LME Problem

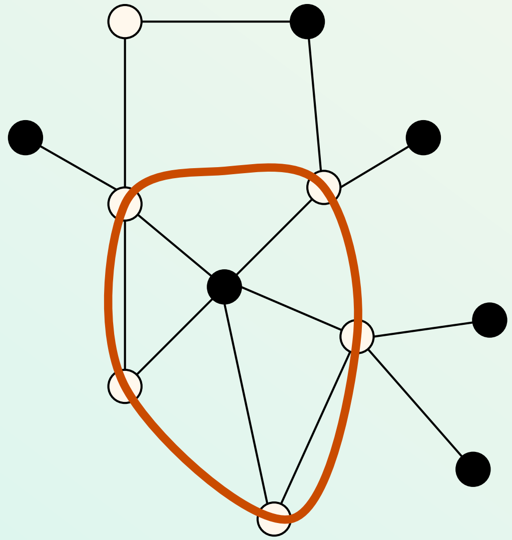
- Safety

  If a process is executing its critical section, then none of its neighbors is executing its critical section simultaneously.

- Liveness

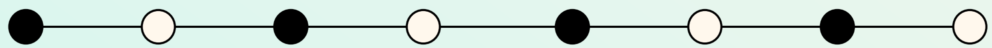
  If a process requests to enter its critical section, then it will eventually execute its critical section.

LME Algorithm

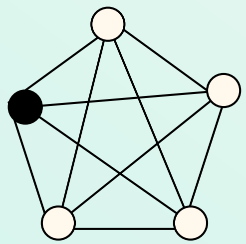


Concurrency

Best (expected) case



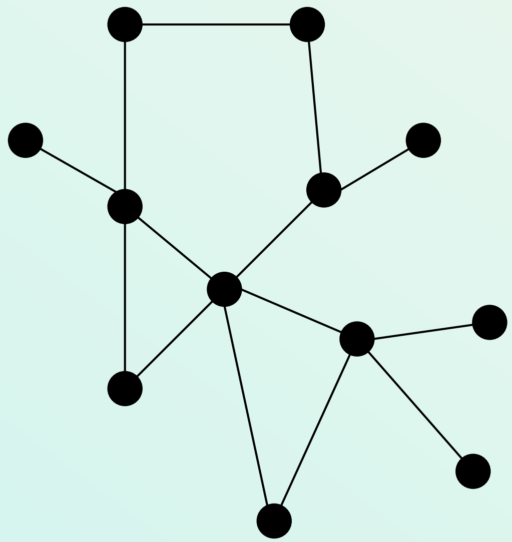
Worst case



i.e., Mutual Exclusion

Question

Could we increase concurrency?



Answer

Yes!

Local Resource Allocation !

Local Resource Allocation

- Statement

  Neighboring Processes share *m* resources with compatibility criteria

- Compatibility

  Two resources X and Y are said to be compatible, denoted $X \leftrightarrow Y$,

  if two neighboring processes can access X and Y concurrently

Local Resource Allocation

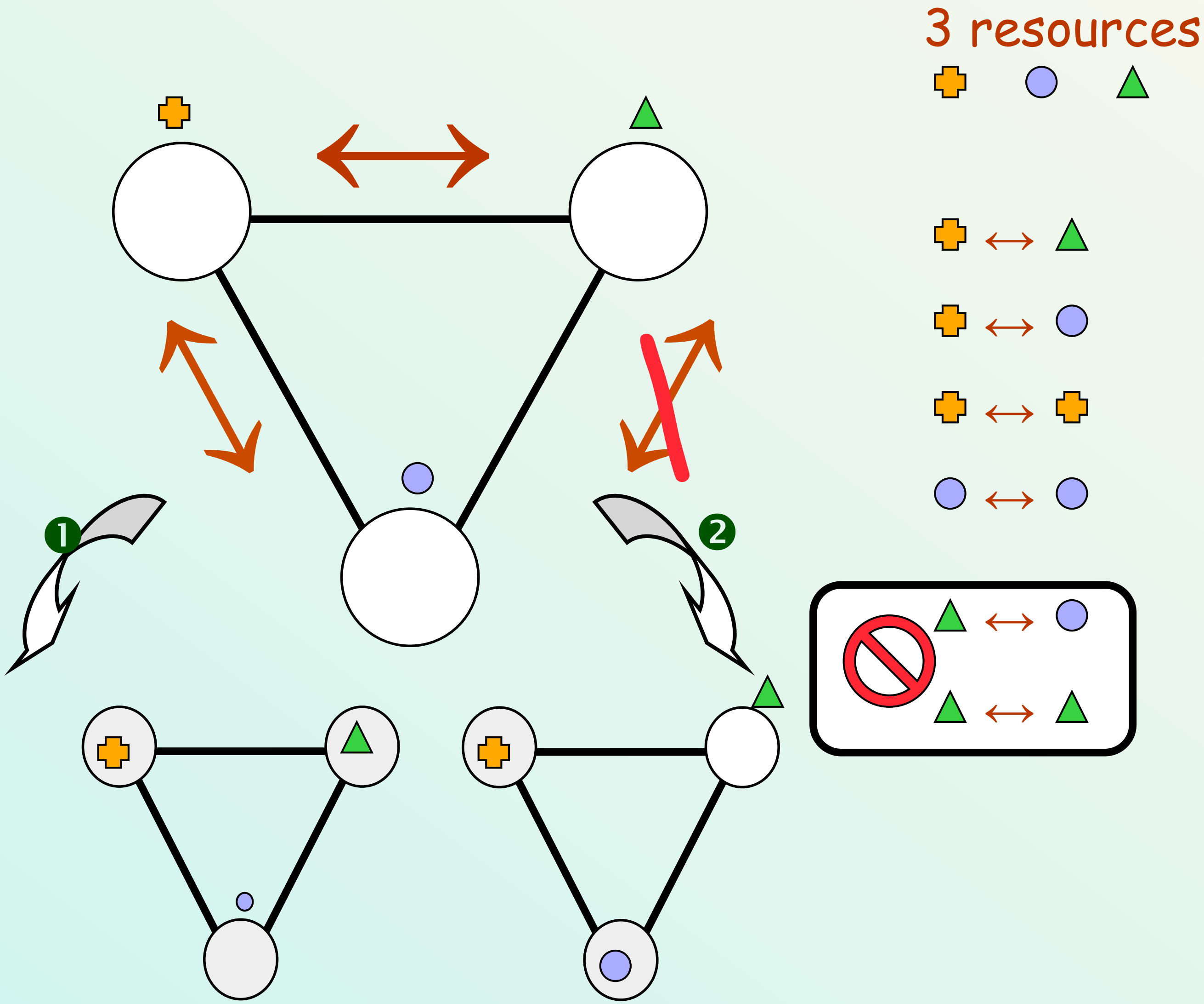
- Safety

  If two neighboring processes *p* and *q* (p<>q) are executing their critical section concurrently using X and Y, respectively, then $X \leftrightarrow Y$.

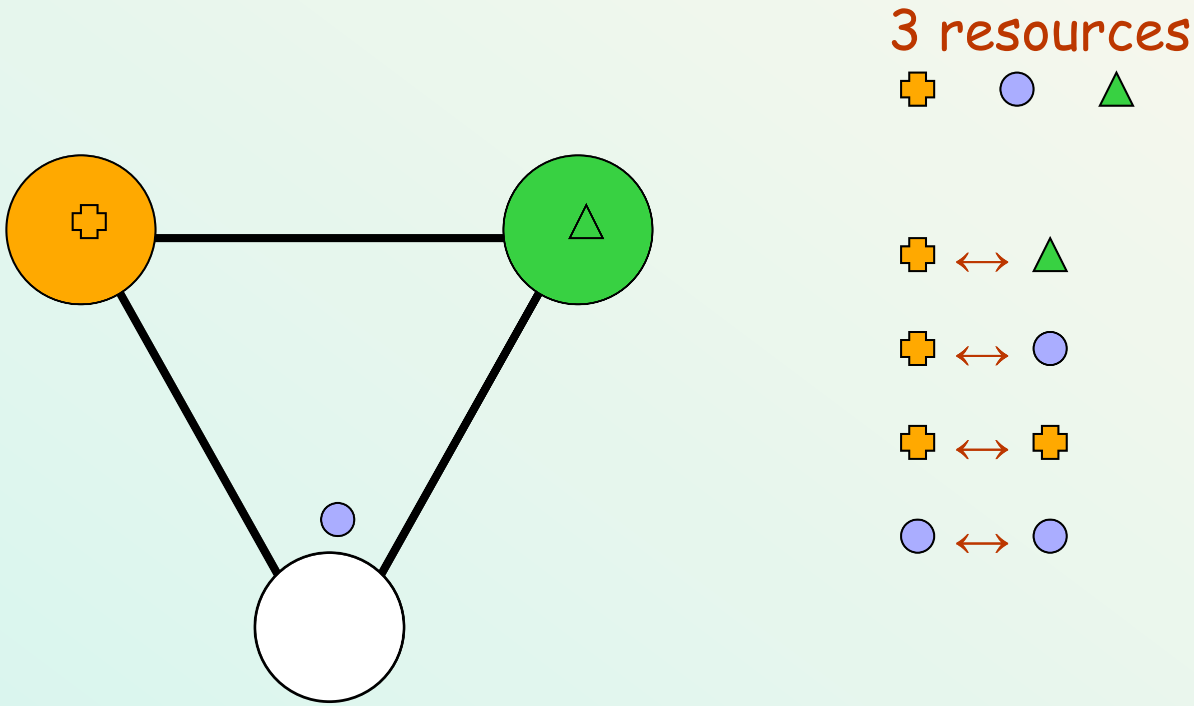
- Liveness

  If a process requests to enter its critical section, then it will eventually execute its critical section.

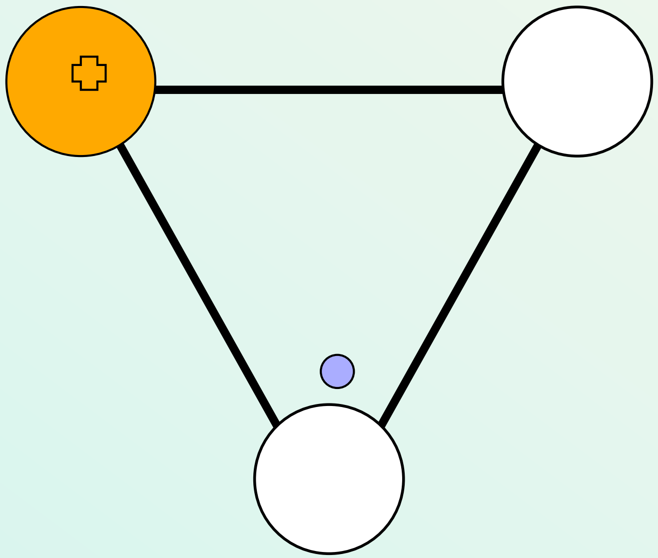
Local Resource Allocation



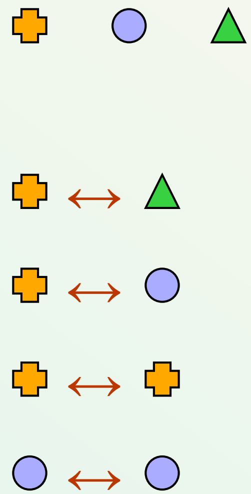
Local Resource Allocation



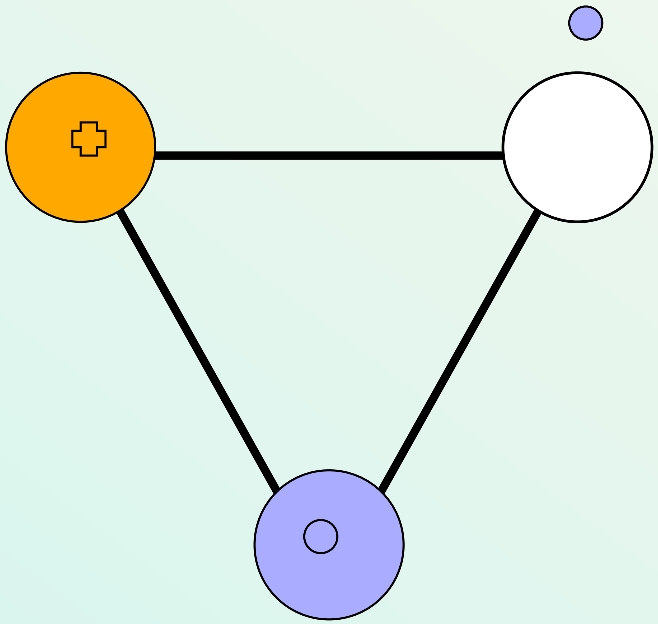
Local Resource Allocation



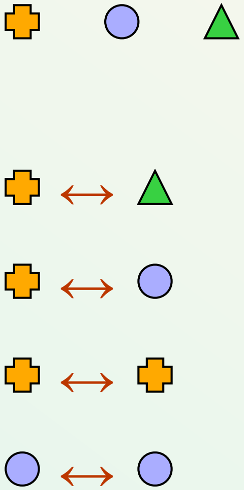
3 resources



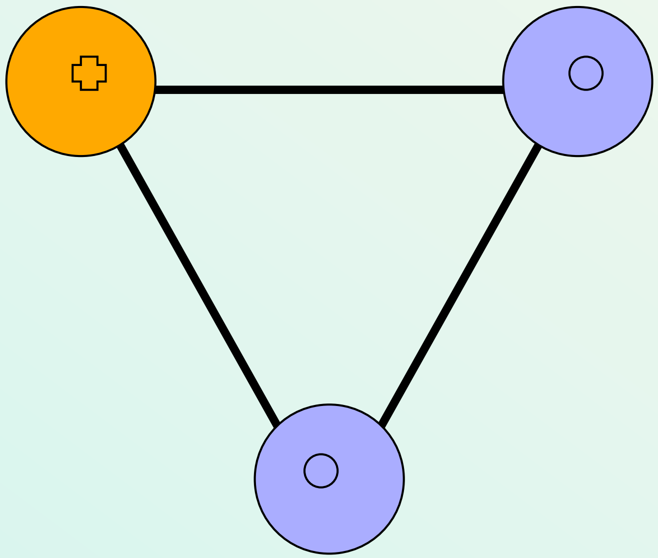
Local Resource Allocation



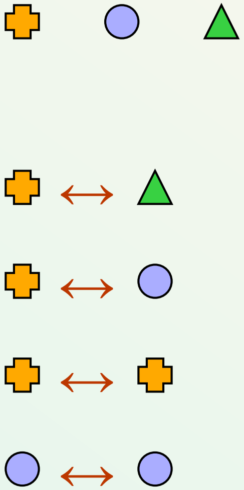
3 resources



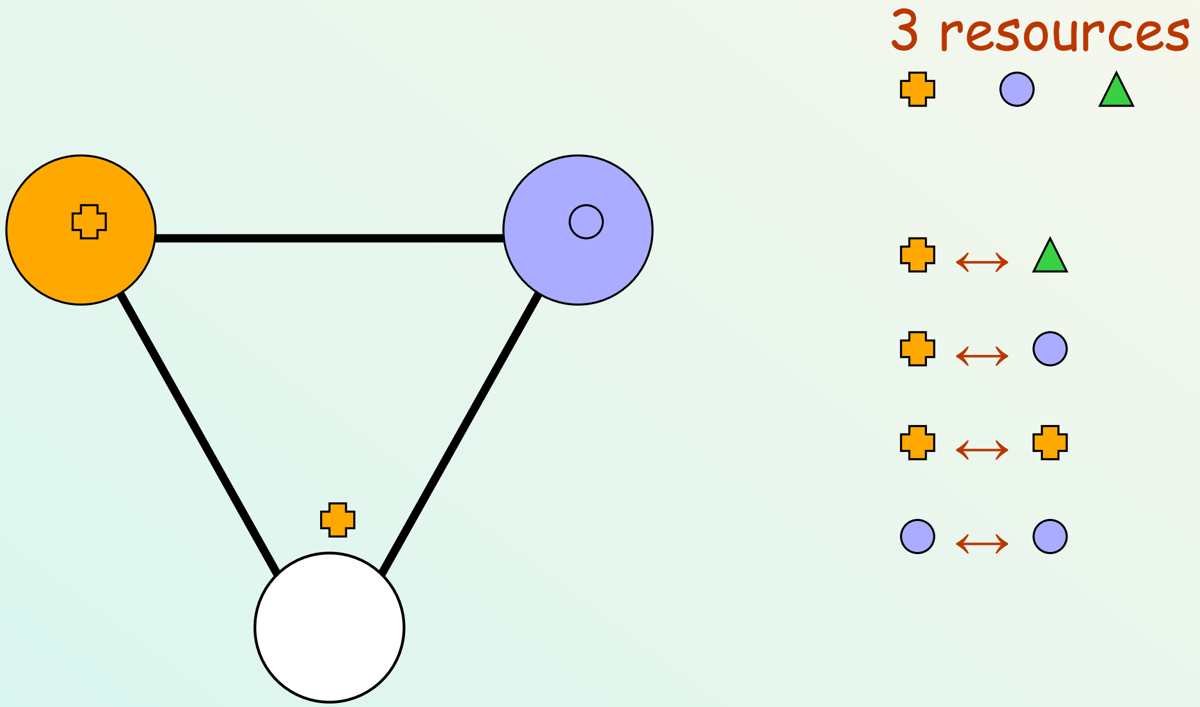
Local Resource Allocation



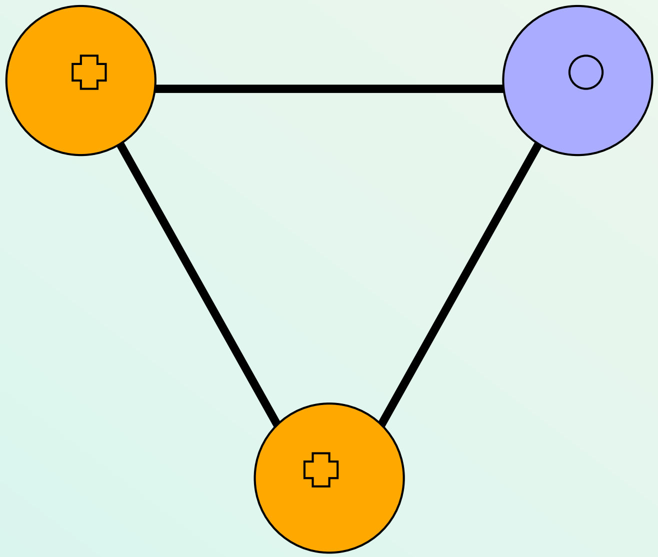
3 resources



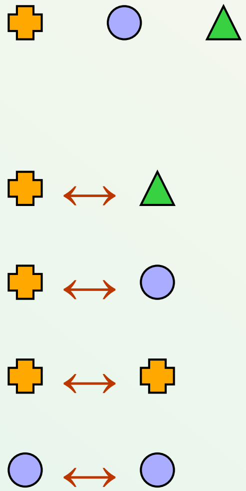
Local Resource Allocation



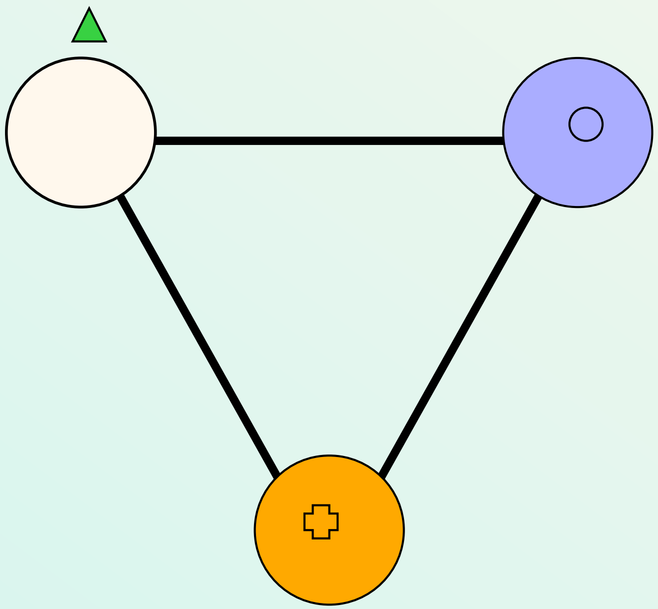
Local Resource Allocation



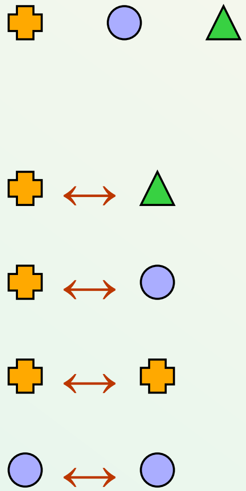
3 resources



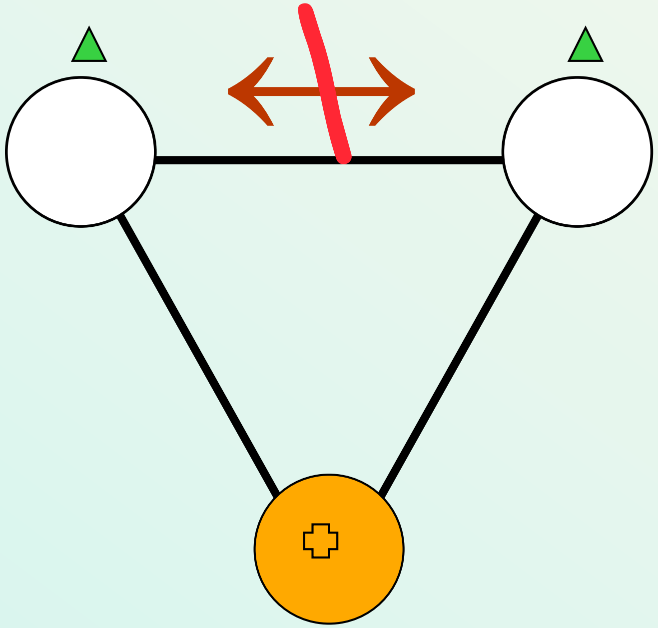
Local Resource Allocation



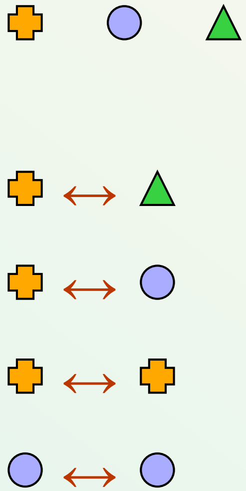
3 resources



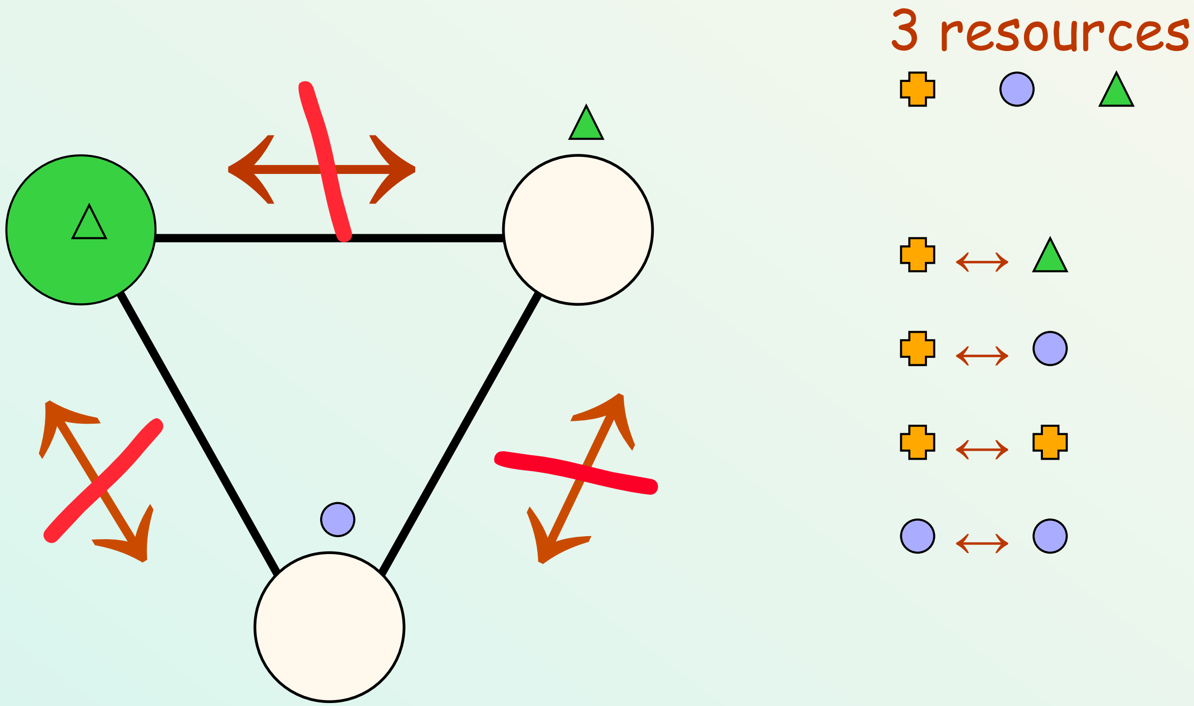
Local Resource Allocation



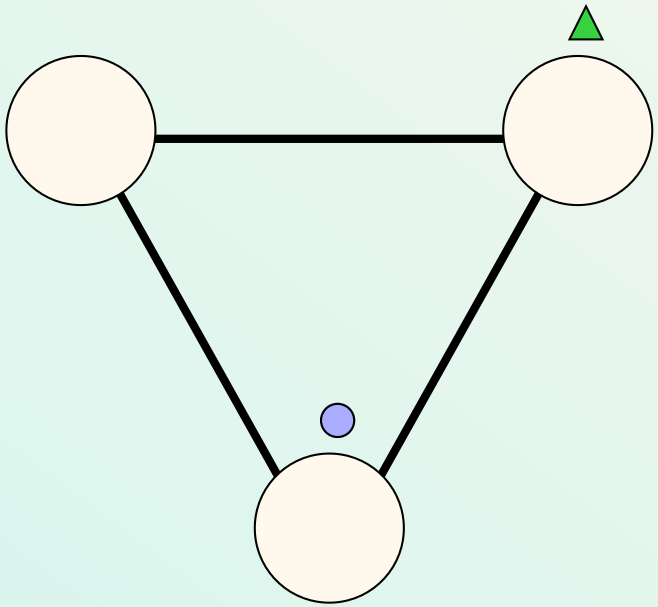
3 resources



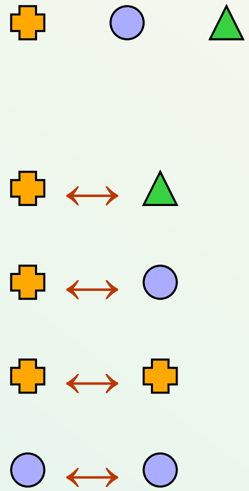
Local Resource Allocation



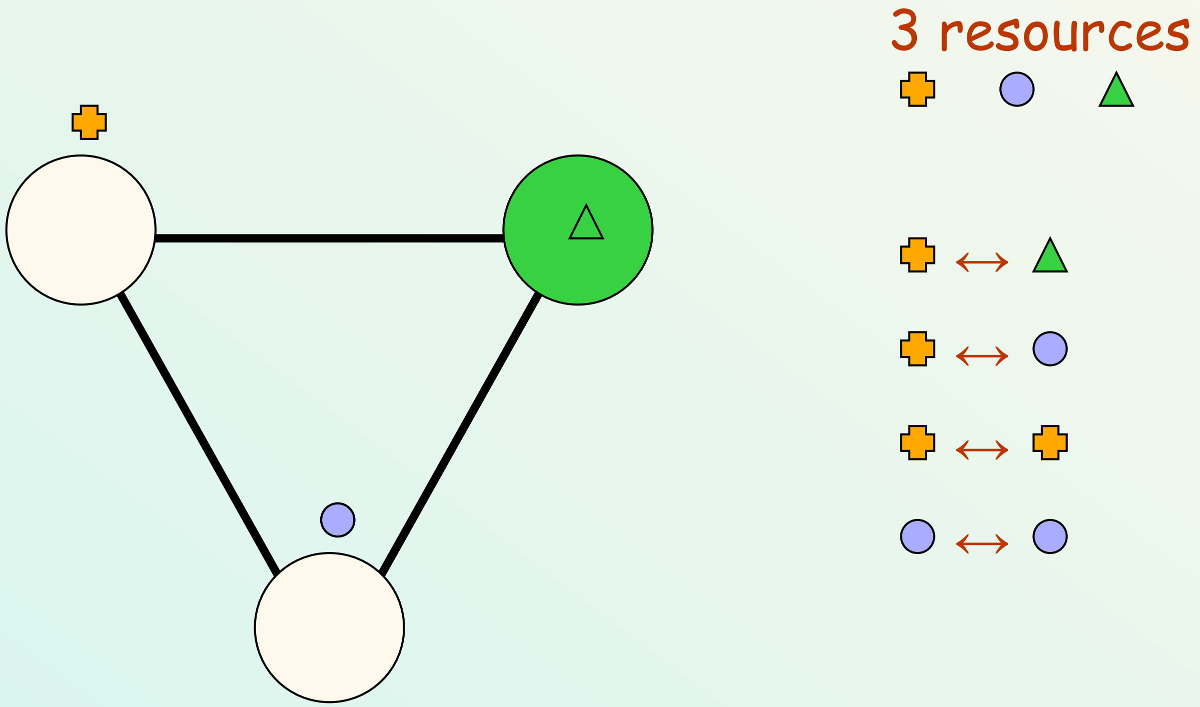
Local Resource Allocation



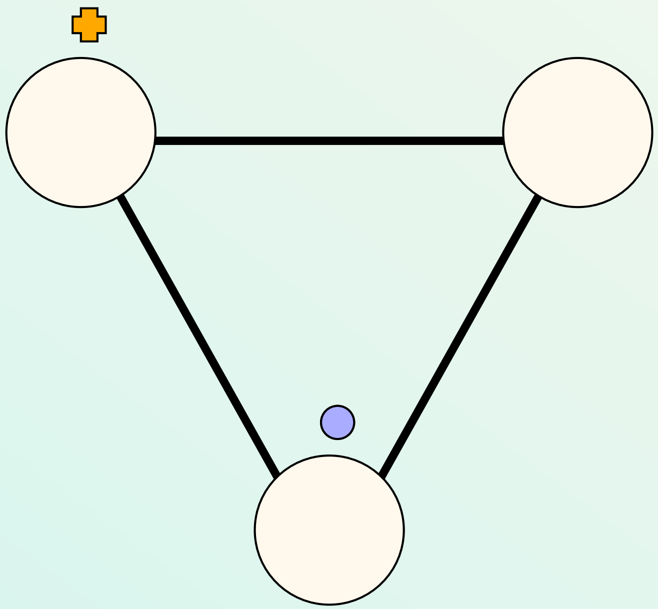
3 resources



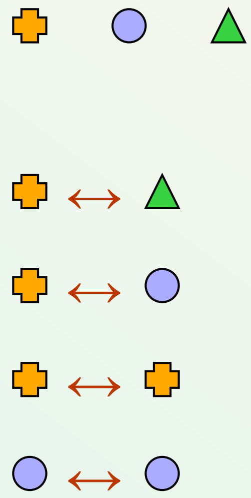
Local Resource Allocation



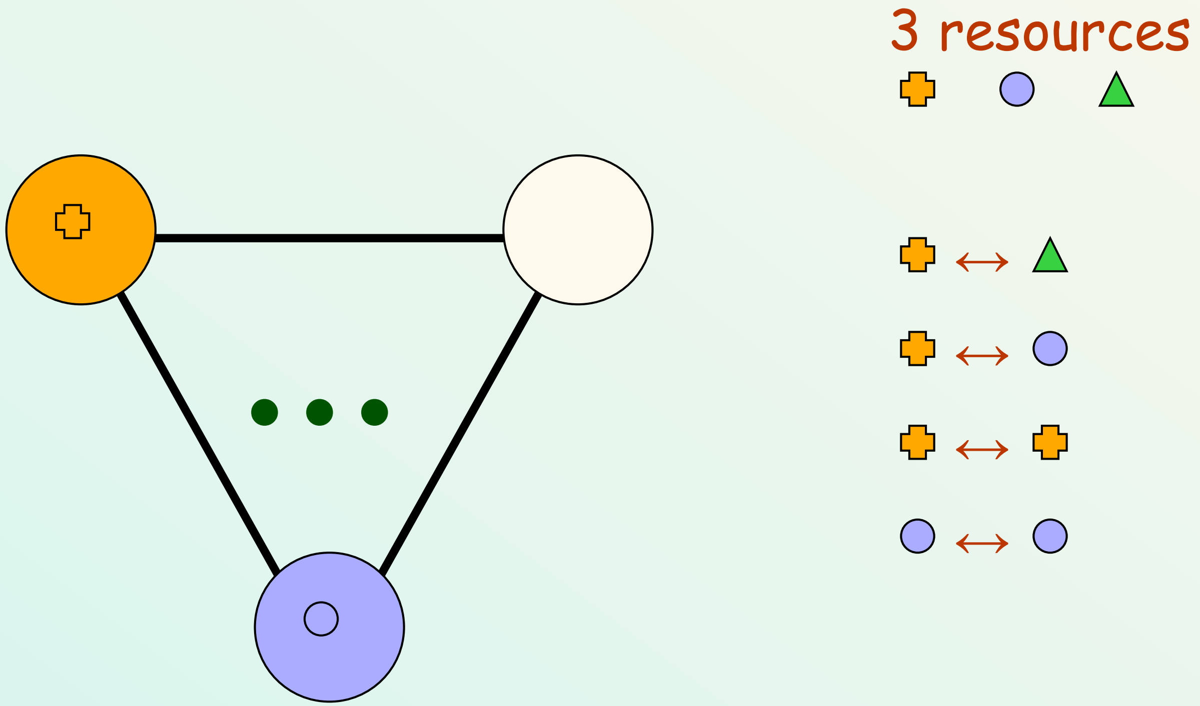
Local Resource Allocation



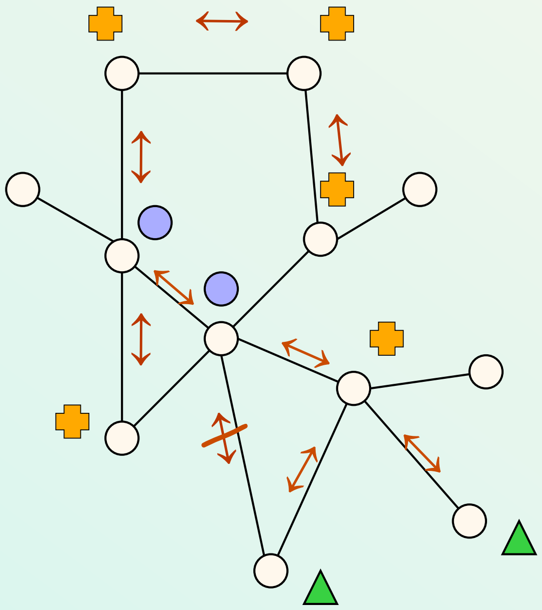
3 resources



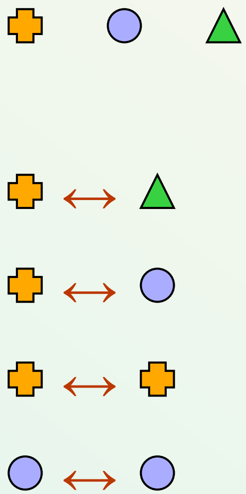
Local Resource Allocation



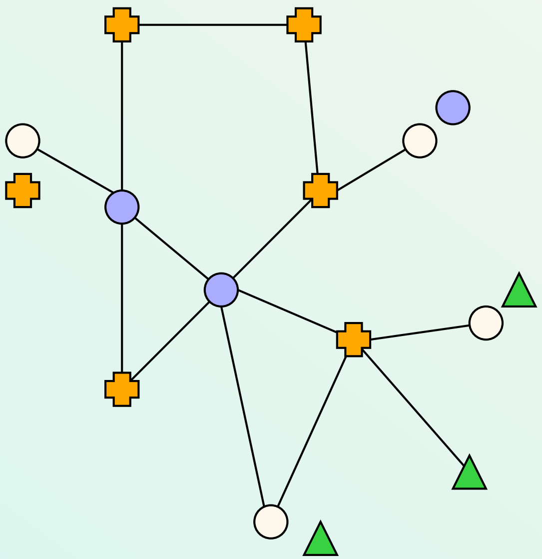
Local Resource Allocation



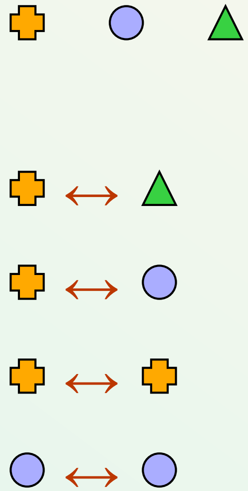
3 resources



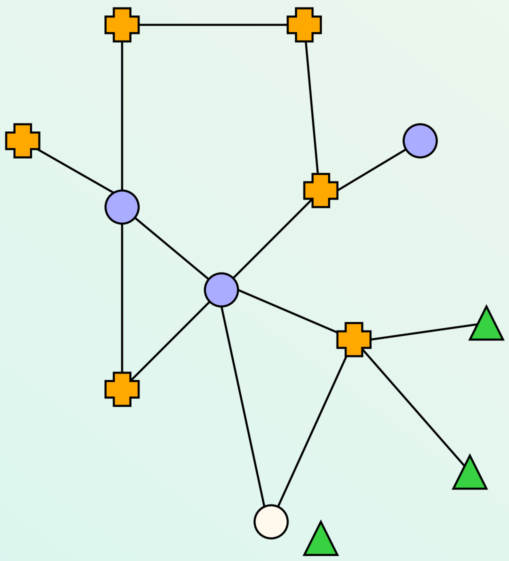
Local Resource Allocation



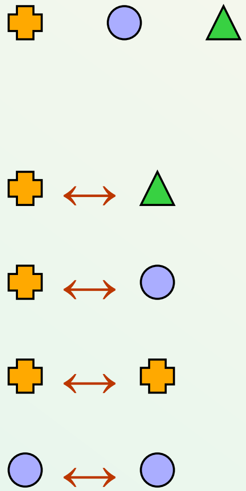
3 resources



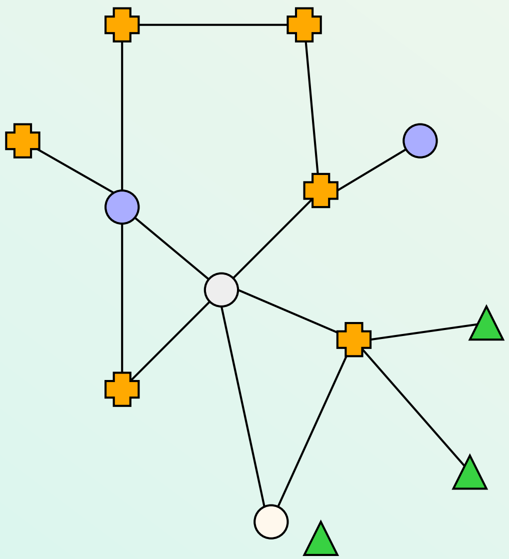
Local Resource Allocation



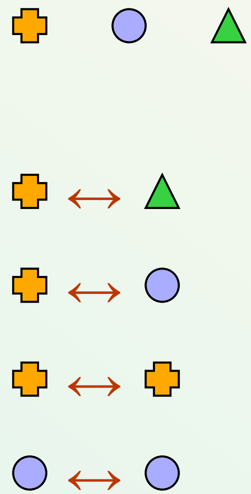
3 resources



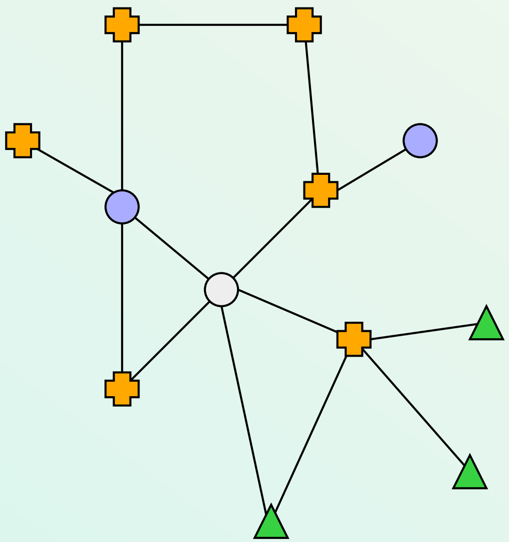
Local Resource Allocation



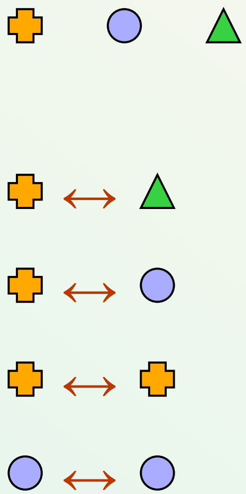
3 resources



Local Resource Allocation



3 resources



Relation With Other Resource Allocation Problems?

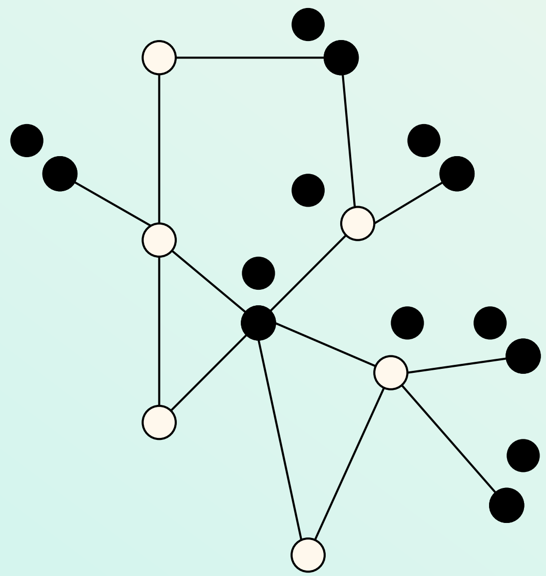
Related Resource Sharing Problem

- Dinning Philosophers [Dijkstra, EWD 625, 1978] (Local Mutual Exclusion)

  One Resource Shared in Exclusive Access by Neighbor Processes

  Concurrency = 1 process in the neighborhood

LRA solves LME



1 resource only

Related Resource Sharing Problem

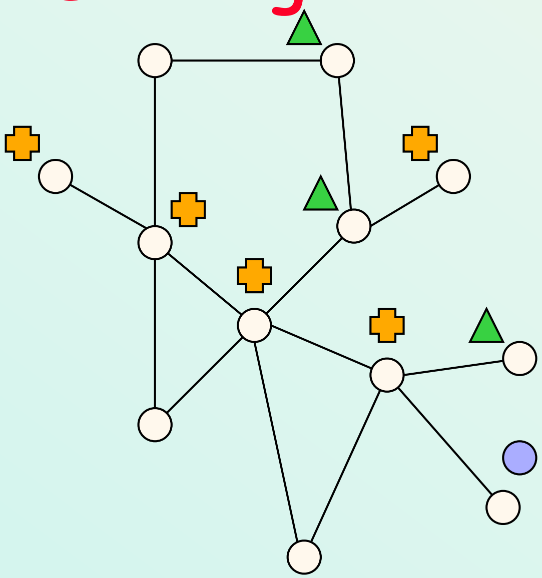
- Drinking Philosophers [Chandy and Misra,TOPLAS 1984]

  Several Resources Shared in Exclusive Access by Neighbor Processes

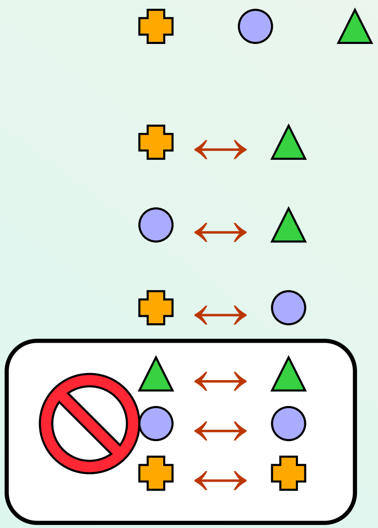
  Concurrency = 1 process in the neighborhood for each resource

  Neighbors are allowed to use resources simultaneously provided that they are using different resources

LRA solves Drinking Philosophers Problem



3 resources



Related Resource Sharing Problem

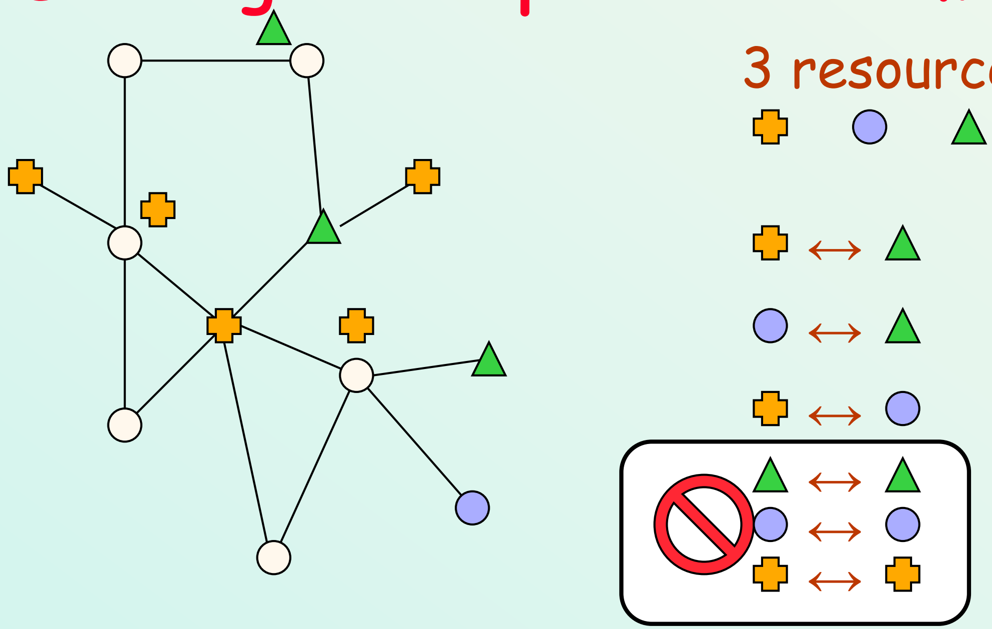
- Drinking Philosophers [Chandy and Misra, TOPLAS 1984]

  Several Resources Shared in Exclusive Access by Neighbor Processes

  Concurrency = 1 process in the neighborhood for each resource

  Neighbors are allowed to use resources simultaneously provided that they are using different resources

LRA solves Drinking Philosophers Problem



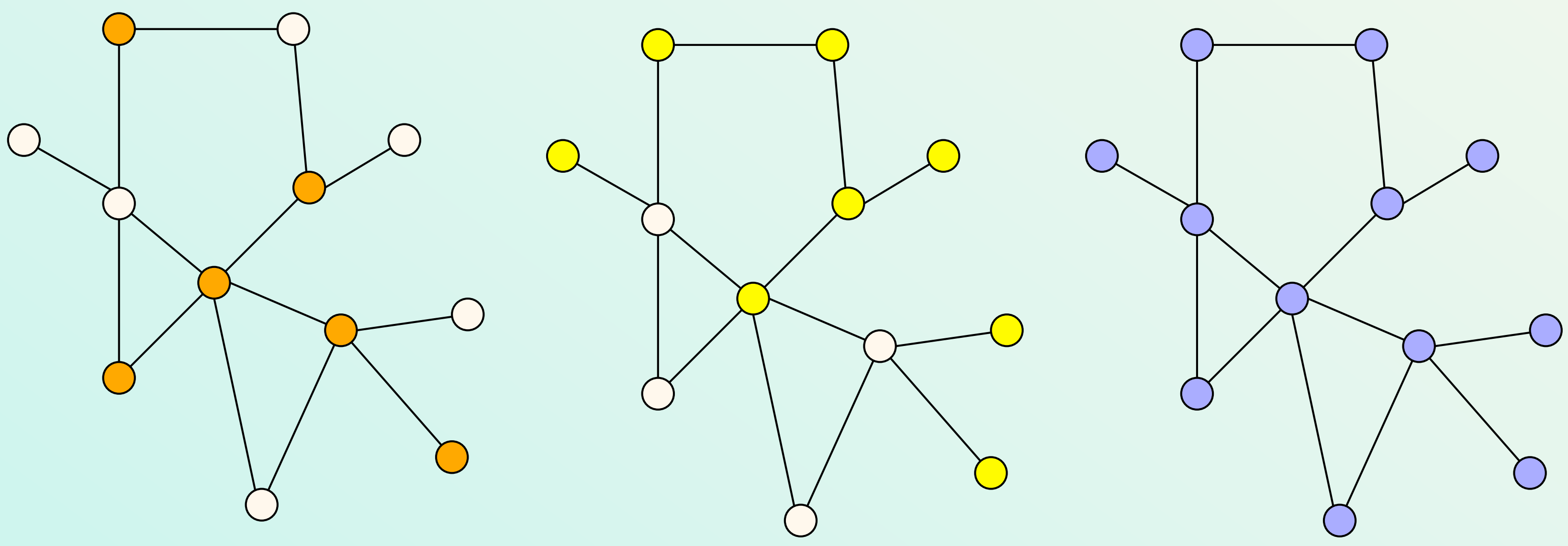
Related Resource Sharing Problem

- Group Mutual Exclusion [Joung, DISC 1998]

  M Exclusive Resources Shared in Concurrent Access

  Concurrency = n processes

  ... but exclusive resource access



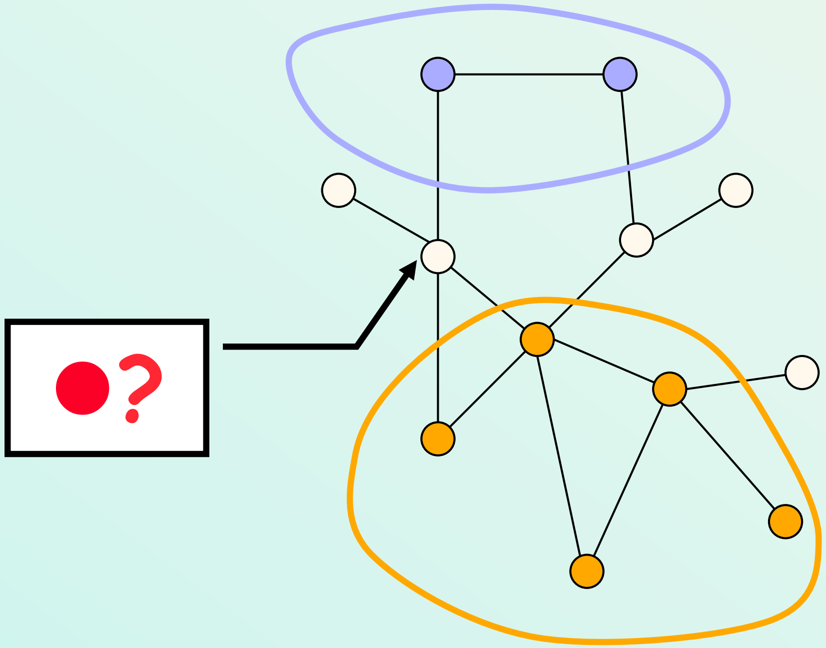
Related Resource Sharing Problem

- Local Group Mutual Exclusion

  M Local Exclusive Resources Shared in Concurrent Access

  Concurrency = Δ+1 processes in the neighborhood

  ... but exclusive resource access



Related Resource Sharing Problem

- Local Group Mutual Exclusion

  M Local Exclusive Resources Shared in Concurrent Access

  Concurrency = Δ+1 processes in the neighborhood

  ... but exclusive resource access

Does LRA solves the (Local) GME Problem?

Yes !

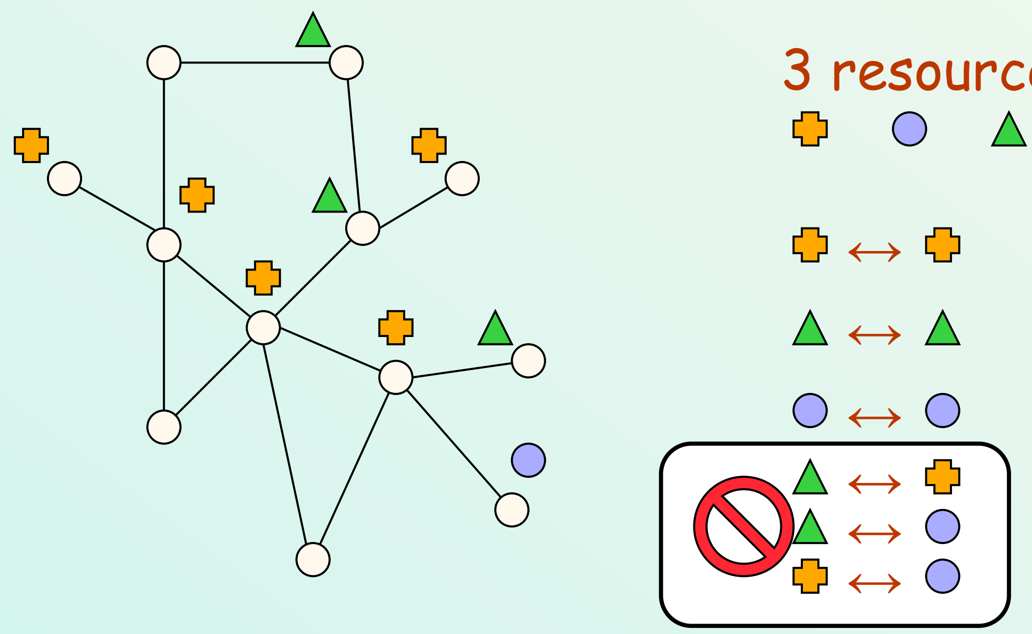
Related Resource Sharing Problem

- Local Group Mutual Exclusion

  M Local Exclusive Resources Shared in Concurrent Access

  Concurrency = Δ+1 processes in the neighborhood

  ... but exclusive resource access



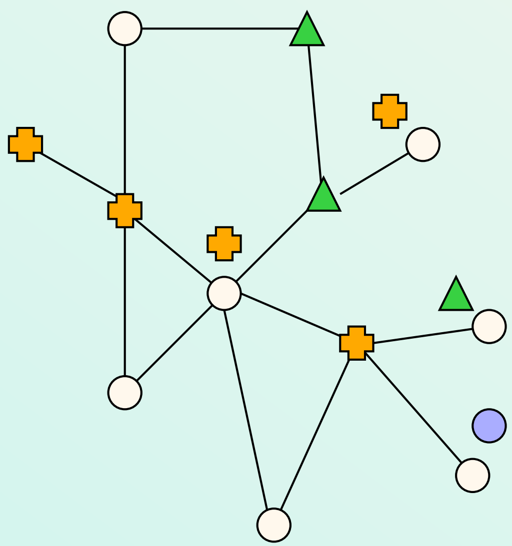
Related Resource Sharing Problem

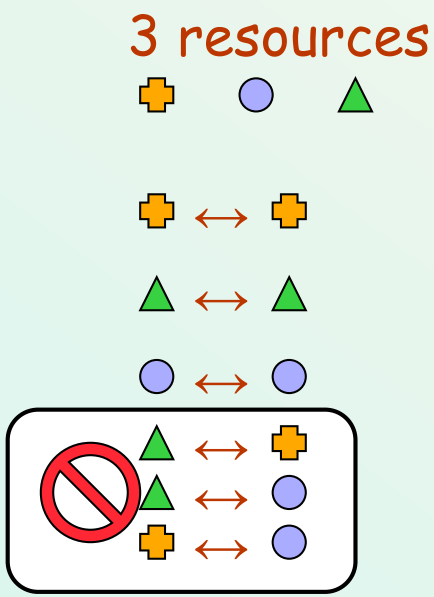
- Local Group Mutual Exclusion

  M Local Exclusive Resources Shared in Concurrent Access

  Concurrency = Δ+1 processes in the neighborhood

  ... but exclusive resource access





Related Resource Sharing Problem

LRA is a generalisation of

- Local Mutual Exclusion
- Dining and Drinking Philosophers
- Local Group Mutual Exclusion
  (and then Local Readers/Writers)

An SS LRA Algorithm

Each process p maintains

- A compatibility graph $CG = (R, C)$

  R = Resources accessed by p

  C = Compatibility Edges

- *Request* $\in R \cup \{\perp\}$ (upper layer)
- ***Grant***: Boolean (LRA layer)
- ***c***: integer, a Lamport's timestamp (LRA layer)

An SS LRA Algorithm

Entry Section

- Each time a process *p* requests to access the critical section with Resource *X* (Upper Layer):
  - p sets Request to X (Upper Layer)
  - p updates its timestamp c (LRA Layer)
- Then, *p* waits for one of the two following conditions to become true (Grant to true by the LRA Layer):
  - *p* has the maximum timestamp in its neighborhood, or
  - Request (X) is compatible with its neighbors

Exit Section

- *p* resets Request to $\perp$, Grant to false, and c to 0 (LRA Layer)

The Upper Layer Algorithm

$\vdots$

REQ : $(Request = \perp) \wedge \neg Grant \wedge (X \in \mathcal{R}\ requested) \rightarrow Request := X$

CS : $(Request \neq \perp) \wedge Grant \rightarrow$ <<CS>>; $Request := \perp$

$\vdots$

No request $\Rightarrow$ No action $\Rightarrow$ Silent Algorithm

Self-stabilizing LRA using Unison

$p \prec q \equiv (c < c_q) \vee ((c = c_q) \wedge (p < q))$

$\text{Ready} \equiv \forall\, q \in N\colon (c_q > 0) \Rightarrow ((p \prec q) \vee ((Request_q \neq \perp) \Rightarrow (Request \leftrightarrow Request_q)))$

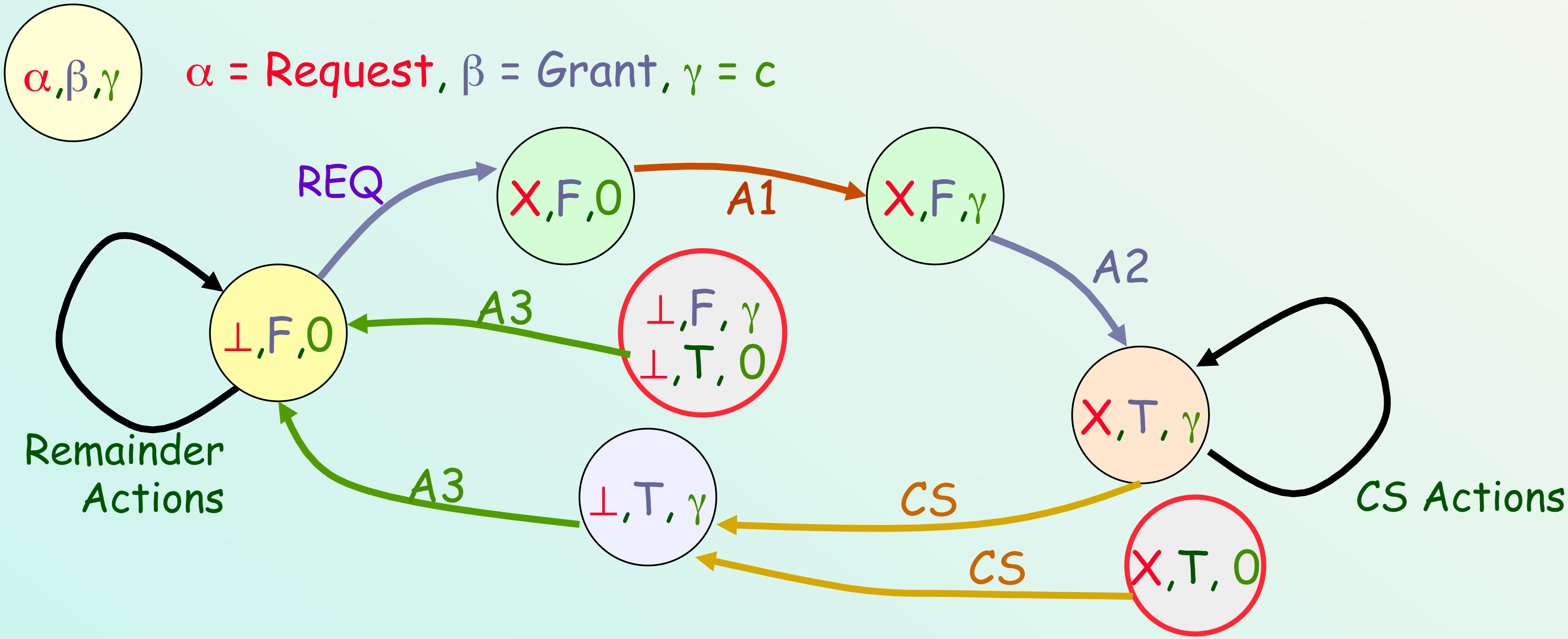
A1 : $(\text{Request} \neq \perp) \wedge \neg\text{Grant} \wedge (c = 0) \rightarrow c := \max\{c_q \in N\} + 1$

A2 : $(\text{Request} \neq \perp) \wedge \neg\text{Grant} \wedge (c > 0) \wedge \text{Ready} \rightarrow \text{Grant} := \text{true}$

A3 : $(\text{Request} = \perp) \wedge (\text{Grant} \vee (c > 0)) \rightarrow \text{Grant} := \text{false};\ c := 0$

Self-Stabilizing LRA Scheme

REQ :	$(Request = \perp) \wedge \neg Grant \wedge (X \in R\ requested)$	$\rightarrow Request := X$
CS :	$(Request \neq \perp) \wedge Grant \rightarrow \langle\langle CS \rangle\rangle;\ Request := \perp$	
A1 :	$(Request \neq \perp) \wedge \neg Grant \wedge (c = 0) \rightarrow c := \max\{c_q \in N\} + 1$	
A2 :	$(Request \neq \perp) \wedge \neg Grant \wedge (c > 0) \wedge Ready \rightarrow Grant := true$	
A3 :	$(Request = \perp) \wedge (Grant \vee (c > 0)) \rightarrow Grant := false;\ c := 0$	



Lamport's Clock

Drawback

- *c*: infinite integer

Is there any possibility to use a finite clock ?

Yes !

UNISON

The LRA Layer Algorithm

$C \in \{0..2n\}$

$p \prec q \equiv (c < c_q) \vee ((c = c_q) \wedge (p < q))$

DoNotStop $\equiv (\exists\ q \in N: (c_q \neq c_p)$

Ready $\equiv$ DoNotStop $\Rightarrow ((p \prec q) \vee ((Request_q \neq \perp) \Rightarrow (Request \leftrightarrow Request_q)))$

A1 : (DoNotStop $\vee$ Request $\neq \perp$) $\wedge$ $\neg$Grant $\rightarrow$ c := $\varphi(r_p)$

A2 : (Request $\neq \perp$) $\wedge$ $\neg$Grant $\wedge$ Ready $\rightarrow$ Grant := true

A3 : (Request = $\perp$) $\wedge$ (Grant $\vee$ (c > 0)) $\rightarrow$ Grant := false

References

- Sébastien Catarel, Ajoy K. Datta, Franck Petit. *Self-Stabilizing Atomicity Refinement Allowing Neighborhood Concurrency.* SSS 2003: 102-112

- Christian Boulinier, Franck Petit, Vincent Villain. *When graph theory helps self-stabilization.* PODC 2004: 150-159