2: Petr doručuje zprávu od Davida: „ Já a Karel chceme vědět, jak hrála Zbrojovka na Slávii“

t_3: Petr posílá zprávu Karlovi a Davidovi: „2:0“

t_4: Karel doručuje zprávu od Petra: “2:0” *… a Karel je zmaten … o co jde*?

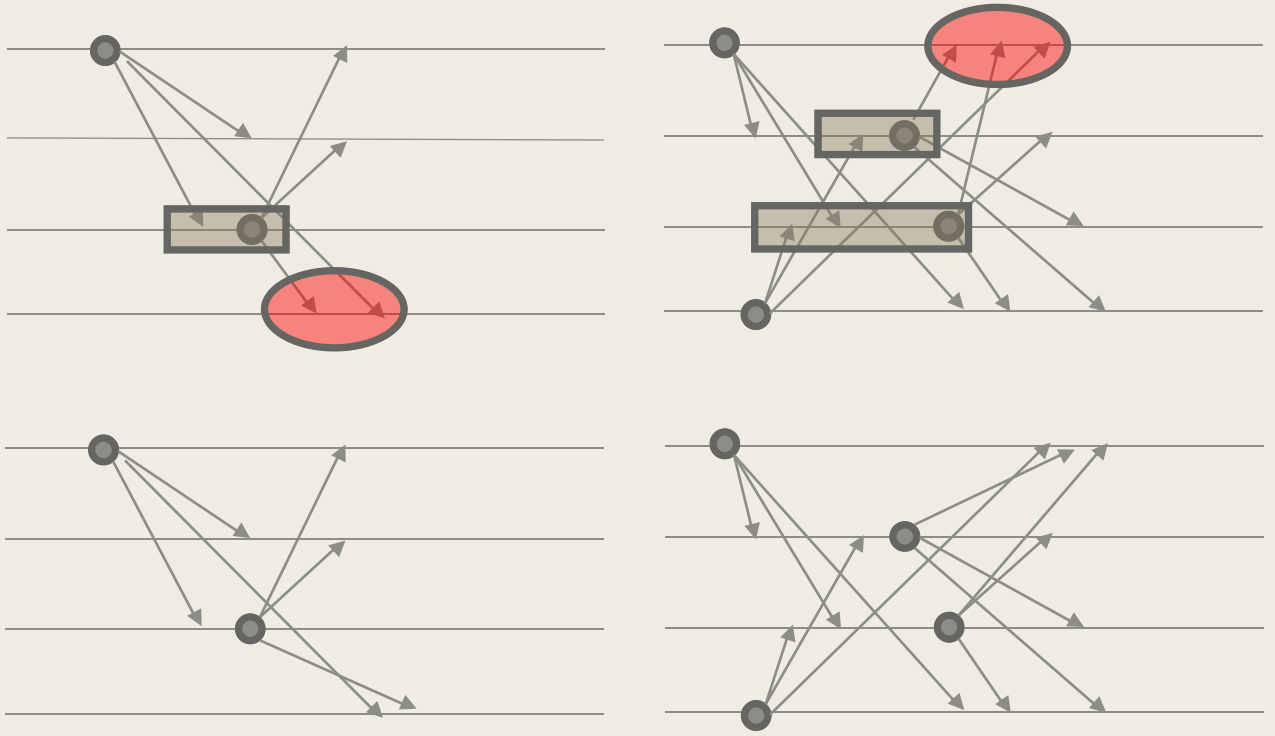
t_5: David doručuje zprávu od Petra: “2:0”

t_6: Karel doručuje zprávu od Davida: „ Já a Karel chceme vědět, jak hrála Zbrojovka na Slávii“

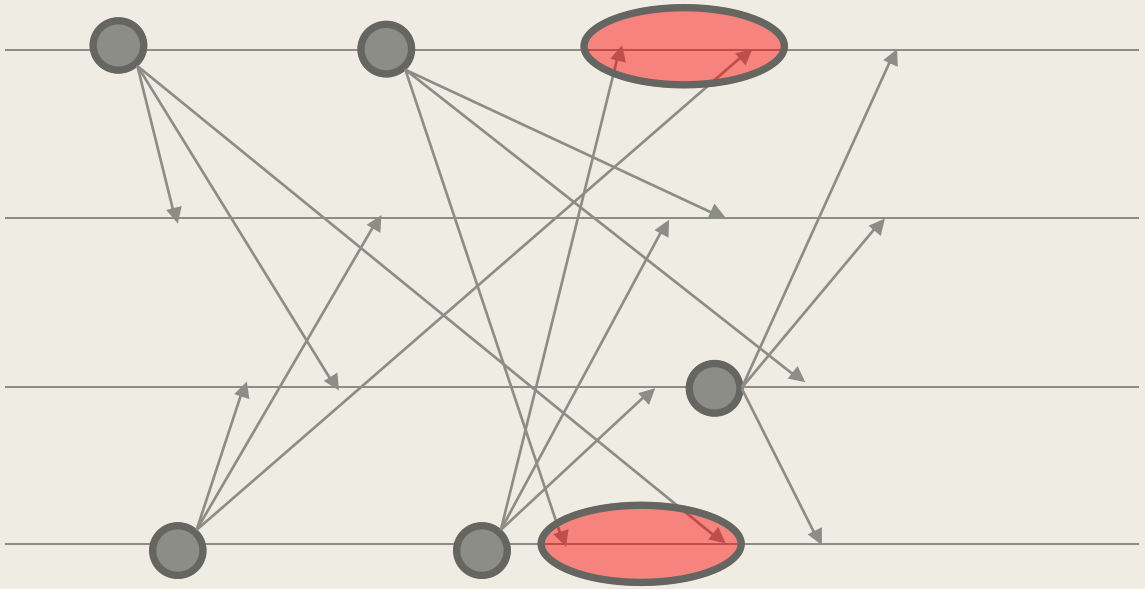
Kauzální všesměrové vysílání

- Zavedeme relace ***kauzality***
- Relace $\rightarrow_e$ kauzálně předchází, když
  - *a* $\rightarrow_e$ *b pokud jeden process vykonal tyto události v tomto pořadí*
  - *broadcast(ma)* $\rightarrow_e$ *deliver (ma)*
  - *Tranzitivita*
- Pokud jeden proces doručí zprávu *ma* a pak vyšle zprávu *mb*, pak všechny procesy musí doručit *ma* před *mb*, nezáleží ale v jakém pořadí doručování zpráv před doručením *mb* probíhá!

Kauzální?



Kauzální, ale ne FIFA (v praxi nepoužívané)



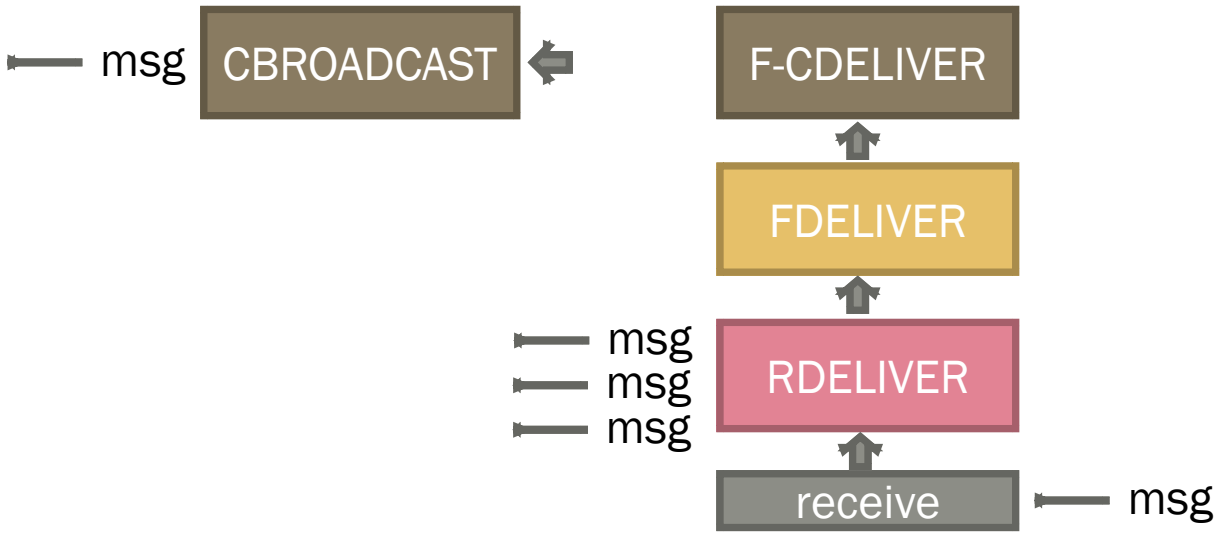
Kauzální všesměrové vysílání

```
init
        prevDlvrs := empty

b'cast(C, m):
        b'cast(F, <prevDlvrs||m>)
        prevDlvrs := empty

deliver(C, m):
        upon deliver(F, <m1, ... mk>) do
                for i := 1 to k do
                        if p has not previously executed deliver(C, mi)
                        then
                                deliver(C, mi)
                                prevDlvrs := prevDlvrs || mi
                        endif
                enddo
```

Kauzální vysílání a doručení

broadcast(C,m) // volá se pro C-bcast

deliver(C,m) // reaguje na přijetí (FIFA-CAUSAL doručení)



Atomické všesměrové vysílání (ABCAST)

- ABCAST odesilatel p zasílá m(p,t) všem ostatním procesům
- Každý příjemce q přidá zprávu do bag[p], označí ji “U” a přiřadí ji hodnotu vyšší **než doposud zjištěné priority** (tedy buď přiřazené tímto procesem dříve, nebo obdržené s rozhodnutím **D**, viz níže)
- Každý příjemce informuje odesilatele o přidělené hodnotě zprávě m, “U” – hodnota odeslaná odesilateli – “D” – hodnota upravená odesilatelem
- Odesilatel sesbírá všechny odpovědi na svoji zprávu, zjistí maximální přiřazenou hodnotu a o té informuje ostatní procesy
- Tyto si hodnotu k procesu poznačí jako „D“
- Doručí všechny zprávy podle hodnot dokud je nějaká zpráva v bagu, nebo nějaká zpráva s aktuálně nejnižší hodnotou je označena jako “U”

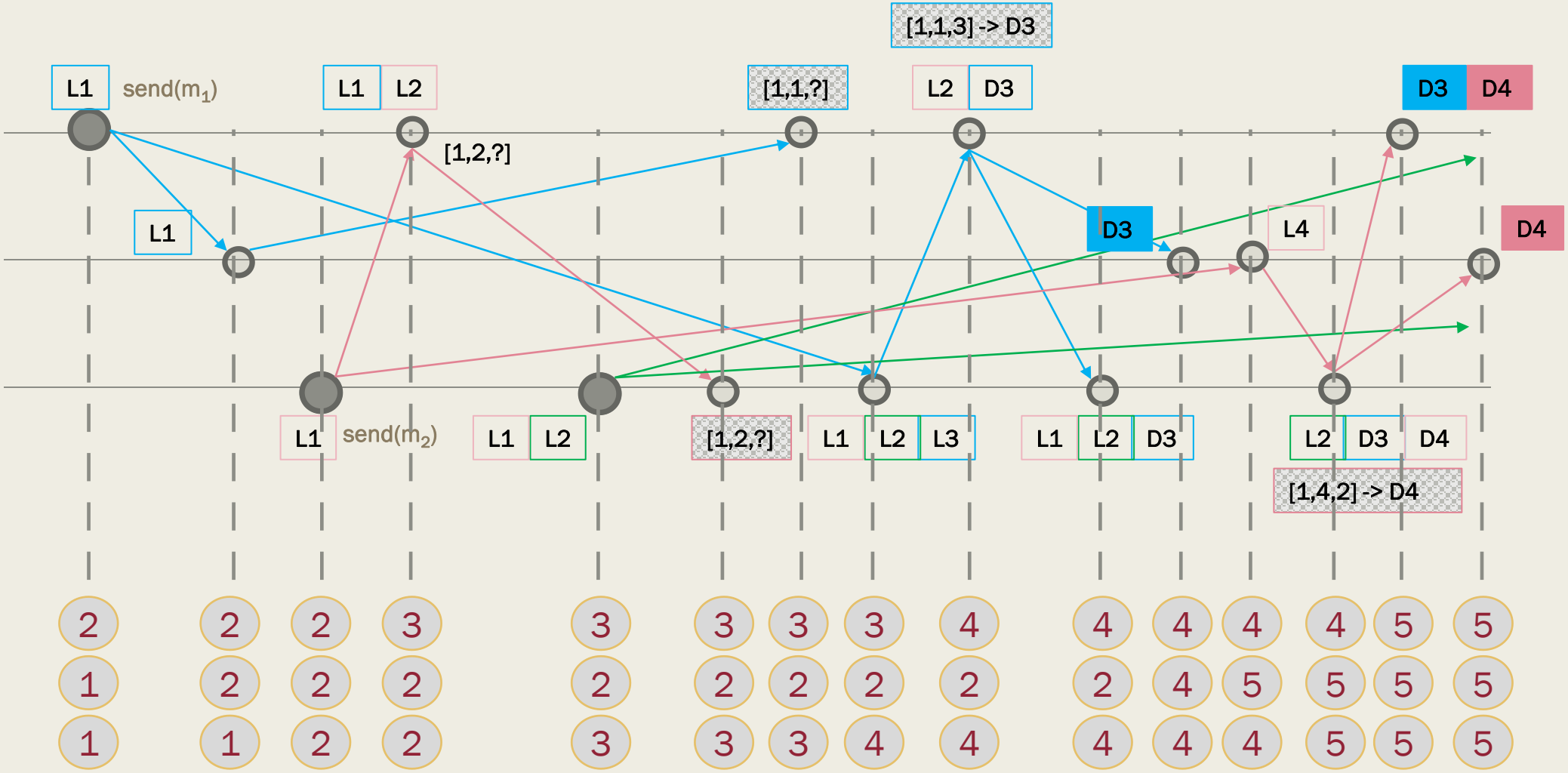
ABCAST – stejná hodnota, rozhodnutí ?

- P1[1]: bcast(m_1,(p_1,1)) ; P2[1]: bcast(m_2,(p_2,1))
- P1[2]: send(P_2,U(m_1,2)); P2[2]: send(P1,U(m_2,2));
- P1[3]: send(P_2,D(m_1,2)); P2[3]: send(P1,D(m_2,2))
- M1 a m2 mají stejnou hodnotu konsensu =>
- V tom případě se doručí podle indexu procesu a časového razítka, které přidělil vysílající proces (tj. sekundární klíče jsou nejprve číslo procesu a pak časové razítko, které dal vysílající proces zprávě).

ABCAST – náznak důkazu

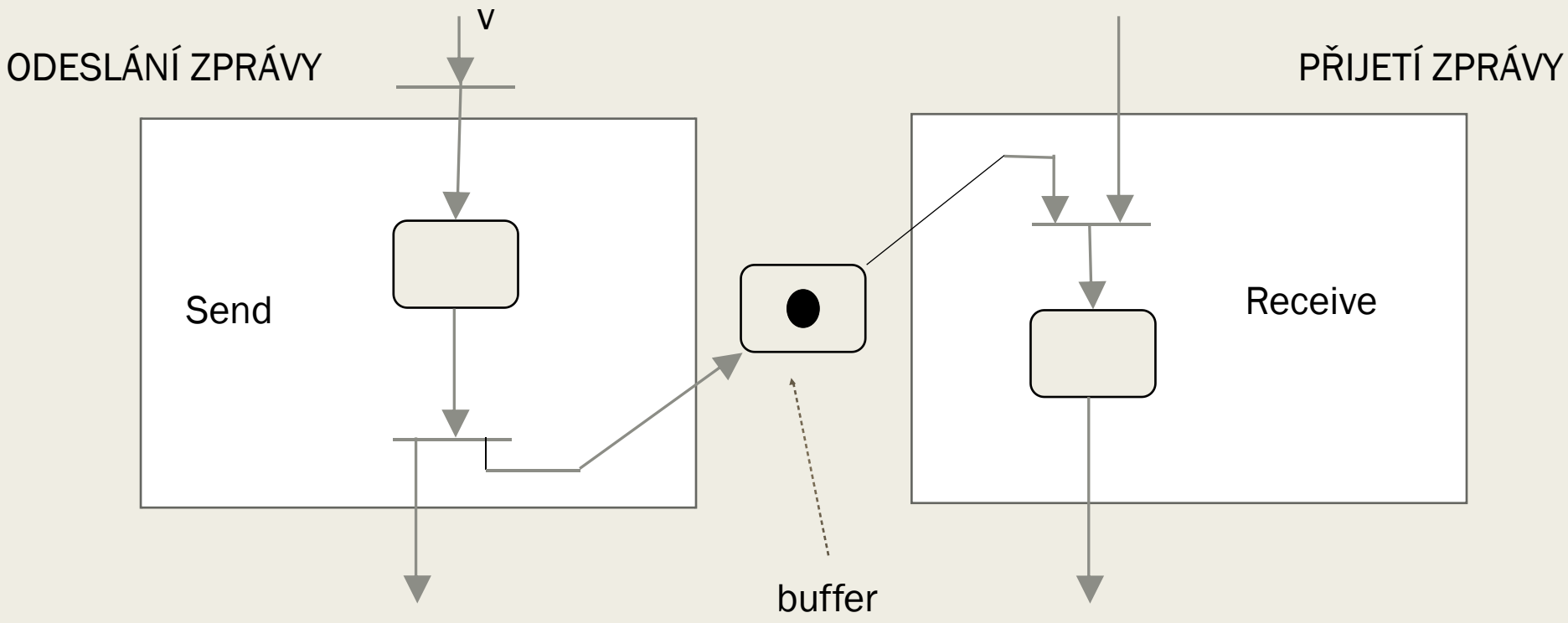
- Zpráva *m* s nejnižším rozhodnutím je nějakým procesem P doručena, a přitom příjde rozhodnutí o zprávě m’, že má vyšší nebo stejnou hodnotu doručení, než má *m (???)*
- *A, tato zpráva ještě nebyla procesu P předána spolehlivým broadcastem, potom ale dostane vyšší číslo U od tohoto procesu (výsledná priorita musí být tím pádem nižší).*
- *B, proces již poslal své U pro tuto zprávu a to je nižší nebo stejné než je D pro zprávu m -> potom musí být ale ve frontě doručovaných zpráv záznam pro m’ před m, což není*
- *Pozn: fronta zpráv k doručení řadí U před L pro stejné časové razítko, nebo alespoň podle sekundárních klíčů!*

ABCAST, příklad

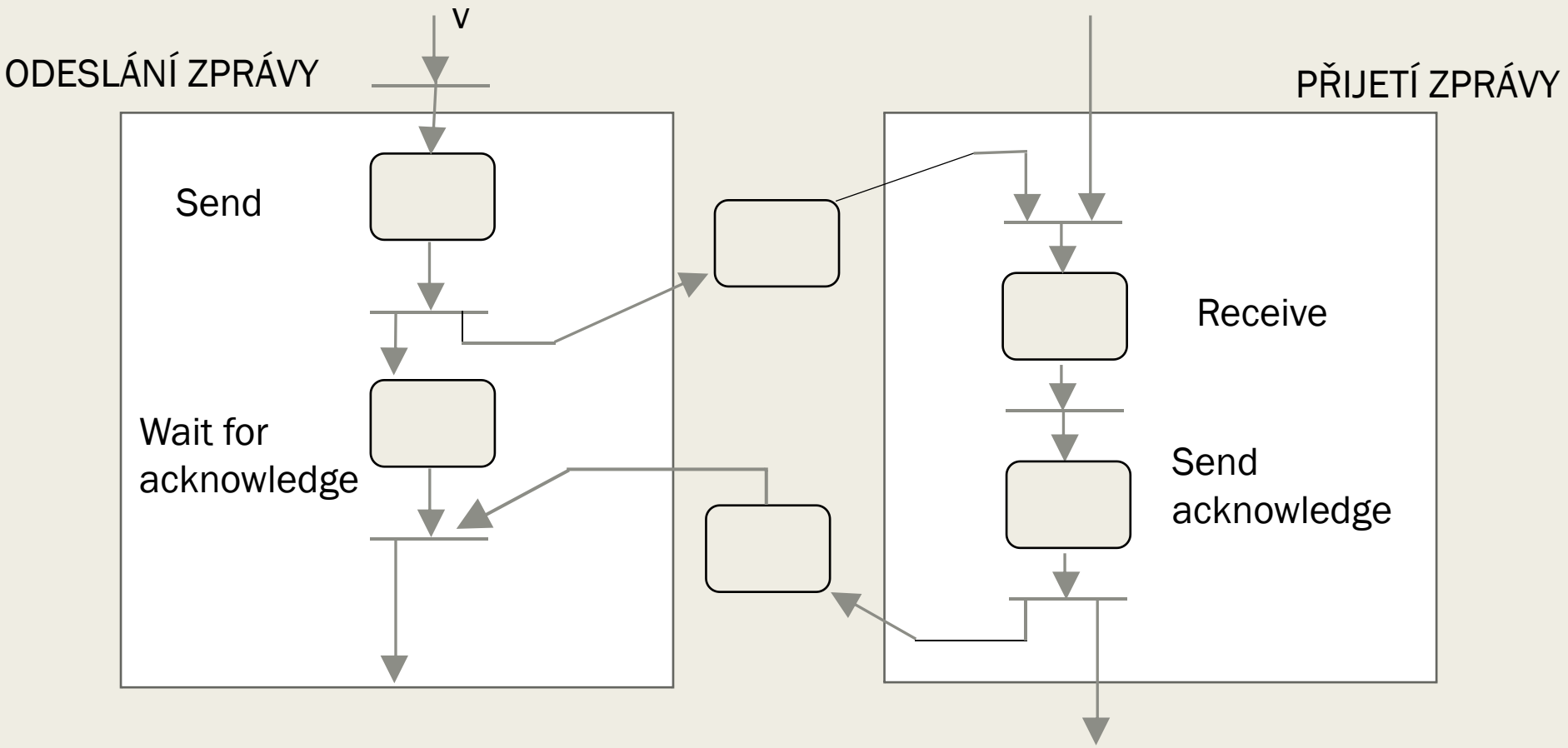


SYNCHRONNÍ KOMUNIKACE

Asynchronní komunikace



Synchronní komunikace



Simulace synchronního kanálu skrz asycnchronní

- Nulová kapacita
  - `send(ch, msg) ⇒ send(ch1, msg)`
    `receive(ch2, dummy)`
  - `receive(ch, msg) ⇒ receive(ch1, msg)`
    `send(ch2, „ack“)`
- Nenulová kapacita (n zpráv)
  - `init(ch) ⇒ n_times {`
    `send(ch2, „ack“)`
    `....`
    `send(ch2, „ack“)`
    `}`

Synchronní komunikace

- **D** je funkce zobrazující množinu událostí v systému na diskrétní množinu ‘fyzického času’
- Pak pro synchronní komunikaci platí
  - ***Platnost*** *– pokud process p1 synchronně obdrží zprávu od p2, pak ji process p2 synchronně odeslal*
  - ***Integrita*** *– žádný proces synchronně neobdrží zprávu víc než jedenkrát*
  - ***Synchronnost***

    Pokud $e1 \rightarrow_e e2$, pak D(e1) < D(e2)

    D(***send***(m)) = D(**receive**(m))
  - ***Ukončení***

    Každá synchronně odeslaná zpráva bude synchronně doručena

Proveditelnost programu v systému se synchronním kanálem

- Synchronní komunikace je kauzální, ale kauzální nemusí být synchronní
- Program pro asynchronní systém (A-execution) je proveditelný v systému se synchronní komunikací (RSC, realizable with synchronous communication) pokud neobsahuje **korunu** (crown)

  Koruna velikosti *k* je, když

  $\mathbf{send}(m_1) \rightarrow_e \mathbf{receive}(m_2)$

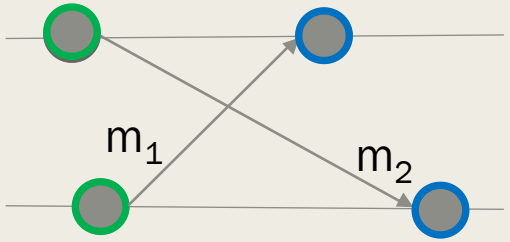
  $\mathbf{send}(m_2) \rightarrow_e \mathbf{receive}(m_3)$

  ...

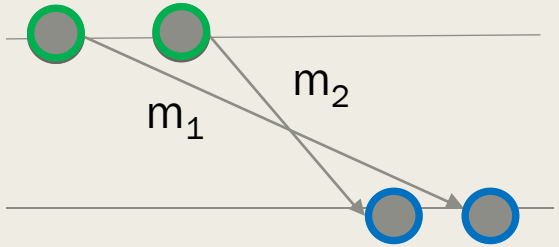
  $\mathbf{send}(m_{k-1}) \rightarrow_e \mathbf{receive}(m_k)$

  $\mathbf{send}(m_k) \rightarrow_e \mathbf{receive}(m_1)$

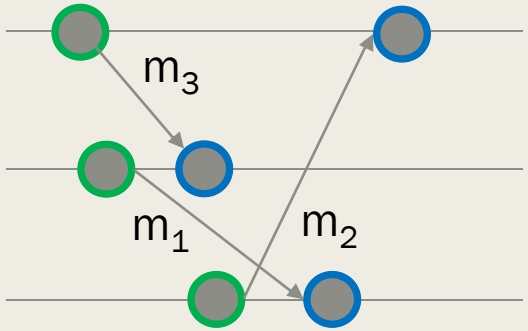
Koruna (příklady velikostí 2 a 3)



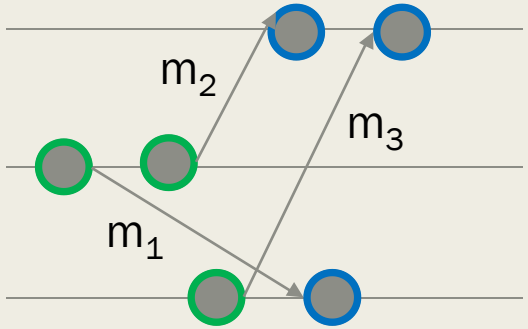
$send(m_1) \rightarrow_e receive(m_2)$
$send(m_2) \rightarrow_e receive(m_1)$



$send(m_1) \rightarrow_e receive(m_2)$
$send(m_2) \rightarrow_e receive(m_1)$



$send(m_2) \rightarrow_e receive(m_1)$
$send(m_1) \rightarrow_e receive(m_3)$
$send(m_3) \rightarrow_e receive(m_2)$



$send(m_3) \rightarrow_e receive(m_1)$
$send(m_1) \rightarrow_e receive(m_2)$
$send(m_2) \rightarrow_e receive(m_3)$

Proveditelnost asynchronního programu synchronním

Program napsaný pro systém s asynchronní komunikací je proveditelný systémem se synchronní komunikací, pokud v něm neexistuje koruna

Pokud ***send***$(m_1) \rightarrow_e$ ***receive***(m_2), pak $(m_1, m_2) \in$ **G**

Jelikož není koruna, pak nemůžou v G existovat cykly:

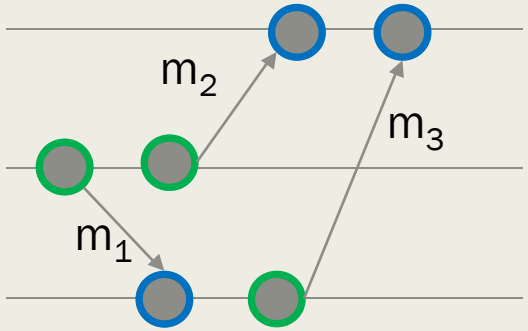
… jakýkoliv cyklus v **G** $(m_1, m_2, m_3 \ldots m_k, m_1)$ znamená, že v **G** jsou hrany

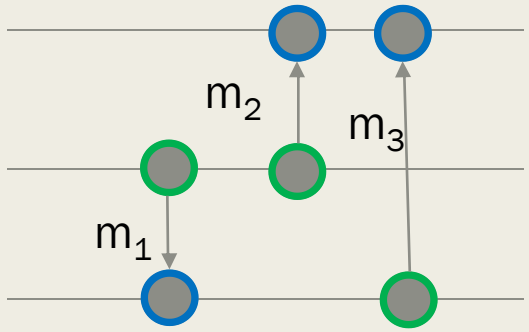
$(m_1, m_2), (m_2, m_3) \ldots (m_k, m_1)$

a tedy i koruna

send$(m_1) \rightarrow_e$ ***receive***(m_2), ***send***$(m_2) \rightarrow_e$ ***receive***$(m_3) \ldots$ ***send***$(m_k) \rightarrow_e$ ***receive***(m_1)

Synchronní provedení asynchronní komunikace bez koruny, příklad





$\text{send}(m_1) \rightarrow_e \text{receive}(m_2)$

$\text{send}(m_1) \rightarrow_e \text{receive}(m_3)$

$\text{send}(m_2) \rightarrow_e \text{receive}(m_3)$

G=$\{(m_1, m_2), (m_1, m_3), (m_2, m_3)\}$ -> splněno pro uspořádání (m_1, m_2, m_3)

Synchronní komunikace klient / server

Klient (process *i*) je odesilatel

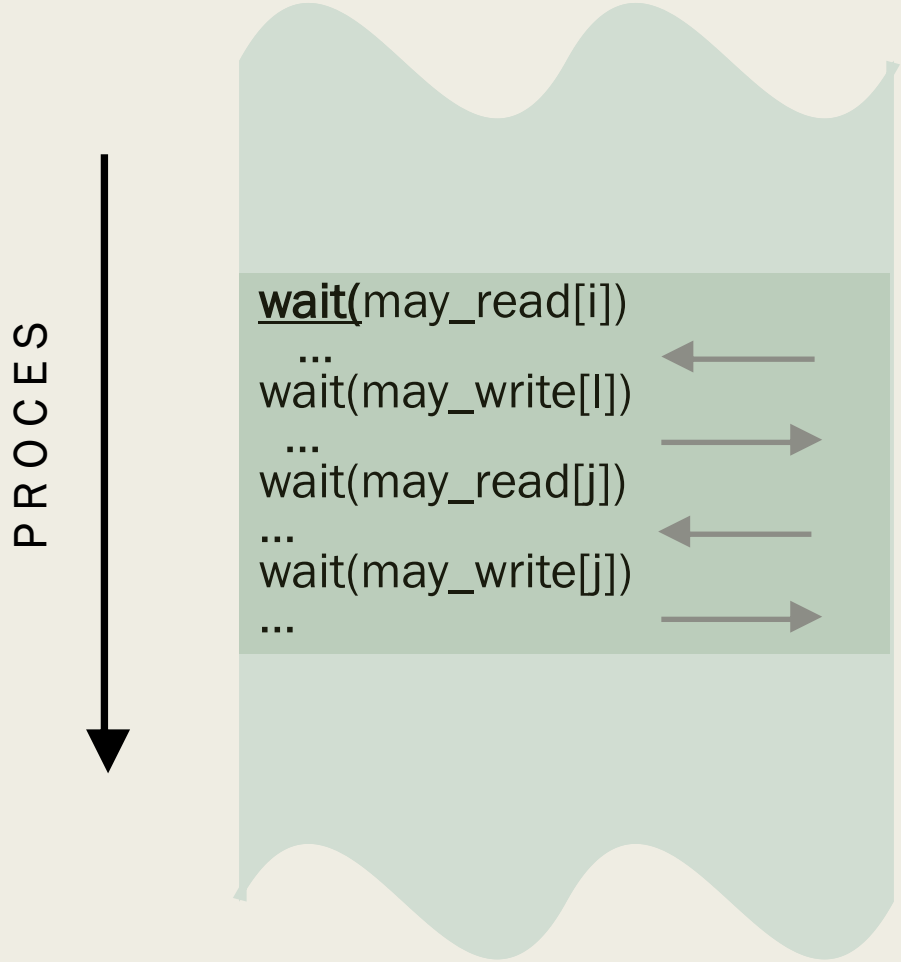
Server	Klient
wait(may_read[i])	buffer ← m
x ← obtain(i)	signal(j, may_read[i])
$may\_read_j[i]$ ← false	$wait(end\_rdv_i)$
signal(i, end_rdv)	end_rdv ← false

Klient (process *i*) je příjemce

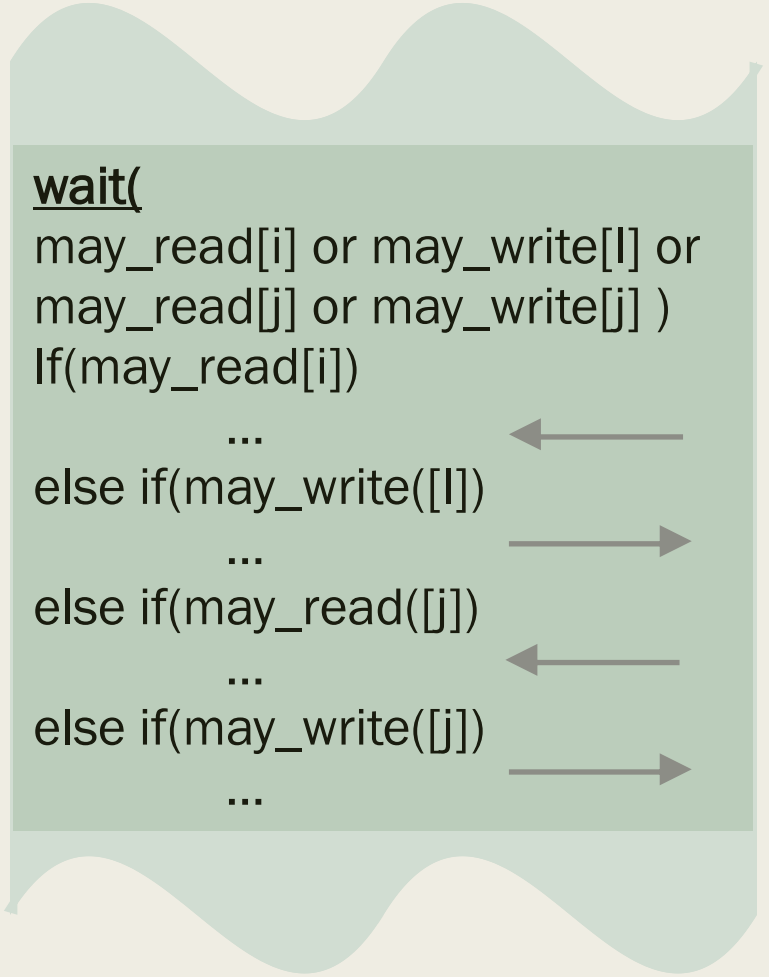
Server	Klient
wait(may_write[i])	signal(j, may_write[i])
delivere(i,m)	$wait(end\_rdv_i)$
may_write[i] ← false	x ← $buffer_i$
signal(i, end_rdv)	end_rdv_i ← false

Rendezvous kontext

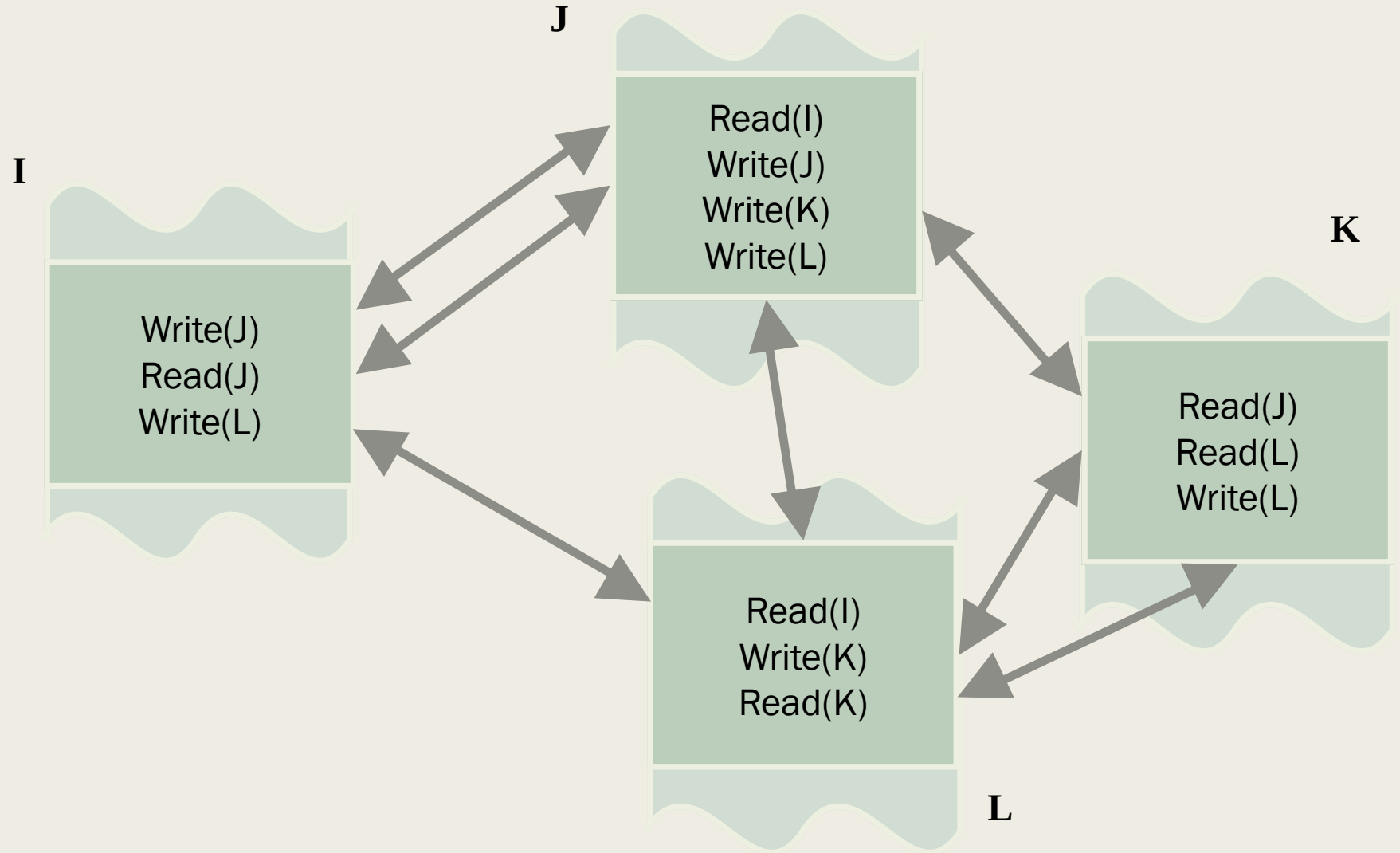
Deterministický rendezvous kontext



Nedeterministický rendezvous kontext



Synchronní komunikace klient / server



Synchronizace rendezvous procesů

- (Bagrodia 1989) - Synchronizace rendezvous procesů, zabránění vzniku koruny,
- Pracuje s tokeny, které reprezentují interakci mezi dvěma procesy
- **Základní princip:**
  - *Aby oba procesy intereagovali (= provedli synchronní komunikaci) musí o to mít oba zájem*
    - Jsou v Rendezvous části program
    - Mají zájem intereagovat s druhým procesem
  - *Právě jeden ze dvou procesů **i** a **j** drží token **TOKEN**({ i, j},i) (token pro komunikaci mezi procesy i a j je držen procesem i. Tento token slouží pro komunikaci v obou směrech, nemáme tedy **TOKEN**({ j, i}, i≠j)*
  - *Procesy jsou prioritně uspořádány, procesy s menším číslem mají větší prioritu*
  - *Proces mimo rendezvous kontext je ve stavu OUT, v tomto kontextu buď ve stavu **INTERESTED** nebo **ENGAGED***
  - *Proces, pokud čeká na návrh interakce, a nějaký jiný proces má zájem o interakci, tak pokud má větší prioritu si tento proces poznačí a odpoví mu až obdrží odpověď, na kterou čeká*

Synchronizace rendezvous procesů

Proces vstupuje do kontextu rendezvous (dále jen kontextu):

1. Proces vstupuje do kontextu, nastavuje svůj stav na "**INTERESTED**"
2. Pokud drží token pro některý poznačený proces, se kterým chce komunikovat, pošle jej tomuto procesu a nastaví svůj stav na "**ENGAGED**"
3. Nastaví záznam o alternativním procesu k interakci na prázdný

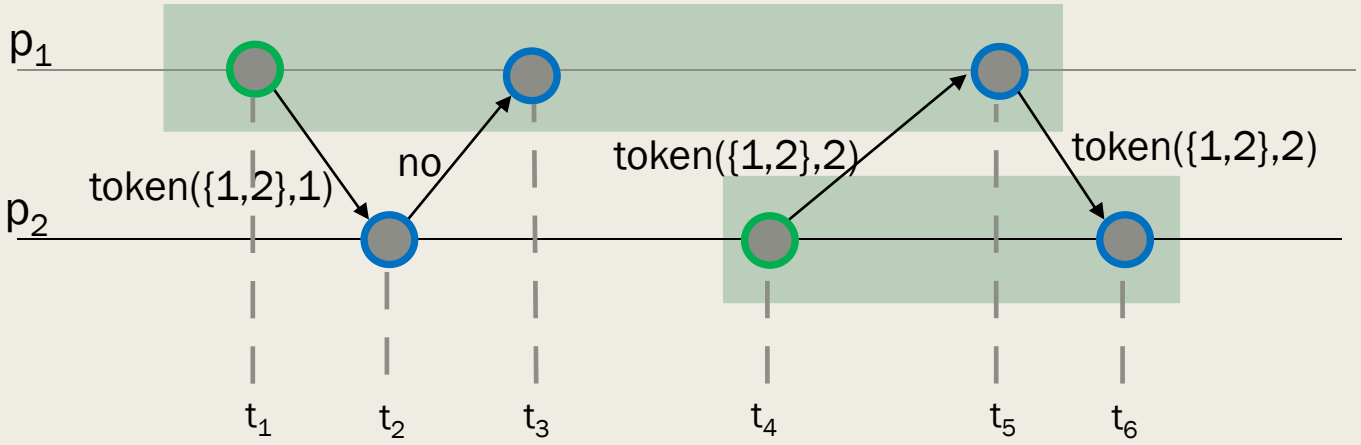
Proces obdržel token:

1. Pokud je mimo kontext, nebo nemá zájem s tímto procesem intereagovat, pošle mu "no", token si ponechá
2. Pokud je ve stavu "**INTERESTED**", pošle token zpět a dojde k interakci – **rendezvous**. Proces je tímto mimo kontext, Stav je "**OUT**"
3. Zbývá možnost, že proces je v kontextu a jeho stav je "**ENGAGED**" (čeká na odpověď na svoji výzvu)
    1. *Pokud je iniciátorem výměny tokenu on sám, pak byla přijata výzva partnerským procesem – **rendezvous**, Proces je tímto mimo kontext, Stav je "**OUT**". Navíc pokud má poznačeno, že by se měl ozvat na později mu doručenou nabídku k interakci, pošle zpět zprávu "no" – již interakci má – a token si ponechá*
    2. *Pokud je iniciátorem proces s nižší prioritou – vyšším číslem, a nemá zatím poznačenou alternativní nabídku, poznačí si tuto a zatím nic neodpoví, ani token nevrací*
    3. *Pokud je iniciátorem proces s vyšší prioritou – nižším číslem, nebo již má poznačenou alternativní nabídku k interakci, odpoví "no" a token si ponechá*

Proces obdržel zprávu "no":

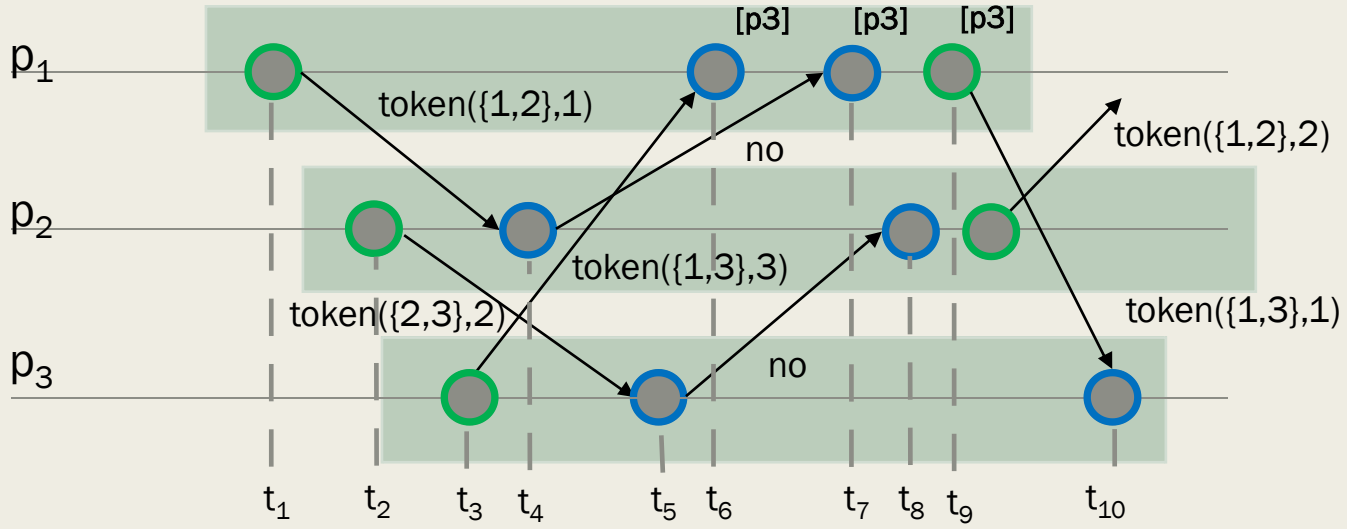
1. Pokud má poznačenou alternativní nabídku, pošle token který s touto nabídkou obrdžel zpět – **rendezvous**. Proces je tímto mimo kontext, Stav je "OUT"
2. Poud nemá, provede opět část "**Proces vstupuje do kontextu**"

Synchronizace rendezvous procesů, příklad



t_1: proces p_1 je v kontextu, vybírá partnera pro komunikaci p_1 a posílá mu token
t_2: proces p_2 není v kontextu a proto odpovídá ‘no’
t_3: proces p_1 nemá token pro komunikaci s žádným jiným partnerem, čeká
t_4: proces p_2 vstupuje do kontextu, vybírá si p_1, token má, a posílá jej partnerovi
t_5: process p_1 má zájem, nečeká na žádnou odpověď na svou žádost, přijímá
t_6: proces p_2 ví, že komunikace je potvrzena … **rendezvous**, pak opouští oba kontext

Synchronizace rendezvous procesů, příklad



t_1, t_2 ,t_3 : procesy vstupují do kontextu, vybírají si partnera, pro kterého drží token a ten odesílají
t_4,t_5: procesy s nižší prioritou p_1 a p_2 odpovídají ‘no’, a token si ponechávají
t_6: proces s vyšší prioritou p_1 neodpovídá, ale značí si alternativní možnost interakce
t_9: proces p_1 obdrží ‘no’ na svoji původní návrh a bere zavděk alternativní nabídce, posílá token
t_{10}: odesilatel alternativního návrhu obdrží token, **rendezvous**

NEDOCHÁZÍ K DEADLOCKU, ANI K ODMÍTNUTÍ INTERAKCE VŠEMI PROCESY

ADA - programovací jazyk

- Vyvíjen od 80. let jako ‘komplexní’ jazyk poskytující, robustní a obecný nástroj pro progarmování systémů
- Strukturovaný, objektový, výjimky, AGOL + PASCAL + …
- Zahrnující vše, co se zdá být užitečné, ale přesto není stříbrnou kulkou, která vyřeší vše, co je třeba (nejen vlkodlaka)

(viz např zde http://www.cs.unc.edu/techreports/86-020.pdf)

ADA LOVELACE

ADA

- **ADA pro synchronizaci používá systém ‘rendezvous’ (aktivní zprávy)**
  - *V Ada je úloha jedním sekvenčním procesem, který má lokální data.*
  - *Úlohy komunikují vyvoláním vstupní procedury jiné úlohy.*
- A má *rendezvous* s těmito úlohami.
  - *Klientská úloha vyvolává vstupní bod který jej může ‘akceptovat’*
- Úloha na straně serveru a způsobí, že mají oba procesy rendevous.
  - *Více akceptovatelných událostí může být na straně serveru očekáváno, pokud je použit příkaz "select";*

ADA - Accept

- **Příkaz accept má tvar**

```
"accept" <entry-name>
[<formal parameter list>]
["do" <statements> "end"];
```

- Příkaz je proveden jen pokud jej vyvolá vzdálený proces

  entry-name (s parametry)

  - *Po provedení příkazu "end", výsledky jsou předány a oba procesy pokračují paralelně.*
  - *Oba procesy, jak volající, tak přijímající, mohou být blokovány, dokud druhý proces není připraven*

ADA – Select, strážci

- **Příkaz Select má následující tvar**

```
– "select"
            ["when" <boolean-expresssion=>]
                  <accept-statement>
            [<statements>]
            {< "or" ["when" <boolean-expression>=>]
                  <accept-statement }
            [<statements>]
            "else" [<statements>]
            "end select;"
```

- 1. Vyhodnotí všechny *boolovské* výrazy, označí všechny “accept” příkazy se splněnou podmínkou jako otevřené *open*. Pokud podmínka není uvedena, pak je *accept* vždy otevřený.
- 2. Zvolte nějaký otevřený "accept" z volaného místa, pokud je otevřených acceptů více, zvolte nějaký nedeterministicky. Pokud žádný accept otevřený není, vykonejte ‘else’ část.
- 3. Pokud tu není žádný "else" a také žádný otevřený "accept„, pak je vyvolána výjimka.

ADA, příklad, omezený buffer

```
task body bounded_buffer is
      buffer: array[0..9] of item;
      in,out: integer;
      count: integer;
      in := 0;
      out := 0;
      count := 0;
      [JÁDRO]
end bounded_buffer;
```

ADA, příklad, omezený buffer [JÁDRO]

```
begin loop
        select
                when count < 10 =>
                        accept insert( it: item )
                                do buffer[in mod 10] := it;
                                        in := in + 1;
                                        count := count - 1;
                                end;
                or when count > 0 =>
                        accept remove( it: out item )
                                do it := buffer[out mod 10];
                                        out := out + 1;
                                        count := count - 1;
                                end;
        end select;
end loop;
```

NÁSTĚNKOVÝ SYSTÉM, LINDA

LINDA - úvod

- Vyvinuta AT&T Bell Labs, Yale University (Ahuja, Carriero, Gelenter, LINDA & FRIENDS, 1986)
- Host programming language + LINDA -> Jazyk pro programování paralelních systémů
- Platformě a aplikačně nezávislé
- Původně vyvinuta C-Linda, Fortran-Linda
- Nyní například v Sicstus PROLOG, Agilla (agenti pro WSN)

LINDA

- Paralelní programování založené na jazyce C
- Jedná se o *koordinační jazyk* pro *asynchronní* a *asociativní* komunikační mechanismus nas síleným *globálním prostorem* nazývaným prostor n-tic *Tuples Space* (TS)
- Co je prostor n-tic?
  - *Virtuální sdílená paměť*
- Co je n-tice?
  - *Základní datová struktura nástěnky*
  - *Sekvence aktuálních a formálních položek*
  - *Příklad:*

```
('arraydata', dim1, 13, 2)
```

Nástěnka / prostor n-tic (TUPLESPACE)

Generování n-tic

- **out**
  - *Generuje data (pasivní) n-tice*
  - *Každá položka je vyhodnocena a umístěna na nástěnku*
  - *Řízení je potom navráceno volajícímu procesu*
  - *Příklad:* `out ('array data', dim1, dim2);`
- **eval**
  - *Generuje procesy – aktivní n-tice*
  - *Řízení je navráceno volajícímu procesu ihned po aktivaci nového procesu n-ticí*
  - *Každá položka je teoreticky vyhodnocena souběžně s ostatními a výsledek vložen do n-tice umístěné na nástěnce*
  - *Položky obsahující funkci (nebo podproceduru) způsobí vznik nového procesu, který ji vykoná*
  - Příklad: `eval ("test", f(i));`

Čtení a odstraňování n-tic

- in
  - *Používá šablony pro získání n-tic z nástěnky.*
  - *Pokud je n-tice získána, je odstraněna z nástěnky a dále pak již není zde pro další přístupy.*
  - *Pokud neexistuje n-tice, která by šabloně vyhovovala, je proces pozastaven. Takto lze dosáhnout synchronizace mezi procesy.*
  - *Příklad:* `in("arraydata", ?dim1, ?dim2);`
- rd
  - *Používá šablonu pro získání dat bez jejich odstraňování z nástěnky.*
  - *Po přečtení je n-tice nadále dostupná ostatním procesům.*
  - *Pokud neexistuje n-tice, která by šabloně vyhovovala, je proces pozastaven.*
  - *Příklad:* `rd ("arraydata", ?dim1, ?dim2);`

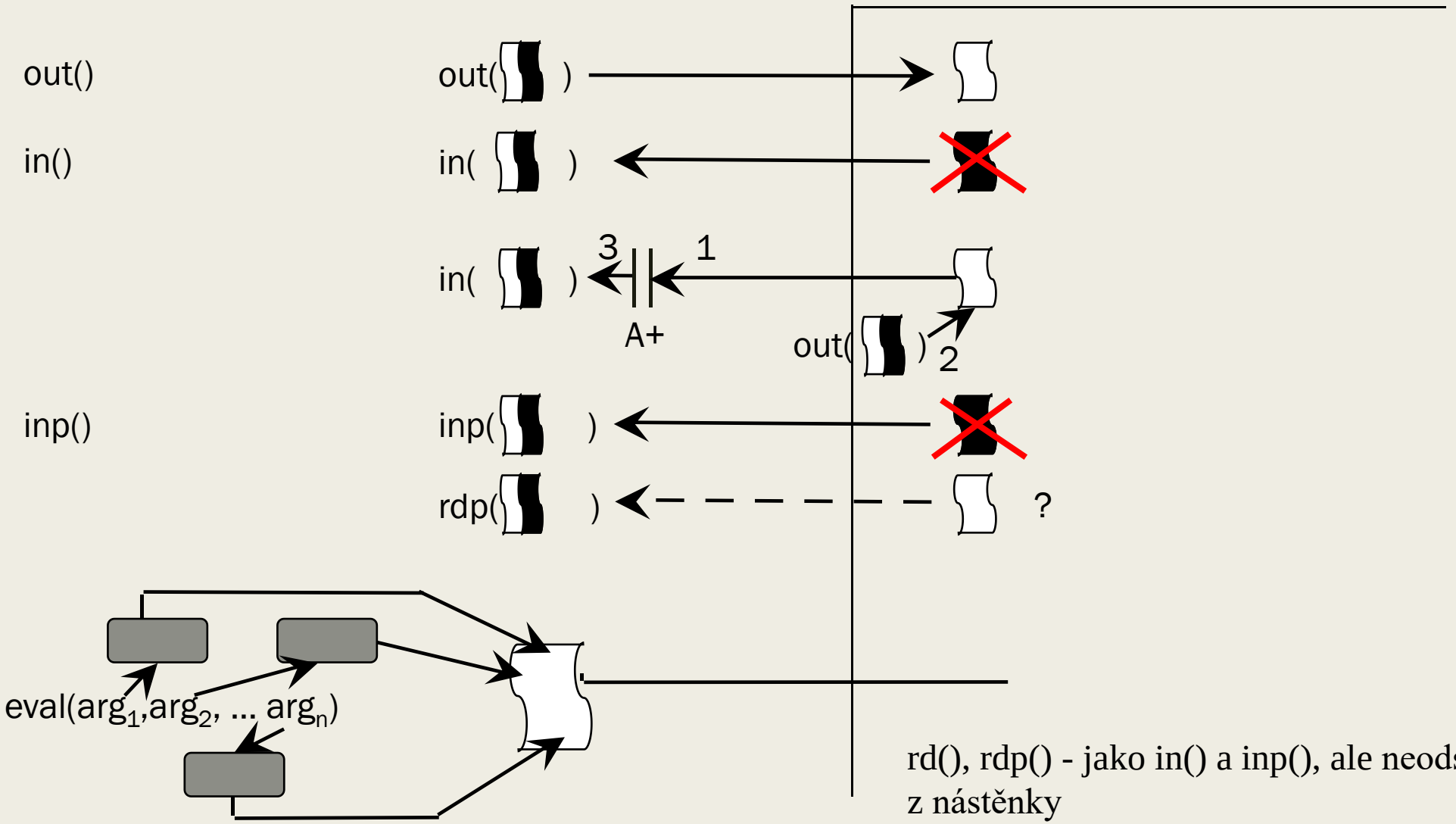
Porovnání n-tice se šablonou

- Musí odpovídat počet položek v šabloně a hledané n-tici
  - *Také musí mýt odpovídající typ, délku, hodnoty*
- Na odpovídající položce musí
  - *Musí odpovídat typ a délka prvků v šabloně s vyhledanou n-ticí*
  - *Pozor – pořadí vyhodnocování položek není definování, proto něčemu jako …*

```
out ("string", i++, i); // bychom se měli vyhnout
```

  - *Pokud je možné šabloně přiřadit více n-tic, není definováno, jak vybrat jednu konkrétní.*

Operátory jazyka LINDA



rd(), rdp() - jako in() a inp(), ale neodstraní tuple z nástěnky

Standardní synchronizační primitiva v LINDA

- *Semafor*
  - P (Sem1) in(Sem1)
  - V (Sem1) out(Sem1)
- *Sdílená paměť*
  - Reading — rd(Variable, ?value)
  - Writing (atomic) — in(Variable, ?value) out(Variable, NewValue)
- *Asynchronní kanál*
  - Send(Chan1, Val) — out(Chan1, val)
  - Receive(Chan1, Val) — in(Chan1, ?val)
  - Ready_to_Receive(Chan1) — rdp(Chan1, ?Dummy)
- *Synchronní kanál*
  - Send(Chan1, Val) — out(Chan1, value, Val) in (Chan1, ?got)
  - Receive(Chan1, Val) — in(Chan1, value, ?Val) out(Chan1, got)

Příklad – pět filosofů

```
phil (i) {
        while (1) {
        think ();
        in(“chopstick”, i);
        in(“chopstick”, (i+1) % Num);
        eat();
        out (“chopstick”, i);
        out (“chopstick”, (i+1) % Num);
        }
}
```

Lístkový algoritmus

```
int incr (name) {
      in (name, ?n);
      out (name, n++);
      return n;
}


{
      ticket = incr (“tick”);
      rd (“next”, ?ticket);
      ...
      incr (“next”);
}
```

Vyhledání prvočísel

```
is_prime (me) {
        limit = sqrt (me) + 1;
        for (i=2; i<limit; i++) {
        rd (“primes”, i, ?ok);
        if (ok && (me % i==0)) return 0;
}
return 1; }
main () {
        for (i=2; i<=LIMIT; i++) eval (“primes”, i, is_prime(i));
        for (i=2; i<=LIMIT; i++) {
        rd (“primes”, i, ?ok);
        if (ok) count++;
}
print (“%d. \n”, count); }
```

Distribuované struktury

- *Seznam* *(Name, Unique_identifier, Next, Val)*
  - *Průchod seznamem*

```
typedef ... TID;
TID id, next;
rd(„list“, „head“, ?id)
while (id != ID_NIL) {
      rd(„list“, id, ?next, ?val);
      id = next;
}
```

- Vytvoření seznamu

```
TID id, next;
next = ID_NIL;
while (val=getval()) {
      id=GetUniqueId();
      out(„list“, id, next, val);
      next = id;
}
out(„list“, „head“, next)
```

Distribuované struktury

- ***Bag (batoh)***
  - *Nerozlišitelné prvku*
  - *Příklad - master-workers (farm) konstukce používá „bag of tasks“:*
  - *Master Worker*

```
out(„task“, Descript) in(„task“, ?New_Task)
```

- ***Sdílená proměnná***
  - *in(„var1“, ?val)*
  - *out(„var1“, NewVal)*
- ***Pole***
  - *(Jméno, Index/Indicie, Hodnoty)*
  - *Průchod polem*

```
for(i=0; i< MAX, i++)
rd(„Array“, i, ?val);
```

LINDA v SICSTUS PROLOGu

Klient – server přístup

```
| ?- use_module(library('linda/server')).
```

```
| ?- linda.
Server address is msi:’61610’
```

```
| ?- use_module(library('linda/client')).
```

```
| ?- linda_client(msi:’61610’).
```

```
| ?- linda_client(msi:’61610’).
```

```
| ?- linda_client(msi:’61610’).
```

LINDA v SICSTUS PROLOGu

- Blokující operace

```
rd(?Tuple), in(?Tuple)
rd(?TupleList, ?Tuple), in_noblock(?TupleList, ?Tuple)
```

- Neblokující operace

```
rd_noblock(?Tuple), in_noblock(?Tuple)
```

- Bag (viz PROLOG) `rd_bag_noblock`

Na nástěnce: `(jmeno muz, franta), (jmeno, zena, pavla),`

```
   (jmeno, zena, hana), (jmeno, zena, tereza),
|?- rd([(jmeno,stroj,X), (jmeno, zena,Y)], JMENO).
Y = pavla
JMENO = (jmeno,zena,pavla) ?
|?- bagof_rd_noblock(jmena(X,Y),(jmeno,X,Y),BAG).
BAG = [jmena(muz,franta), jmena(zena,pavla), jmena(zena,hana), jmena(zena,tereza)] ?
```

LINDA v SICSTUS PROLOGu


```
writepni(L,L).
writepni(A,L):-rd_noblock((prvocislo,A)),write((prvocislo,A)),nl,L2 is
A+1, writepni(L2,L).
writepni(A,L):-L2 is A+1, writepni(L2,L).
writepn(L):-writepni(2,L).

deletepn:-rd_noblock((prvocislo,A)),in((prvocislo,A)),deletepn.
deletepn.

testpni(A,B):- A < B*B, out((prvocislo,A)).
testpni(A,B):- 0=:= (A mod B).
testpni(A,B):- C is B+1, testpni(A,C).

pn(A,A).
pn(A,B):-testpni(A,2),C is A+1, pn(C,B).

do(L):-rd((pnstart,A)),D is A+L, pn(A,D).
```