Clocks and Snapshots

Michael Whittaker

September 30, 2015

http://bit.ly/1PCy5ye

Mathematical Preliminaries: Relations

A *relation* R on sets A and B is a subset of $A \times B$. Alternatively, a relation R on a set A is a subset of $A \times A$.

$$R \subseteq A \times B, \qquad R \subseteq A \times A$$

Mathematical Preliminaries: Relations

A *relation* R on sets A and B is a subset of $A \times B$. Alternatively, a relation R on a set A is a subset of $A \times A$.

$$R \subseteq A \times B, \qquad R \subseteq A \times A$$

For example, the following sets are relations on $A = \{1, 2\}$ and $B = \{x, y, z\}$.

- $\{(1, x), (2, x)\}$
- $\{(1, x), (1, y), (2, z)\}$
- $A \times B$
- $\{\}$

Mathematical Preliminaries: Relations

A *relation* R on sets A and B is a subset of $A \times B$. Alternatively, a relation R on a set A is a subset of $A \times A$.

$$R \subseteq A \times B, \qquad R \subseteq A \times A$$

For example, the following sets are relations on $A = \{1, 2\}$ and $B = \{x, y, z\}$.

- $\{(1, x), (2, x)\}$
- $\{(1, x), (1, y), (2, z)\}$
- $A \times B$
- $\{\}$

We denote $(a, b) \in R$ as aRb. For example $1 = 1$ denotes $(1, 1) \in =$, and $1 \leq 42$ denotes $(1, 42) \in \leq$.

Mathematical Preliminaries: Partial Orderings

An *irreflexive partial ordering* $<$ on a set A is a relation on A that satisfies three properties:

1. **irreflexivity** $a \not< a$.
2. **antisymmetry** If $a < b$ then $b \not< a$.
3. **transitivity** If $a < b$ and $b < c$, then $a < c$.

Mathematical Preliminaries: Partial Orderings

For example, the strict subset relation $\subset$ is an irreflexive partial order on the powerset 2^A of some a set A.

1. **irreflexivity** $\{1, 2, 3\} \not\subset \{1, 2, 3\}$.
2. **antisymmetry** $\{1, 2\} \subset \{1, 2, 3\}$, so $\{1, 2, 3\} \not\subset \{1, 2\}$.
3. **transitivity** $\{1\} \subset \{1, 2\}$ and $\{1, 2\} \subset \{1, 2, 3\}$, so $\{1\} \subset \{1, 2, 3\}$.

Mathematical Preliminaries: Partial Orderings

For example, the strict subset relation $\subset$ is an irreflexive partial order on the powerset 2^A of some a set A.

1. **irreflexivity** $\{1,2,3\} \not\subset \{1,2,3\}$.
2. **antisymmetry** $\{1,2\} \subset \{1,2,3\}$, so $\{1,2,3\} \not\subset \{1,2\}$.
3. **transitivity** $\{1\} \subset \{1,2\}$ and $\{1,2\} \subset \{1,2,3\}$, so $\{1\} \subset \{1,2,3\}$.

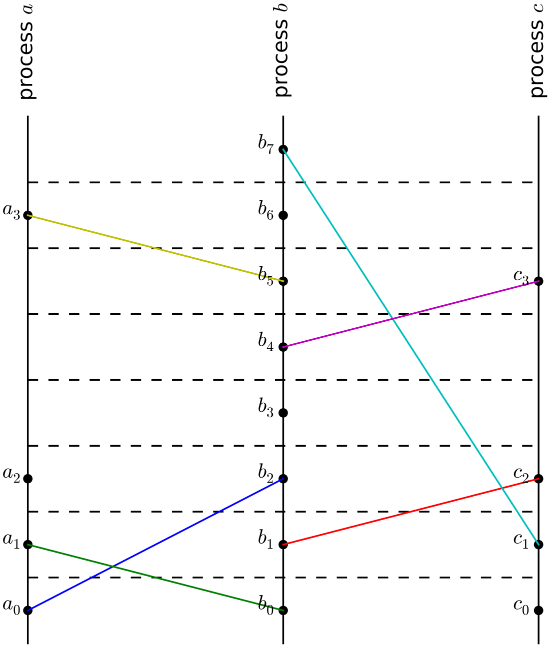
Note it's not always true that $a < b$ or $b < a$. For example, $\{1,2\} \not\subset \{2,3\}$ and $\{2,3\} \not\subset \{1,2\}$.

Mathematical Preliminaries: Total Orderings

A *irreflexive total ordering* $<$ on a set A is an irreflexive partial ordering on A that satisfies the additional property:

1. **totality** If $a \neq b$ then $a < b$ or $b < a$.

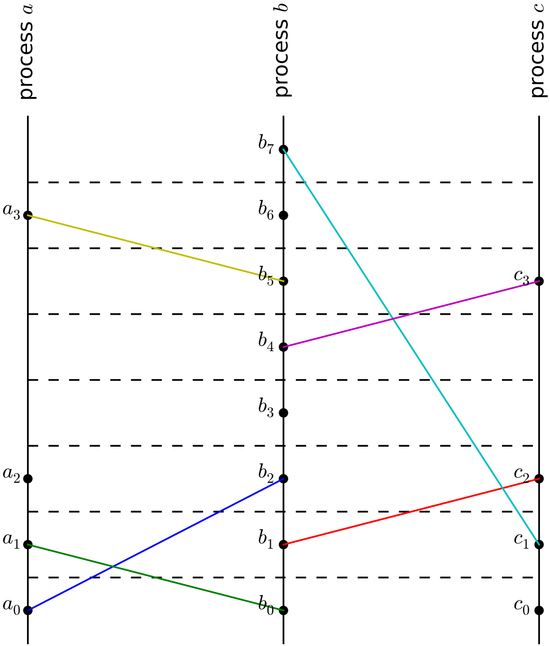
http://bit.ly/1WnG32c


process a
process b
process c
b_7
a_3
b_6
b_5
c_3
b_4
b_3
a_2
b_2
c_2
a_1
b_1
c_1
a_0
b_0
c_0

Constructing a Partial Order

We want to define an irreflexive partial ordering $\rightarrow$ on the set of events in a distributed system. Define $\rightarrow$ to be the smallest relation satisfying the following rules:

1. If a_i comes before a_j is a process a, then $a_i \rightarrow a_j$.
2. If a is the sending of a message and b is the receipt of the message, then $a \rightarrow b$.
3. If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$.


process a
process b
process c
b_7
a_3
b_6
b_5
c_3
b_4
b_3
a_2
b_2
c_2
a_1
b_1
c_1
a_0
b_0
c_0

Logical Clocks

Define a clock C as a function from events to natural numbers where we denote $C\langle a\rangle$ as the number assigned to a by C. A clock is correct if it satisfies the **Clock Condition**:

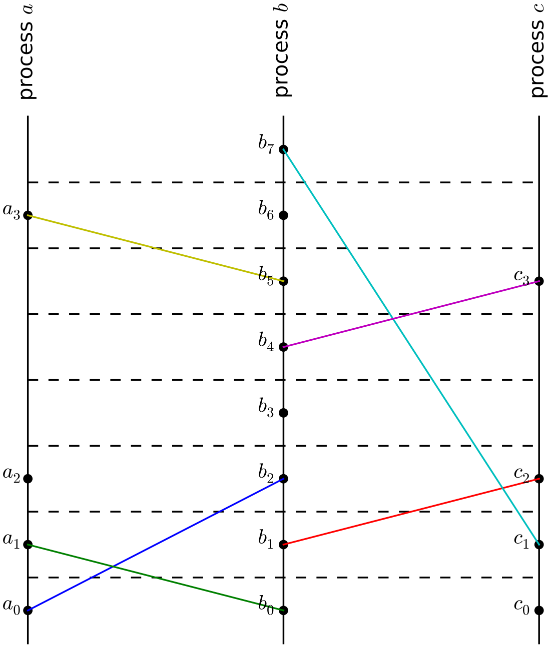
$$\forall a, b.\, a \rightarrow b \implies C\langle a\rangle < C\langle b\rangle$$

Note that the converse does not need to be satisfied!

Implementing Logical Clocks

Each process i maintains a register C_i. For an event a that occurs on process i, let $C\langle a\rangle$ be C_i at the time of a. Each process updates C_i as follows:

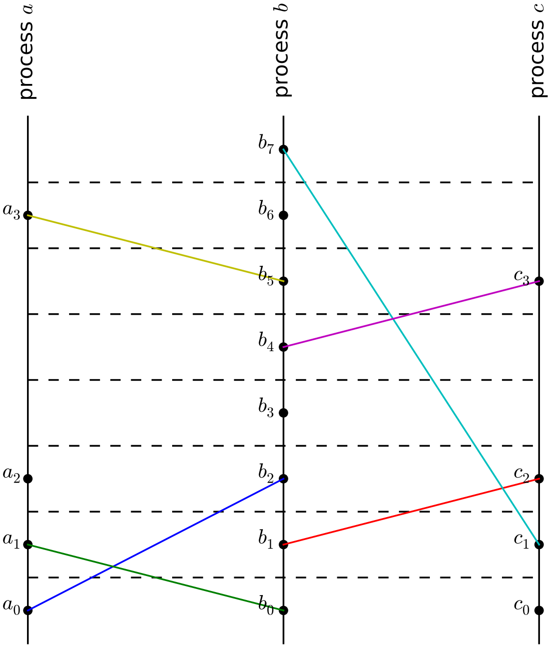
1. C_i is incremented between any two events.
2. If a is the sending of a message m from process i to process j, then m includes $C\langle a\rangle$ and j updates C_j to be larger than the old value of C_j and $C\langle a\rangle$.


process a
process b
process c
a_3
a_2
a_1
a_0
b_7
b_6
b_5
b_4
b_3
b_2
b_1
b_0
c_3
c_2
c_1
c_0

Constructing a Total Ordering

Consider an arbitrary total ordering $<$ on processes. Let's define a $\Rightarrow$ be a total ordering of events where $a_i \Rightarrow b_j$ if and only if

1. $C\langle a_i \rangle < C\langle b_j \rangle$, or
2. $C\langle a_i \rangle = C\langle b_j \rangle$ and $a < b$.

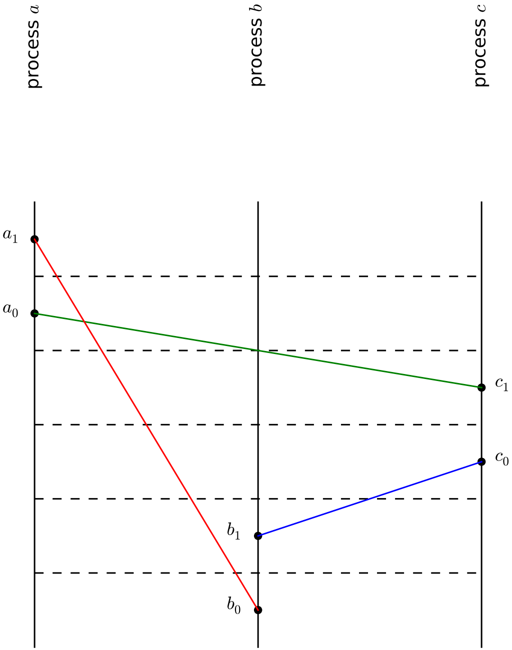

process a
process b
process c
b_7
a_3
b_6
b_5
c_3
b_4
b_3
a_2
b_2
c_2
a_1
b_1
c_1
a_0
b_0
c_0

Distributed Mutual Exclusion

A set of processes share a single resource that should be held by at most one processor at a time. We want an algorithm to enforce mutual exclusion such that:

1. **safety:** At most one process holds the resource.
2. **ordering:** Resource requests should be granted according to the happens before relation $\rightarrow$.
3. **progress:** If the resource is held for a finite amount of time, all requests will eventually be granted.

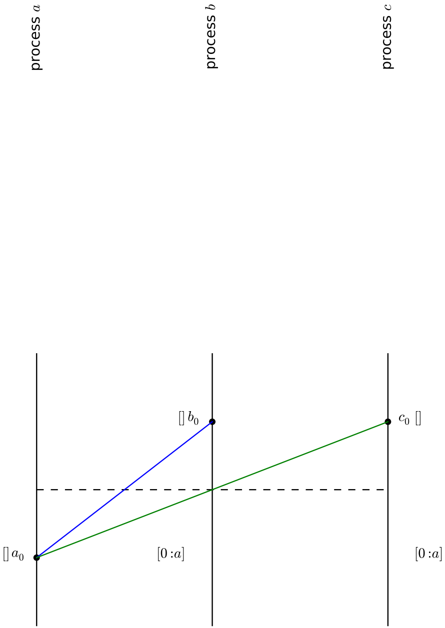
Assume processes form a clique and never fail and that the network guarantees reliable FIFO communication. Also assume one process has the resource initially.

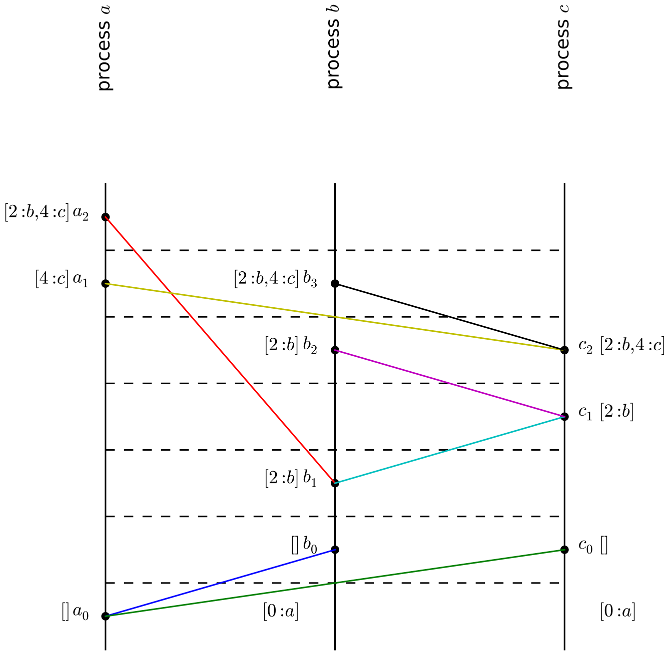

process a
process b
process c
a_1
a_0
c_1
c_0
b_1
b_0

Lamport's Mutual Exclusion Algorithm

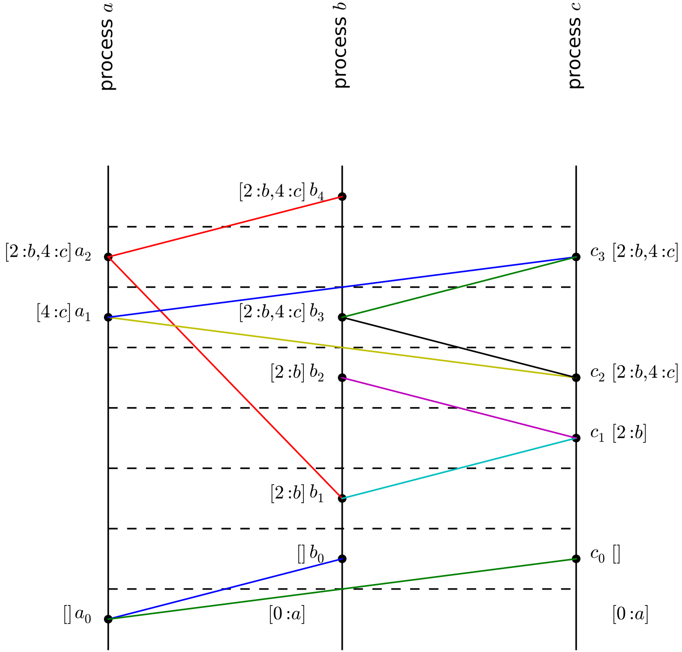
Each process maintains a *request queue* which initially contains $0 : p$. Each process follows five rules.

1. To request the resource, process a sends $i : a$ to all processes.
2. When a process receives $i : a$, it inserts it in the queue and acknowledges.
3. To release the resource, sends a release message to all processes.
4. When a process receives a release message from a it removes all $i : a$ from its queue.
5. Process a is granted the resource when the head of the queue is $[i : a]$ and it has seen acknowledgements from all processes later than i.

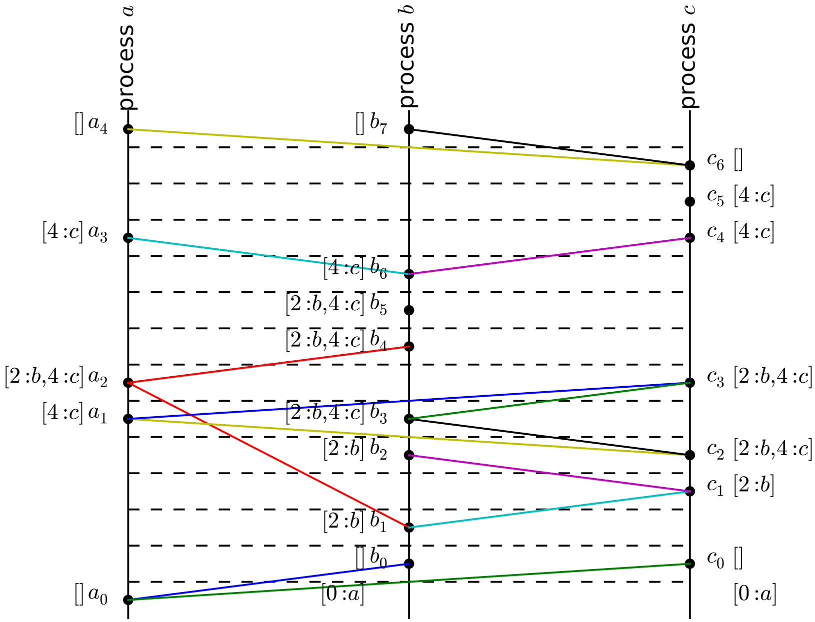

process a
process b
process c
$[]\,b_0$
$c_0\ []$
$[]\,a_0$
$[0:a]$
$[0:a]$



process a
process b
process c
$[2:b,4:c]\,a_2$
$[4:c]\,a_1$
$[2:b,4:c]\,b_3$
$[2:b]\,b_2$
$c_2\ [2:b,4:c]$
$c_1\ [2:b]$
$[2:b]\,b_1$
$[]\,b_0$
$c_0\ []$
$[]\,a_0$
$[0:a]$
$[0:a]$

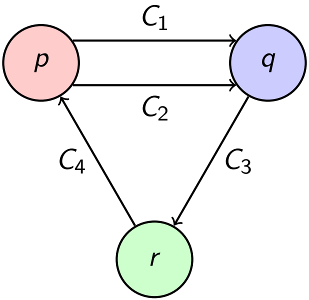


process a
process b
process c
$[2:b,4:c]\ b_4$
$[2:b,4:c]\ a_2$
$c_3\ [2:b,4:c]$
$[4:c]\ a_1$
$[2:b,4:c]\ b_3$
$[2:b]\ b_2$
$c_2\ [2:b,4:c]$
$c_1\ [2:b]$
$[2:b]\ b_1$
$[]\ b_0$
$c_0\ []$
$[]\ a_0$
$[0:a]$
$[0:a]$

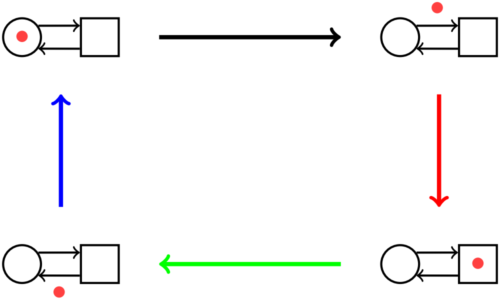


process a
process b
process c
$[]\, a_4$
$[]\, b_7$
$c_6\ []$
$c_5\ [4:c]$
$[4:c]\, a_3$
$c_4\ [4:c]$
$[4:c]\, b_6$
$[2:b,4:c]\, b_5$
$[2:b,4:c]\, b_4$
$[2:b,4:c]\, a_2$
$c_3\ [2:b,4:c]$
$[4:c]\, a_1$
$[2:b,4:c]\, b_3$
$[2:b]\, b_2$
$c_2\ [2:b,4:c]$
$c_1\ [2:b]$
$[2:b]\, b_1$
$[]\, b_0$
$c_0\ []$
$[]\, a_0$
$[0:a]$
$[0:a]$

http://bit.ly/1LIDXXG



C_1
p
q
C_2
C_4
C_3
r



Distributed System Model

A *process* p is a set of states S, an initial state s, and a set of events E.

$$p \triangleq (S, s, E)$$

Distributed System Model

A *process* p is a set of states S, an initial state s, and a set of events E.

$$p \triangleq (S, s, E)$$

An *event* $e \in E$ is defined by a process p, the state s and s' of p before and after e, the channel $(c|\perp)$ modified by e, and the message $(M|\perp)$ sent or received on c.

$$e \triangleq (p, s, s', (M|\perp), (c|\perp))$$

Distributed System Model

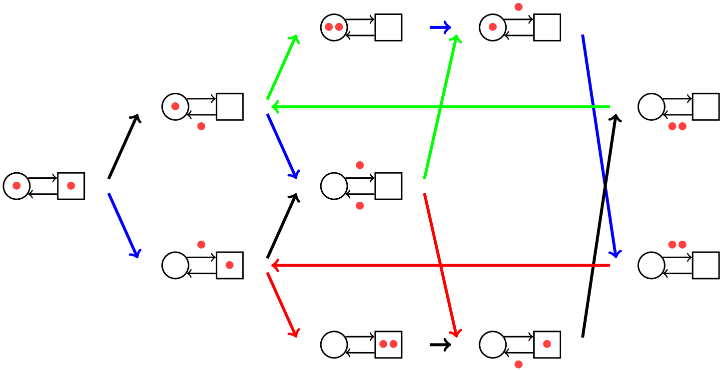
A *process* p is a set of states S, an initial state s, and a set of events E.

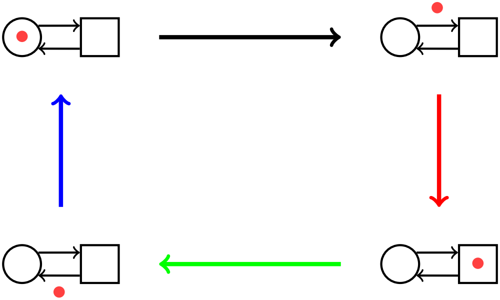
$$p \triangleq (S, s, E)$$

An *event* $e \in E$ is defined by a process p, the state s and s' of p before and after e, the channel $(c|\perp)$ modified by e, and the message $(M|\perp)$ sent or received on c.

$$e \triangleq (p, s, s', (M|\perp), (c|\perp))$$

A *global state* is a set of process and channel states. The *initial global state* has all processes in their initial states and all channels empty. A *computation of the system* is a sequence of events.



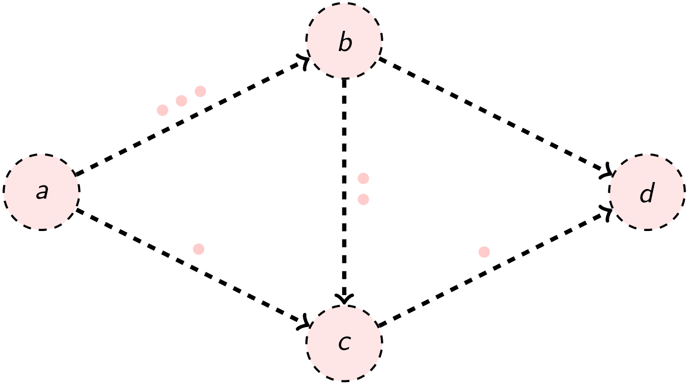


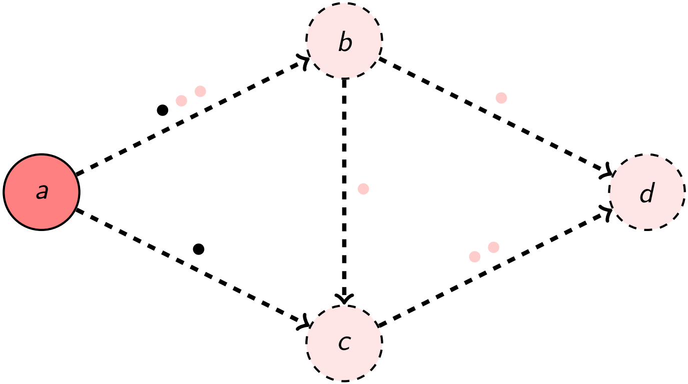
Snapshot Algorithm

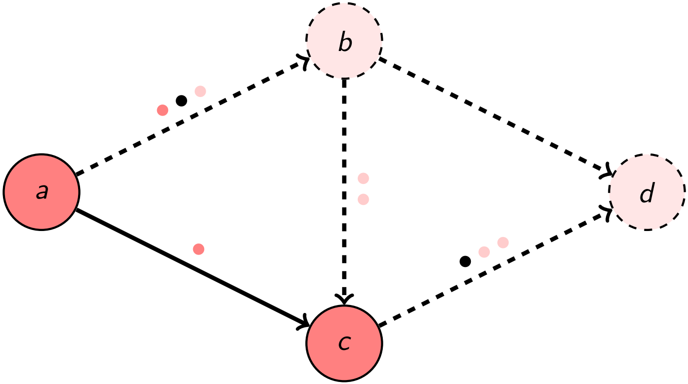
Marker-Sending Rule. For each process p and for each channel c away from p, p records its state and immediately sends a token along c.

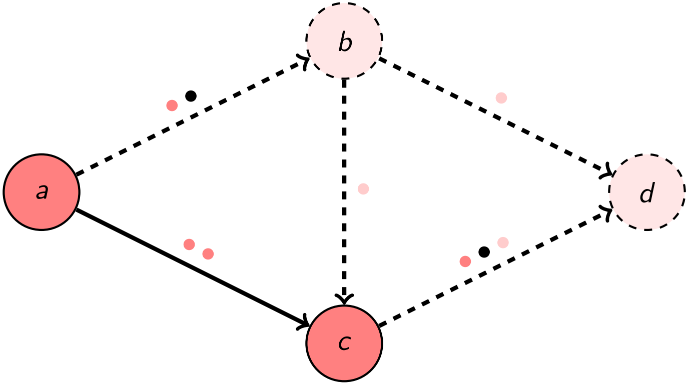
Marker-Receiving Rule. For each process q and for each channel c into q, when q receives a token from c,

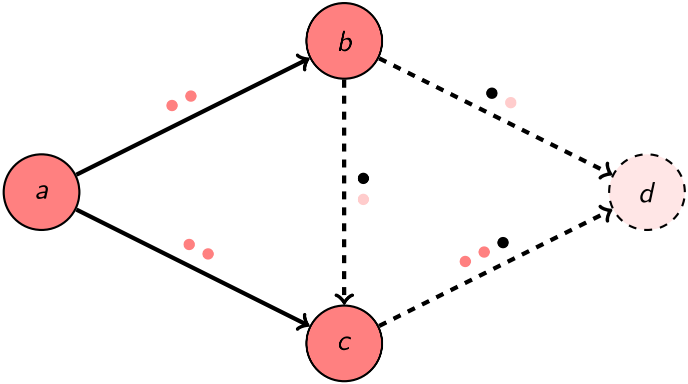
- If q has not yet recorded its state, it records its state and records the state of c as the empty sequence.
- If q has recorded its state, it records the state of c as the sequence of messages since it recorded its state.


a
b
c
d

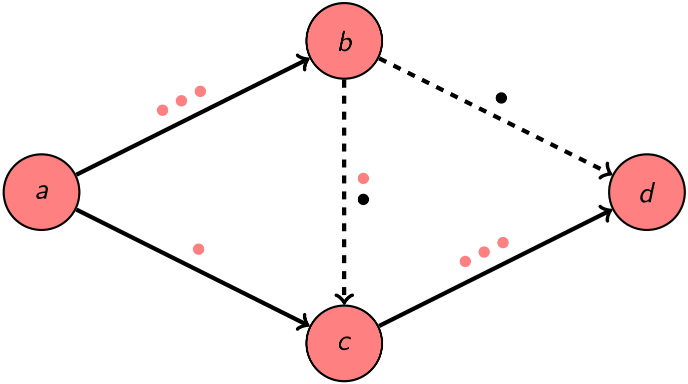

a
b
c
d

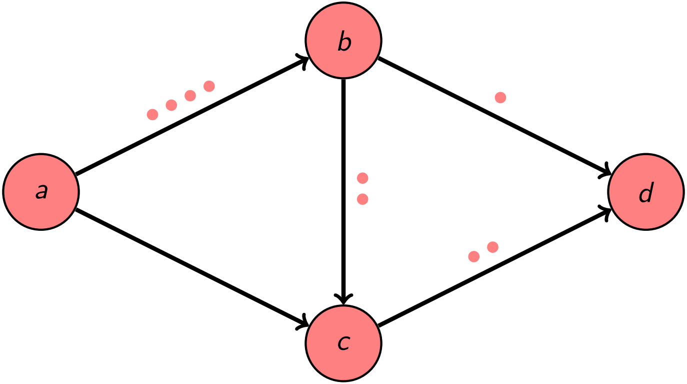

b
a
d
c


a
b
c
d

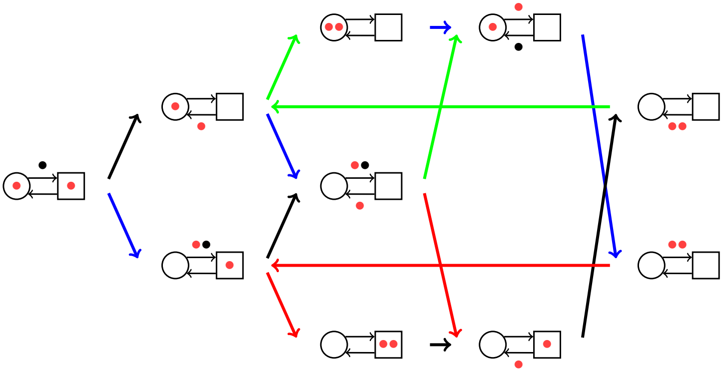


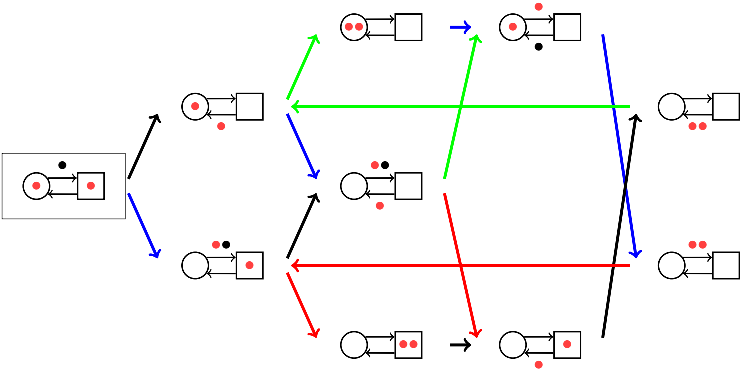
b
a
d
c

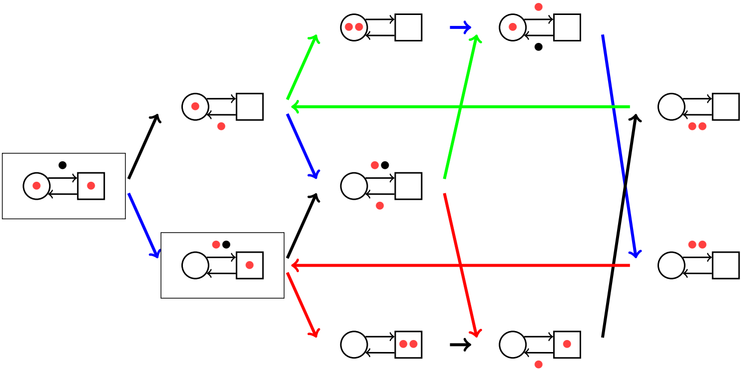

a
b
c
d

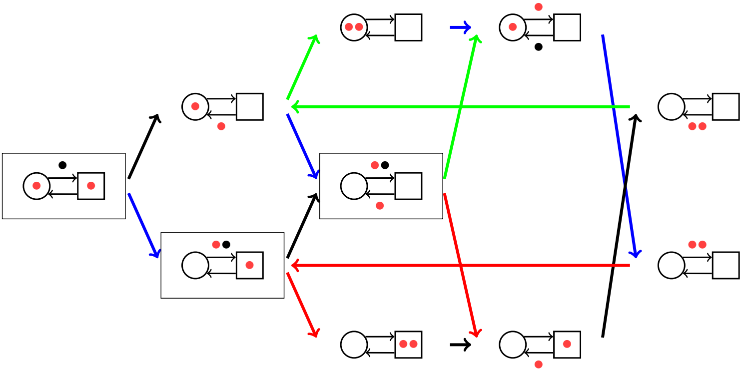


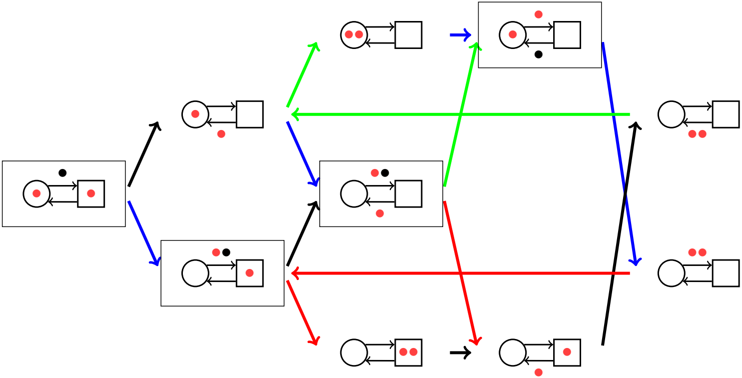
a
b
c
d

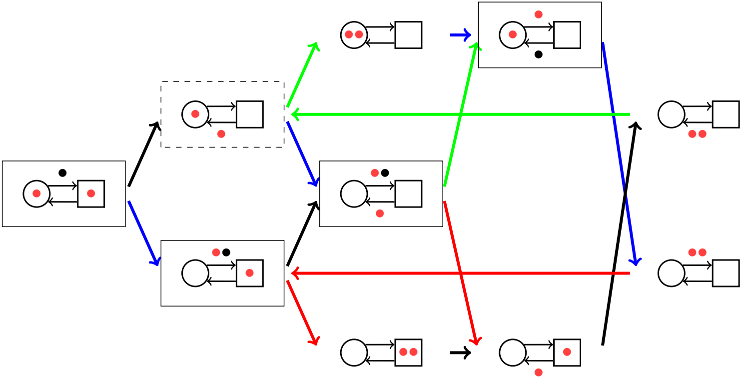


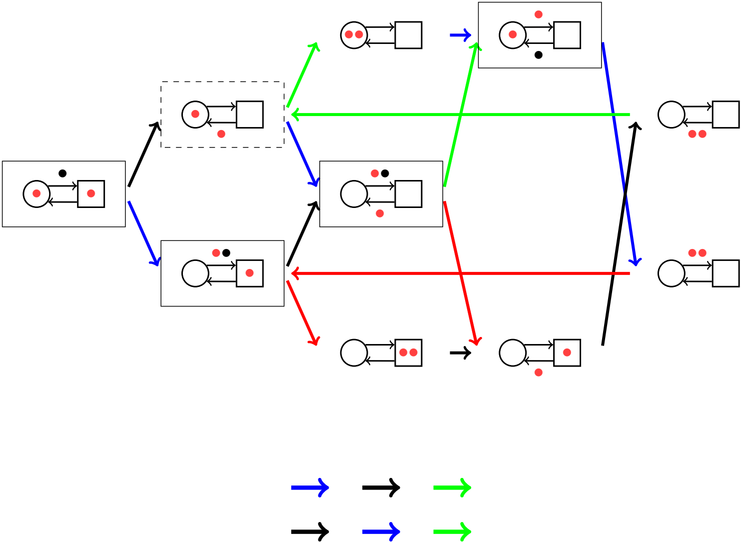












Snapshot Properties

Consider a computation $seq = S_0 e_0 S_1 e_1 \ldots S_\iota e_\iota \ldots S_\phi e_\phi \ldots S_n e_n$ where we initiate the snapshot algorithm in S_ι and the algorithm termiantes in S_ϕ. Denote the snapshot state S^*. We've seen that S^* might not be equal to any S_j for $\iota \leq j \leq \phi$. Howover, we can show that:

1. S^* is reachable from S_ι, and
2. S_ϕ is reachable from S^*.

Snapshot Properties

Even stronger, we can show that there exists a computation seq' such that:

1. For all $i < \iota$, $i \geq \phi$, $e'_i = e_i$, and
2. $(e'_j, \iota \leq j < \phi)$ is a permutation of $(e_j, \iota \leq i < \phi)$, and
3. there exists some $\iota \leq k \leq \phi$ such that $S^* = S'_k$.

Stable Properties

Conceptually, a stable property of a distributed system D is a property that is monotonically true. That is, once it becomes true, it remains true.

Stable Properties

Conceptually, a stable property of a distributed system D is a property that is monotonically true. That is, once it becomes true, it remains true.

For example, the following are stable properties:

- Our one-token system has one token
- Our two-token system has two tokens
- Computation has terminated
- Computation is deadlocked

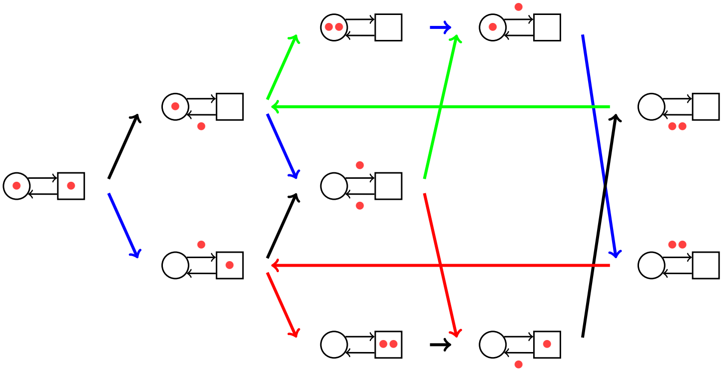
Stable Properties

Conceptually, a stable property of a distributed system D is a property that is monotonically true. That is, once it becomes true, it remains true.

For example, the following are stable properties:

- Our one-token system has one token
- Our two-token system has two tokens
- Computation has terminated
- Computation is deadlocked

Formally, a stable property y is a predicate on the global states S of a distributed system D. with the property that if $y(S)$ is true then $y(S')$ is true for all states S' reachable from S.



Stable Property Detection

We want to construct an algorithm that takes as input a distributed system D and a stable property y, and outputs a boolean b such that

$$y(S_\iota) \implies b, \qquad b \implies y(S_\phi)$$

Intuitively, if b is true, then $y(S_\phi)$ is true. If b is false, then $y(S_\iota)$ is false.

Stable Property Detection

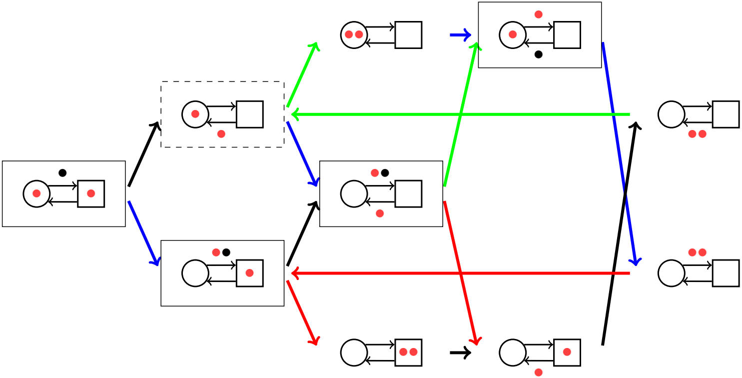
We want to construct an algorithm that takes as input a distributed system D and a stable property y, and outputs a boolean b such that

$$y(S_\iota) \implies b, \qquad b \implies y(S_\phi)$$

Intuitively, if b is true, then $y(S_\phi)$ is true. If b is false, then $y(S_\iota)$ is false.

The algorithm itself is trivial:

1. Record a global state S^*
2. Output $y(S^*)$



Discussion

Discussion

- Lamport clocks map a set of partially ordered events to a totally ordered set. Does this make sense? If not, how could we improve on Lamport clocks?
- How adequate are the system models presented in the papers?
- Is it reasonable to abandon physical clocks because of their inaccuracy, or is that an overreaction? Can physical clocks and logical clocks be combined?

https://github.com/mwhittaker/clock_snapshot_slides