Advanced Network Security

1. Course Outline

Jaap-Henk Hoepman

Digital Security (DS)
Radboud University Nijmegen, the Netherlands
@xotoxot // ✉ *jhh@cs.ru.nl* // 🖱 *www.cs.ru.nl/~jhh*

About me

About you

Applied cryptography?

Administrative details

- **Course code:**
  - NWI-IMC050, 5 ects
- **Teachers:**
  - Jaap-Henk Hoepman
  - Harald Vranken
- **Written exam**
- **Course website**
  - Not using Blackboard
  - Instead see: http://www.cs.ru.nl/~jhh/ans.html
- **Literature**
  - Selected papers, see website.
- **Excercises**
  - See website.

Schedule: see website

Date	Topic	Literature
February 3	Introduction to distributed algorithms : slides	Papers: - L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System." Communications of the Association for Computing Machinery 21, no. 7 (July 1978): 558-565. (upto the section called "Physical clocks")
February 10	'Prevention' and 'Intrusion detection systems and netflows: slides	Papers: - Rick Hofstede, Pavel Čeleda, Brian Trammell, Idilio Drago, Ramin Sadre, Anna Sperotto, Aiko Pras, "Flow Monitoring Explained: From Packet Capture to Data Analysis With NetFlow and IPFIX", IEEE Communications Surveys & Tutorials (Volume: 16, Issue: 4, Fourthquarter 2014). *Additional, optional, reading::* - Rick Hofstede, Vaclav Bartos, Anna Sperotto, Aiko Pras: Towards real-time intrusion detection for NetFlow and IPFIX. Proceedings 9th International Conference on Network and Service Management (CNSM), 2013 - Rick Hofstede, Luuk Hendriks, Anna Sperotto, Aiko Pras: SSH Compromise Detection using NetFlow/IPFIX. ACM SIGCOMM Computer Communication Review archive, Volume 44, Issue 5, Oct. 2014, p. 20-26 - Anna Sperotto, Ramin Sadre, Pieter-Tjerk de Boer, and Aiko Pras: Hidden Markov Model Modeling of SSH Brute-Force Attacks. Lecture Notes in Computer Science, vol. 5841, 2009, Springer, p. 164-176 - Laurens Hellemons, Luuk Hendriks, Rick Hofstede, Anna Sperotto, Ramin Sadre and Aiko Pras: SSHCure: A Flow-Based SSH Intrusion Detection System Lecture Notes in Computer Science, vol. 7279, 2012, Springer, p. 86-97.
February 17	Economics/governance of network security: slides	Papers: - Why information security is hard – an economic perspective Ross Anderson Proceedings 17th Annual Computer Security Applications Conference (ACSAC), 2001 - So long, and no thanks for the externalities: the rational rejection of security advice by users Cormac Herley Proceedings of the 2009 workshop on New security paradigms workshop (NSPW) *Additional, optional, reading::* - 'Hacks, sticks and carrots' by prof.dr. Michel van Eeten - Economics of fighting botnets: lessons from a decade of mitigation H. Asghari, M.J.G. van Eeten, J.M. Bauer IEEE Security & Privacy, September/October 2015, 16–23 - Cybersecurity: Stakeholder incentives, externalities, and policy options J.M.Bauer & M.J.G. van Eeten Telecommunications Policy 33(2009):706–719
February 24	(no lecture)	
March 2	Distributed Algorithms: Leader Election: slides	Papers: - G.L. Peterson, "An O(n log n) unidirectional algorithm for the circular extrema problem". ACM TOPLAS 4 (1982), 758–762.
March 9	Distributed Algorithms: Mutual Exclusion: slides	Papers: - L. Lamport, "A new solution of Dijkstra's concurrent programming problem." Commun. ACM 18, 8 (1974), 453–455.
March 16	Wifi security slides	Papers: - S. Brenza, A. Pawlowski, and C. Pöpper: A Practical Investigation of Identity Theft Vulnerabilities in Eduroam Proceedings 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks, 2015 - M. Vanhoef and F. Piessens: Key Reinstallation Attacks: Forcing Nonce Reuse in WPA2 Proceedings 24th ACM Conference on Computer and Communication Security, 2017 (Note: you may skip sections 4, 5, and 7)
March 23	(no lecture)	
March 30	(no lecture)	
April 6	(no lecture)	
April 13	(no lecture)	
April 20	Self-Stabilisation: slides	Papers: - E.W. Dijkstra, "Self-Stabilizing Systems in Spite of Distributed Control." Communications of the Association for Computing Machinery 17, no. 11 (November 1974): 643-644.
April 27	(no lecture)	
May 4	Joeri de Ruiter, "Routing security: BGP and future internet architecture" slides	Papers: The SCION Internet Architecture - An Internet Architecture for the 21st Century David Barrera, Laurent Chuat, Adrian Perrig, Raphael M. Reischuk, Pawel Szalachowski Communications of the ACM 60 (6), June 2017
May 11	Agreement and consensus I: concepts and protocols for crash failures: slides	Papers: - M. Pease, R. Shostak, L. Lamport. "Reaching Agreement in the Presence of Faults" (PDF). Journal of the ACM. 27 (2): 228–234, April 1980.
May 18	Agreement and consensus II: handling Byzantine failures: slides	Papers: - L. Lamport, R. Shostak, M. Pease, "The Byzantine Generals Problem", ACM TOPLAS 4(3), pp. 382-401, July 1982.
May 25	Botnets slides	Papers: - Sheharbano Khattak, Naurin Rasheed Ramay, Kamran Riaz Khan, Affan A. Syed, and Syed Ali Khayam: "A Taxonomy of Botnet Behavior, Detection, and Defense"
June 1	(no lecture)	
June 8	Fabian v.d. Broek "Mobile telephony security" slides	Papers: Chapters 2 and 3 of "Mobile communicaton security", Fabian van den Broek, PhD thesis, 2016

Rough lecture setup

- **Several papers per lecture**
  - Read them in advance
  - Ask questions you may have about them after lecture
- **Homework**
  - Not graded
  - But discussed at start of next lecture
  - You are strongly advised to do the homework to get some exercise before the exam

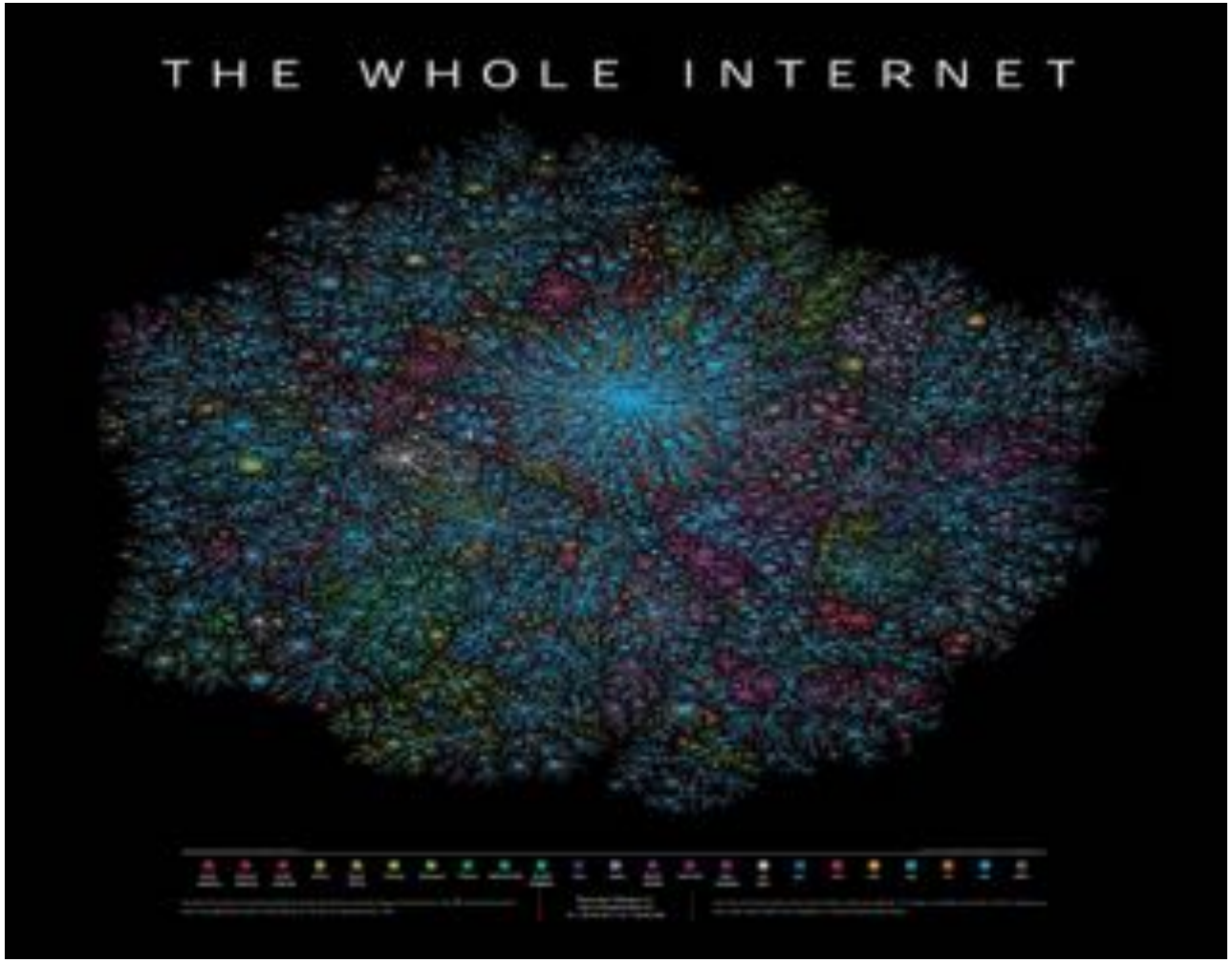
Course contents

Advanced Network Security

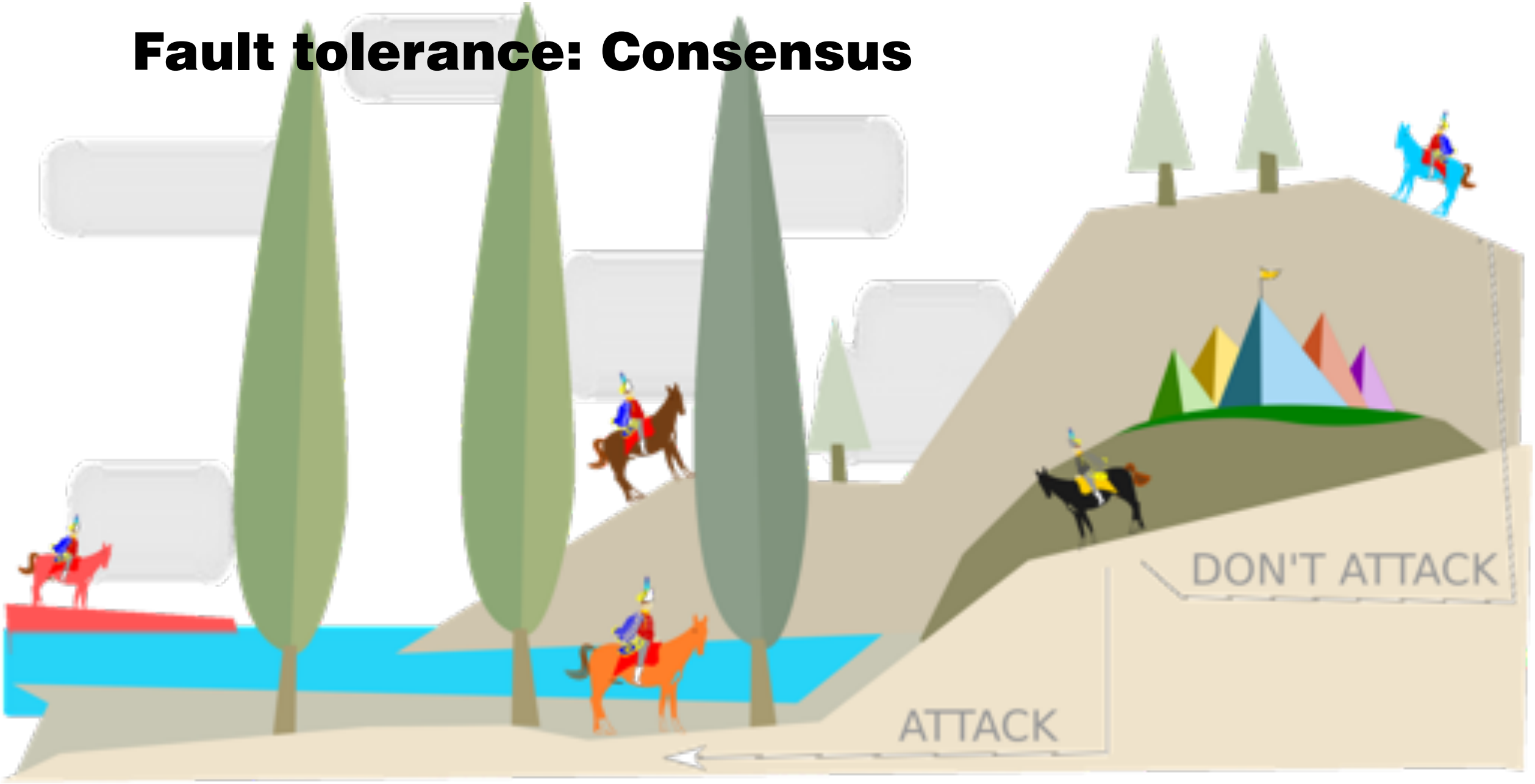
CIAvailability

- **Two quite separate and distinct parts**
  - Distributed algorithms, in particular fault tolerance
  - More traditional advanced network security, like intrusion detection, wireless/cellular network security, etc.

Distributed algorithms



Fault tolerance: Consensus



Consensus

- How could you solve it?

Fault tolerance: self stabilisation

Self-stabilisation

- How could it be achieved?

Meta-knowledge

- **Distributed algorithms**
  - Modelling
  - Reasoning
  - Designing: “Algorithmics”
- **Global ‘emergent’ behaviour based on local decisions**
- **The ‘forgotten’ security properties**
  - Availability
  - Privacy

Questions? If not, let's hit the road!

✉ *jhh@cs.ru.nl* 🖱 *blog.xot.nl* 🖱 *www.cs.ru.nl/~jhh* *twitter: @xotoxot*

Basics of distributed algorithms

Distributed algorithms are everywhere!

- **Computer networks**
  - Message passing
  - Routing
  - DNS
  - …
- **Multi-threaded applications**
  - Shared memory / message passing
  - User interface
  - Browser loading page elements
  - Operating system
  - Services
  - Parallel processing

A distributed system has concurrency

- **Several processes**
  - Each executing actions sequentially
- **But working independently**
  - Individual actions on different processes may overlap in time
  - Some actions may take much longer to complete than others
- **And order of actions *on different processes* not guaranteed**

Concurrency: a basic example

```
i := 1,
j := 0,

thread i := 2 end thread;
thread j := i end thread;

print j
```

Assume for the moment this is an atomic operation

- **What will be the output of this simple program?**
- **It depends**
  - Events never take place instantaneous
  - Order of execution is not fixed; determined by scheduler

A basic example: answer

```
i := 1,
j := 0,

thread i := 2 end thread;
thread j := i end thread;

print j
```

- 0
- 1
- 2

A slightly more complex example

```
i := 1,
j := 0,

thread i := 2;
       print j
end thread;
thread j := i;
       print j
end thread;
```

- **Again, what will be the output of this program?**

A slightly more complex example: answer

- **2 2**
  ⊢ i:=2 ⊣ ⊢ print j ⊣ → 2
  ⊢ j:=i ⊣ ⊢ print j ⊣ → 2
- **0 2**
  ⊢ i:=2 ⊣ ⊢ print j ⊣ → 0
  ⊢ j:=i ⊣ ⊢ print j ⊣ → 2
- **1 1**
- **0 1??**
  - Depends….

0 1

~~1 0~~ ← ?

2 2

0 2

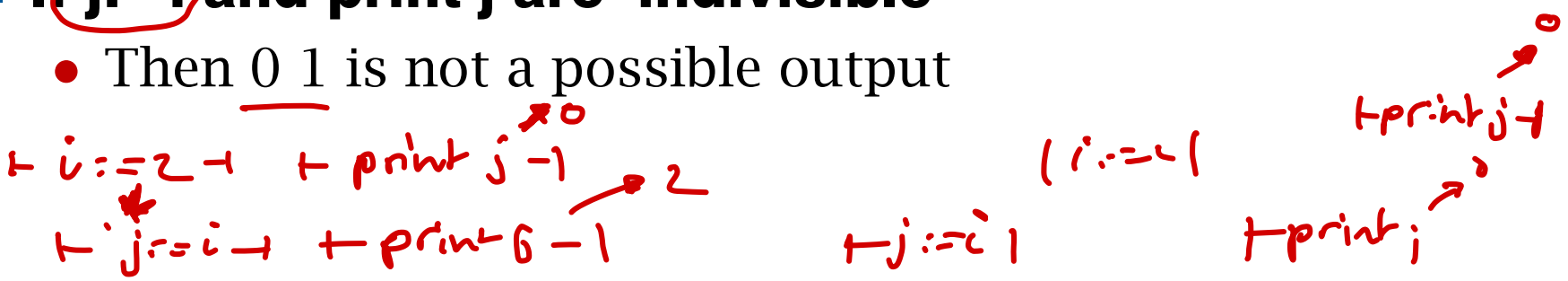
~~2 0~~ ← ?

```
i := 1,
j := 0,

thread i := 2;
       print j
end thread;
thread j := i;
       print j
end thread;
```

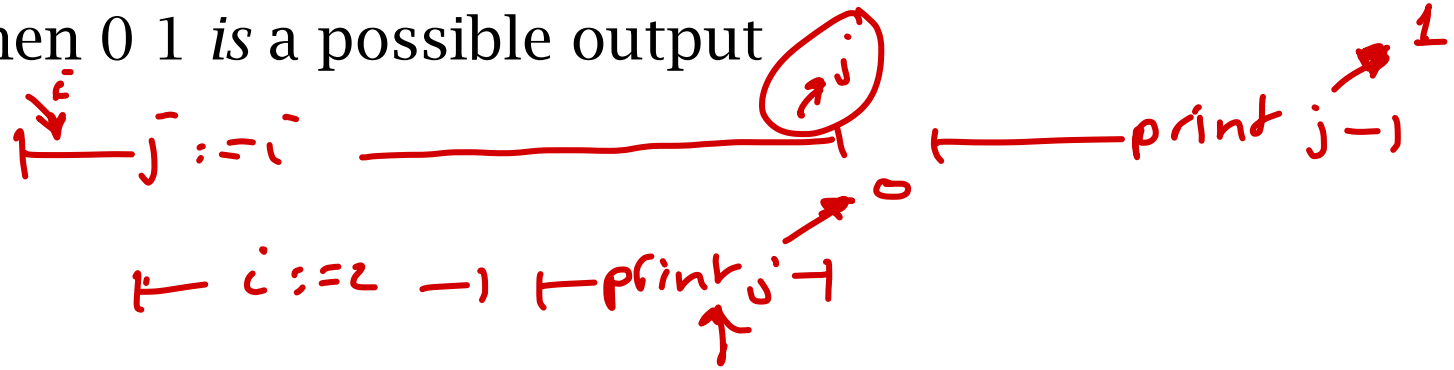
Indivisibility of events/atomicity

```
i := 1,
j := 0,

thread i := 2;
        print j
end thread;
thread j := i;
        print j
end thread;
```

- **If j:=i and print j are 'indivisible'**
  - Then 0 1 is not a possible output



- **If j:=i is a read of i followed by a write to j, or if print j is a read of j followed by writing the output to the screen**
  - Then 0 1 *is* a possible output



What about infinite executions?

```
i := 0;

thread while i == 0
          do print i ;
          print i
end thread;

thread i := 1
end thread;
```

Note: indentation!

- **Possible outputs:**

11
011
0011
0001 1

1
01
001
0001

Scheduling

- **Scheduler determines next action to be executed**
  - In a non-deterministic way
  - An executed action is called an event

- **Note: scheduler is *not* a real system component**
  - It just models the influence of (external) factors on which action is executed

- **Fairness:**
  - A scheduler is *fair* if an action that is *continuously* enabled will always executed *eventually*

  - So: the protocol on the previous slide will eventually terminate

Non determinism vs randomness

Non-deterministic

```
i := 0,

thread while i == 0
       do print i;
       print i
thread i := 1
```

We cannot say anything about likelihood!

Possible outputs:

```
1
01
001
0001
00001
```

Random

```
i := 0,

while i == 0
do i := random(0,1) ; print i
```

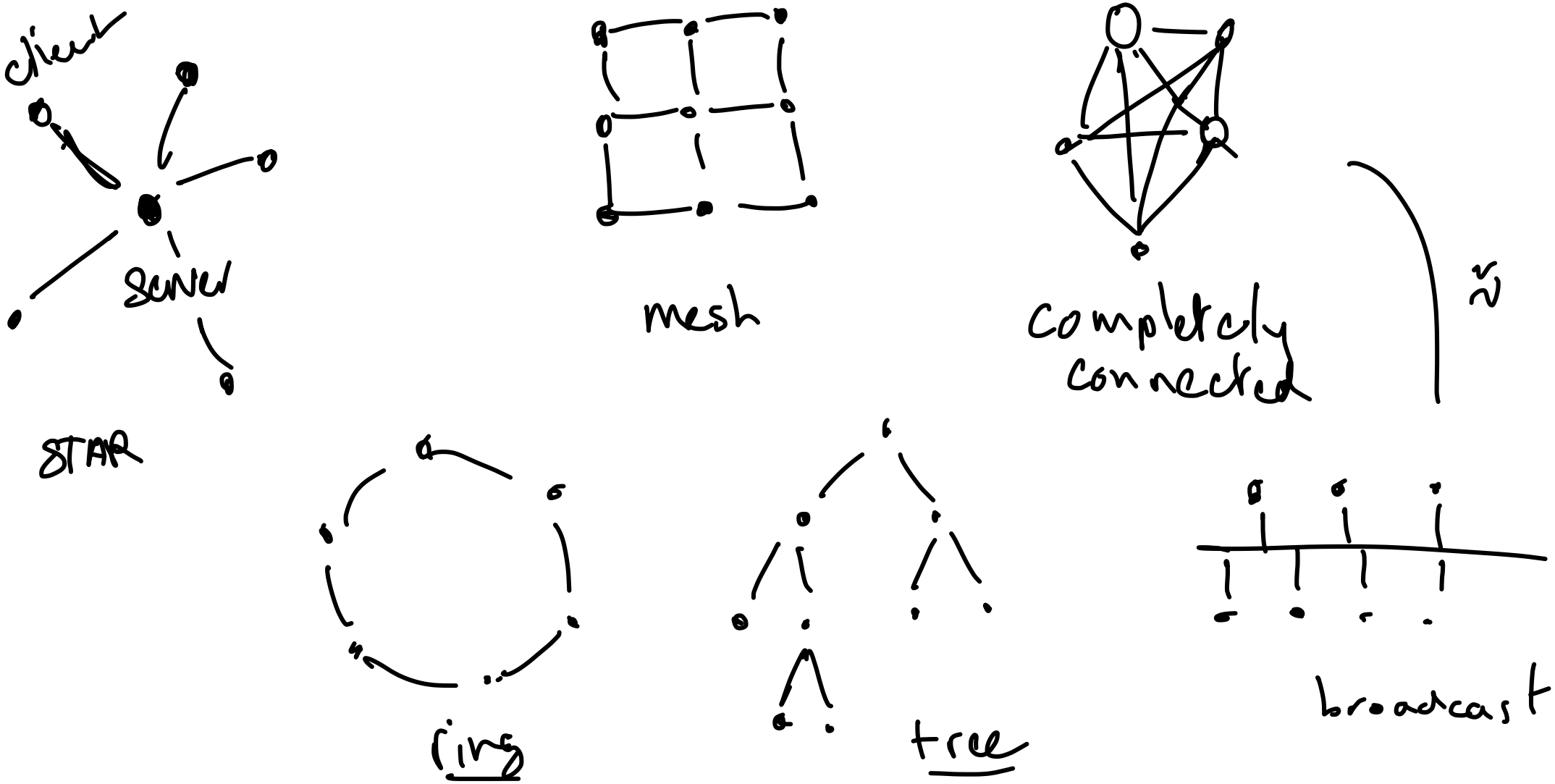
Possible outputs:

```
1      p = 0.5
01     p = 0.25
001    p = 0.125
0001   p = 0.063
00001  p = 0.031
```

Modelling a distributed system

- **Node (aka process)**
  - Executes a sequence of actions
  - Each action-execution is an event
  - Communicates with other nodes through shared memory or message passing
- **Graph $G = (V, E)$ of nodes V and edges E**
  - $N = |V|$ the number of nodes
  - $(v, w) \in E$ if node v can communicate data to w
  - Graph can be directed or undirected

Some common topologies



Causality: Ordering events

while true
do print j ← action

print j ; print j ---- ← events

- **Let A be the set of events, and let $a, b \in A$ be events**
- **Define the 'happened before' relation $a \to b$ as follows .**
  - If a is executed before b by the same process, or
  - (Message passing) If a is a send event whose value is received by receive event b, or
  - (Shared memory) If a is a write event whose value is read by read event b.
- **Transitivity: if $a \to b$ and $b \to c$ then $a \to c$**
- **We assume $a \nrightarrow a$**
- **If neither $a \to b$ nor $b \to a$ then they are concurrent: $a \parallel b$**
  - We sometimes write $b \nleftarrow a$ to visually emphasise that b may precede a.
- **$a \to b$ means that a can have a causal influence on b**
- **This is a irreflexive partial order over all events**
  - Defined by only looking what can externally be observed
  - Does not depend on 'global time'
  - Extending the *total* order of events on each individual process
  - Modelling a *causal order* among events

i, j local variables

node a : ← i:=2 → ← j:=i → ← print j ←

node b : ← k:=2-1 → ← r:=1-1

i shared variable

⇓

{i=0}

node a : ← i:=2-1

node b : {k=φ} ← k:= read i ←

{k=2}

{i=0} ← i:=2-1

← k:= read i-1

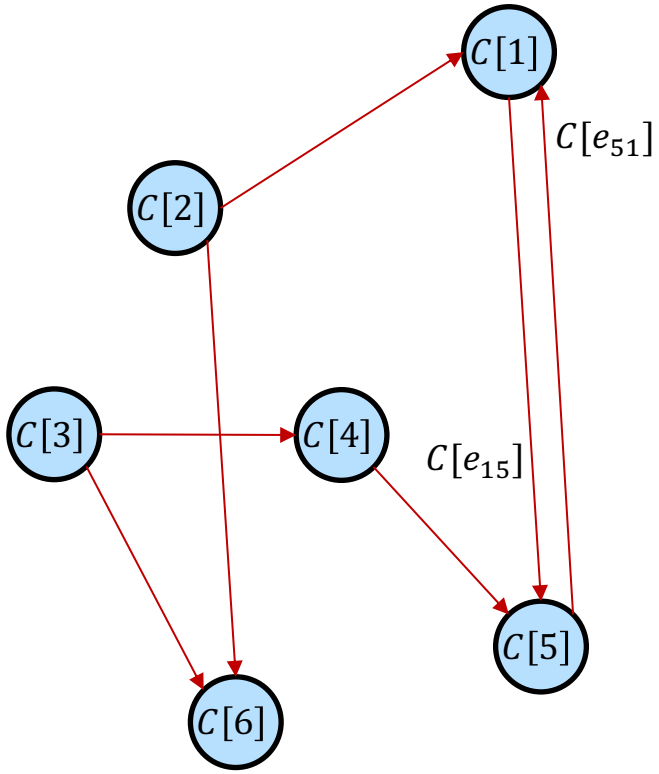
{k=0}

Execution

- **When we run a distributed system, the nodes execute their actions. This leads to**
  - a particular set of events A
  - partially ordered using the happened before relation $\rightarrow$
  - Note: every run may result in a different set of events and a different order among them
- **This partial order can be extended to a total order** $\Rightarrow$
  - (There are often many different options)
- $\langle A, \Longrightarrow \rangle$ **is an *execution* of the system**

State, configuration, evolution

- **Every node has a local *state***
- **Every edge has also a state**
  - *Either the values in the shared variables or the messages in the message buffers*
- **The global state of the system is the Cartesian product of all local states and edge states, and called the *configuration* C**
  - *We write $C[i]$ for the state of node i, and $C[i].variable$ for the value of that variable in i's state.*
  - *We write $C[e]$ for the state of edge e. In the shared memory model, $C[e].variable$ denotes the value of that shared variable*



Execution, evolution

- **A node i may have 0 or more actions enabled**
  - Depending on its local state $C[i]$
  - As well as the states of the incoming edges $C[e_{ji}]$
- **An execution $\langle A, \Longrightarrow \rangle$ induces an evolution of the state of the system**
  - Let $a_1, a_2, a_3, \ldots$ be the sequence of events such that $a_i \Rightarrow a_j$ iff $i < j$
  - Let C_0 be the initial configuration
  - Let a_i change C_{i-1} into C_i
  - Then $C_0, C_1, C_2, \ldots$ is the evolution of the system

Execution, evolution (continued)

- **If we want to prove property P about a distributed system, we need to prove P for all possible executions.**
  - Possible executions are restricted by the 'logic' of the distributed system
  - This logic is captured by the happened before relation $\rightarrow$ (which indirectly also captures details of the topology of the network)
- **In other words: to prove property P we need to show that for all total orders $\Rightarrow$ that extend $\rightarrow$ we have that P holds for $\langle A, \Longrightarrow \rangle$**
- **Such a total order is consistent with *all* local observations**
  - Global observations ("god's view") do not exist…

Exploiting causality

- The happens before relation → (i.e. causality) restricts the set of possible executions

- Designing a distributed system that satisfies or reaches a certain property means we have to use the design to shape the → relation so it ensures this property is acheived

- Example: how to guarantee message delivery?

3-2-2020

Partial Orders

~~Guaranteeing message delivery~~

$b \,?\, c$

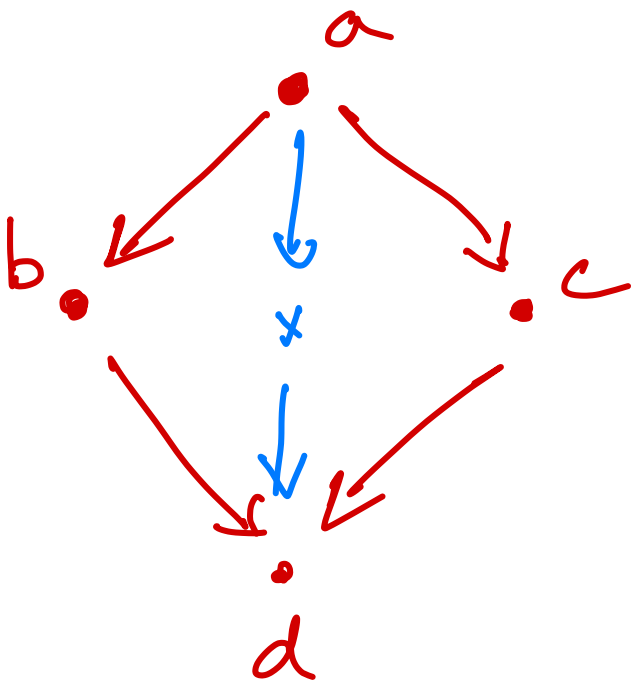
$b \,?\, x$

$x \,?\, c$

$b \Rightarrow x$

$x \Rightarrow c$

$c \Rightarrow b$ ↯



transitive

$$\left.\begin{array}{l} a \to b \\ b \to d \end{array}\right\} \to a \to d$$

$$\left.\begin{array}{l} a \to c \\ c \to d \end{array}\right\} \to a \to d$$

Total ordering (from partial order) $\Rightarrow b \,?\, c$

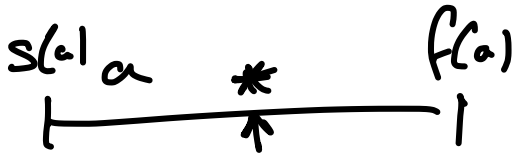
$\Rightarrow$

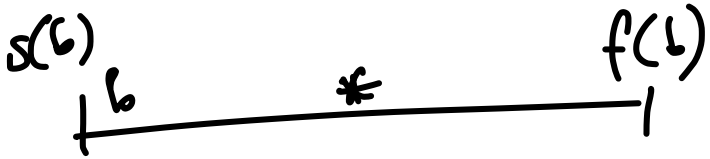
$x \to y : x \Rightarrow y$

$x \,?\, y$ — $x \Rightarrow y$ / $y \Rightarrow x$

Atomicity

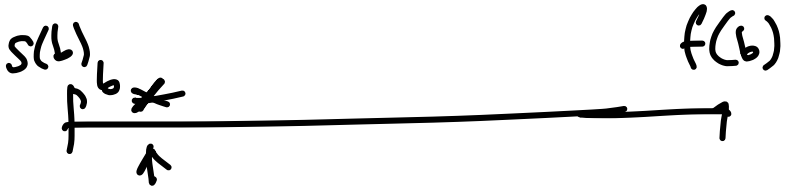
- **So far we have assumed events are instantaneous**
  - But in reality all events take time, sometimes a long time, to complete
- **For $a \in A$, let $s(a)$ be the local start time and $f(a)$ be the local end time**
  - Clearly $s(a) < f(a)$
  - Also, for $a, b \in A$ from the same process, either $f(a) < s(b)$ or $f(b) < s(a)$.
    - *(No events of the same process can overlap.)*
  - For $a, b \in A$ from the same process, if $f(a) < s(b)$ then $a \to b$
  - For $a, b \in A$ from different processes, let $a \to b$ if a causally influences b
    - *By writing to shared memory or sending a message*
  - This again fixes the partial order $\to$
- **Now assume a takes place instantaneously somewhere between $s(a)$ and $f(a)$**
  - This creates a total order $\langle A, \Longrightarrow \rangle$ consistent with $\to$
- **We call a atomic**
  - Atomicty is *an assumption* on the behaviour of actions
  - Atomic operations can be constructed using protocols using smaller building blocks.





$$\Downarrow$$

$$a \Rightarrow b$$



$$\Downarrow$$

$$b \Rightarrow a$$

Back to slightly more complex example

- **Serialisation**
  - The process of ‘shrinking’ actions to a single point in time and creating a total order.
- **Assume j := i is an atomic operation**
- **Now assume only reads and writes are atomic**

```
i := 1,
j := 0,

thread i := 2; print j end thread;
thread j := i; print j end thread;
```

```
i := 1,
j := 0,

thread i := 2;
       x := j;
       print x
end thread;

thread y := i; j := y;
       y := j;
       print y
end thread;
```

read variable v

→ read returns 2

v = 2

read returns 10

write value 10 to v

test & set

read modify write

test & set (b) b : boolean

<u>if</u> b = 0

<u>then</u> b ← 1

<u>return</u> b

b = 0

output

t&s(b)

node a

0 1

node b

1 0

t&s(b)

Logical clocks (Lamport paper): model

- **Goal**
  - Totally order events
- **Model**
  - Every event a of node i is assigned a natural number $C_i(a)$, the local clock value of node i
  - The logical clock C is defined by $C(a) = C_i(a)$ for event a that belongs to node i
  - Correctness: $a \rightarrow b$ implies $C(a) < C(b)$

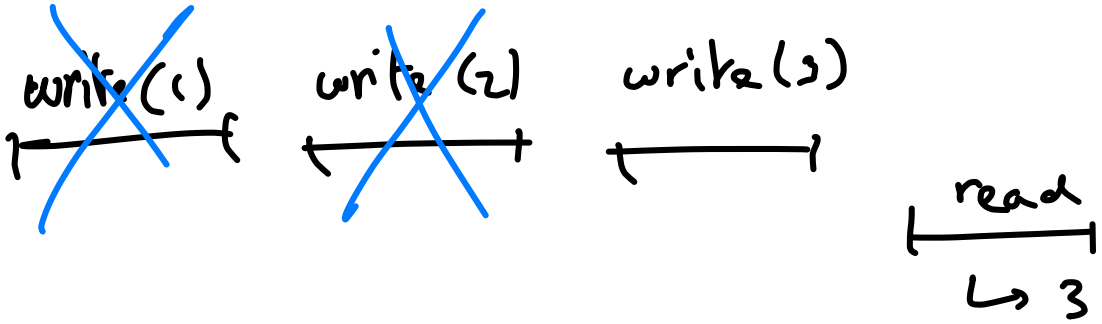
+ either $C(a) < C(b)$ or $C(b) < C(a)$: $\Longrightarrow$

Logical clocks (Lamport paper)

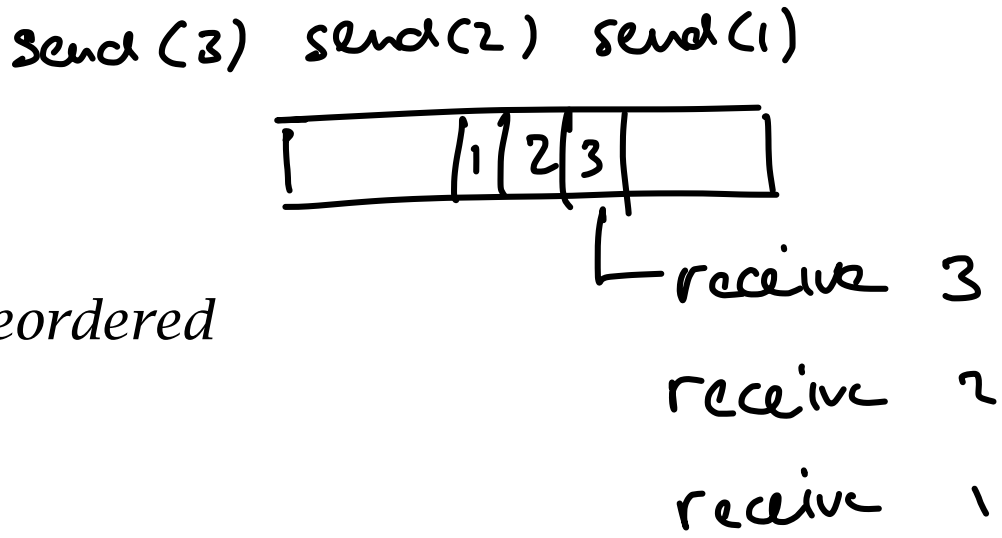
- **Implementation**
  - Each node i maintains a counter c_i
  - Initially $c_i = 0$ for every node i
  - Nodes increment c_i after every event
  - $C_i(a)$ is assigned the value of c_i just before event a
  - If a is a send event of message m, then this message is labelled with $T_m = C_i(a)$
  - When receiving a message m with event b by node j, $c_j \leftarrow \max(c_j, T_m + 1)$ after which $C_j(b) = c_j$
- **Events are now totally ordered (by clock and node number, $\langle C_i(a), i \rangle$)**
  - But this is not the only possible order!

Communication

- **Shared memory**
  - Write overwrites previous value
  - Read returns last value written



- **Message passing**
  - (Infinite) buffers
  - Usually FIFO: First-In-First-Out
    - ★ *But messages may get dropped or reordered*



Properties of distributed systems

- **Communication**
  - Shared memory vs message passing
- **Timing**
  - Synchronous, partially synchronous, asynchronous
- **Scheduling**
  - Fair, round-robin
- **Identity**
  - Uniform, random identifiers, 0..N-1
- **Network topology**
  - G = (V, E) , N = |V|
  - And whether the size of the network N is known
  - Directionality
- **Sense of direction**
- **Run/execution**

Reasoning about distributed systems

- **Prove global property based on local actions**
- **Measure the cost of achieving this**
  - Worst-case, average case
  - Time complexity: number of rounds, number of steps
  - Message complexity
  - Bit complexity
- **Prove lower bounds**
- **Prove impossibililty**

Round complexity

Remember fairness

- **Round**
  - The smallest period in which *all* nodes in the system have executed at least one action
- **Any execution can uniquely split into rounds**
- **Round complexity**
  - The number of rounds in an execution

- **Rationale**
  - Slowest node usually delays progress
  - Without round complexity, asynchronous systems would have unbounded time complexity

Round complexity

Remember fairness

- **Round**
  - The smallest period in which *all* nodes in the system have executed at least one action
- **Any execution can uniquely split into rounds**
- **Round complexity**
  - The number of rounds in an execution

- **Rationale**
  - Slowest node usually delays progress
  - Without round complexity, asynchronous systems would have unbounded time complexity