End-to-End Arguments in System Design

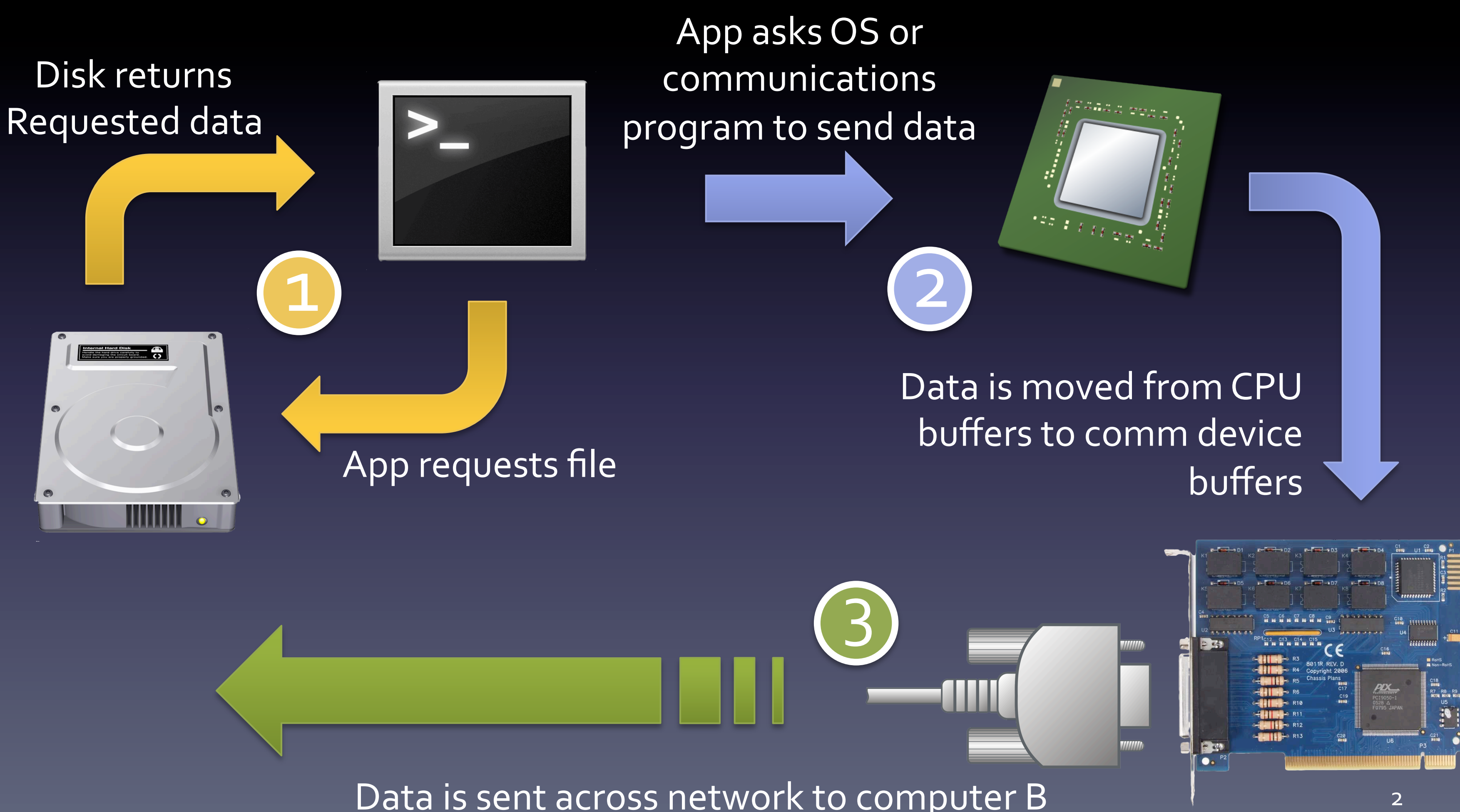
J. H. Saltzer, D. P. Reed, and D. D. Clark

ACM Transactions on Computer Systems, Vol. 2, No. 4, Nov. 1984

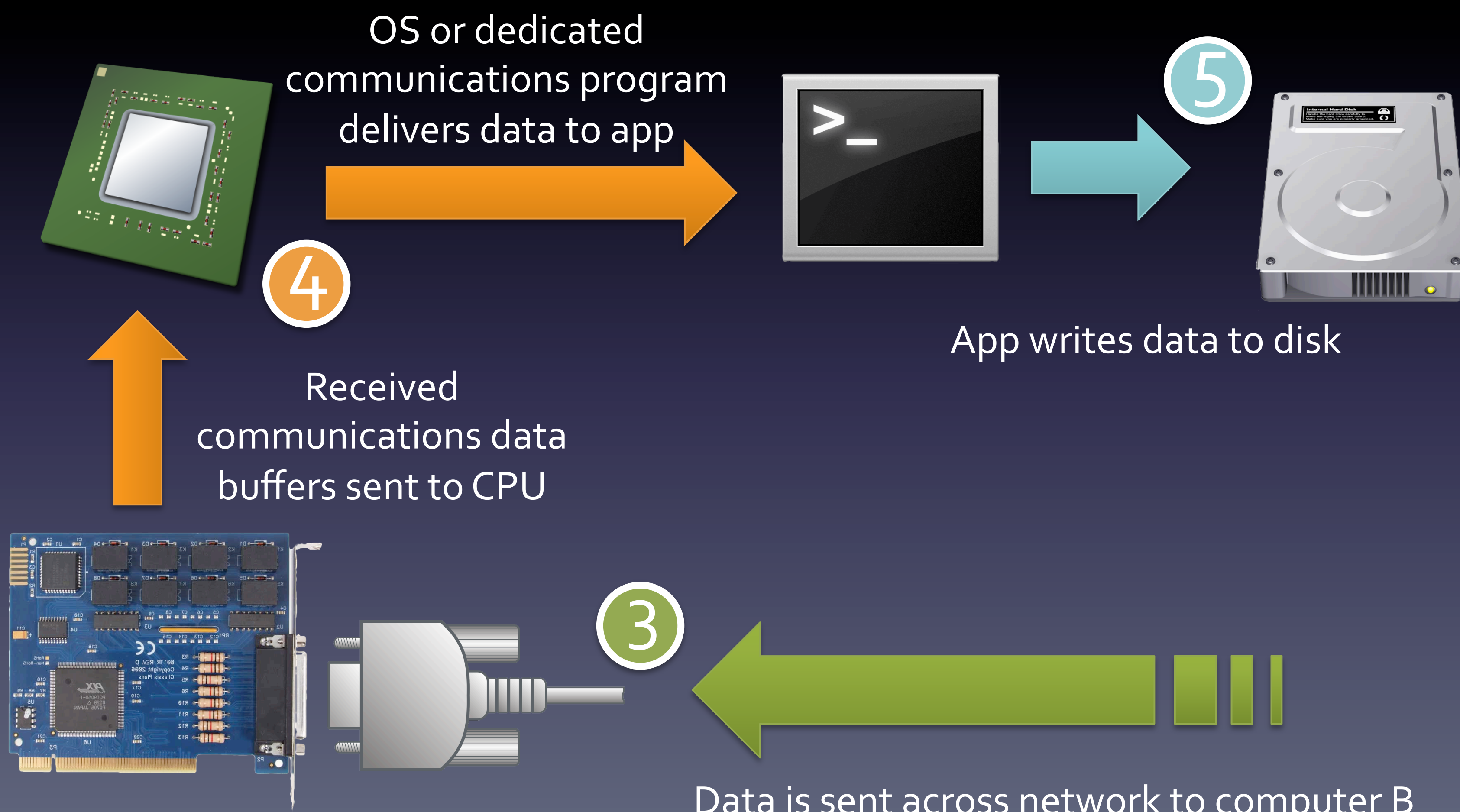
E2EA

- A set of best practices when designing a system
- At any given level in a system, only implement functionality which can be effectively utilized by all higher levels in the system

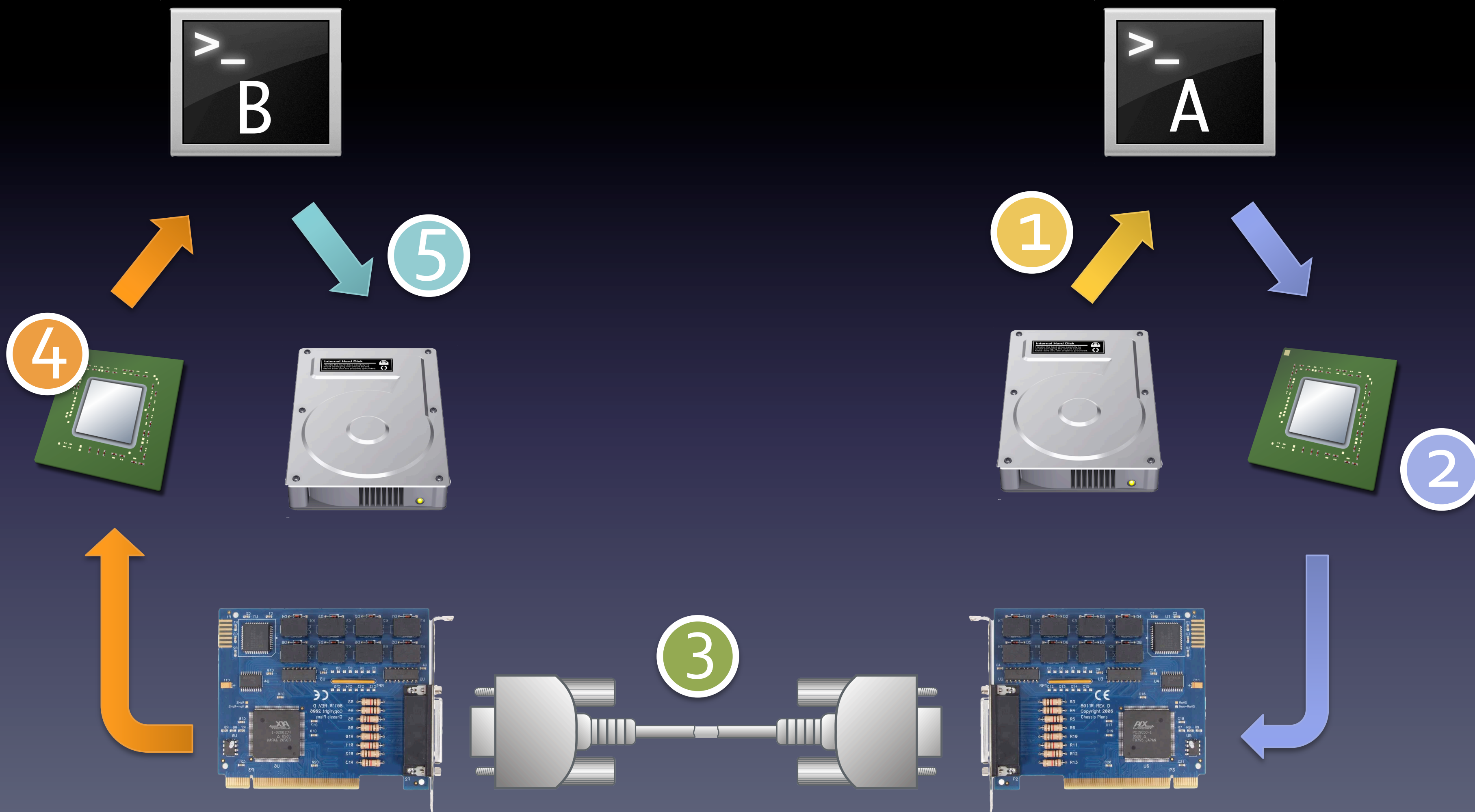
Careful File Transfer (Computer A)



Careful File Transfer (Computer B)



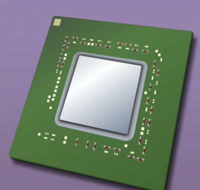
Where can things go wrong?



Where can things go wrong?

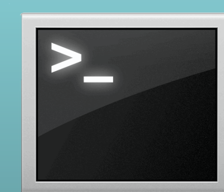
An entire system could crash during the transfer

Hardware faults cause data to be read or written incorrectly

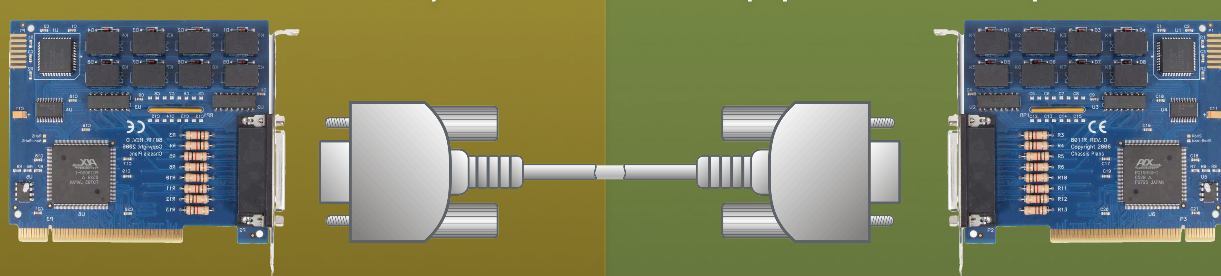


Transient errors in the CPU or RAM subsystems could cause buffers to be corrupted

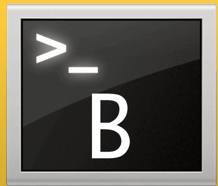
Incorrect logic or other flaws in the OS or file transfer software can corrupt data



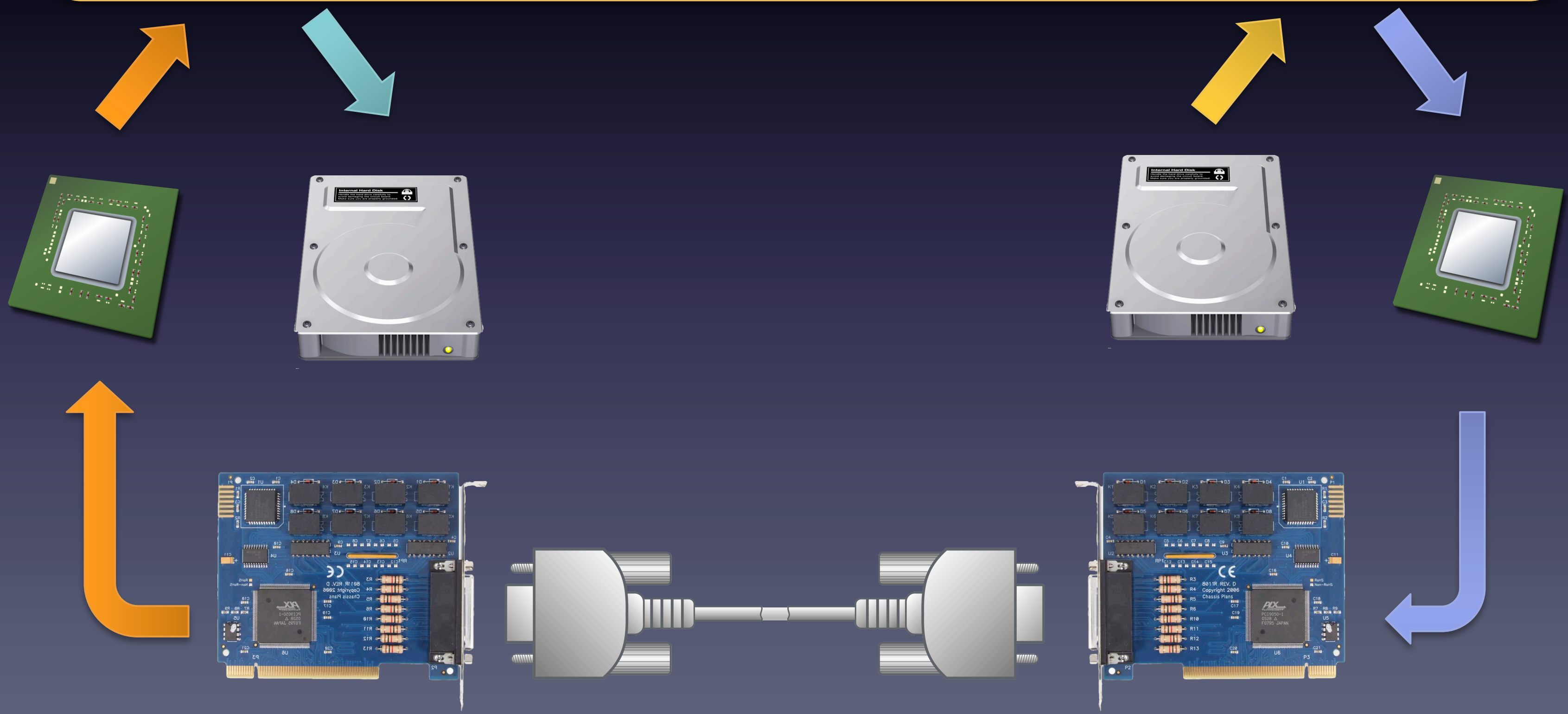
The network subsystem could drop packets or flip bits



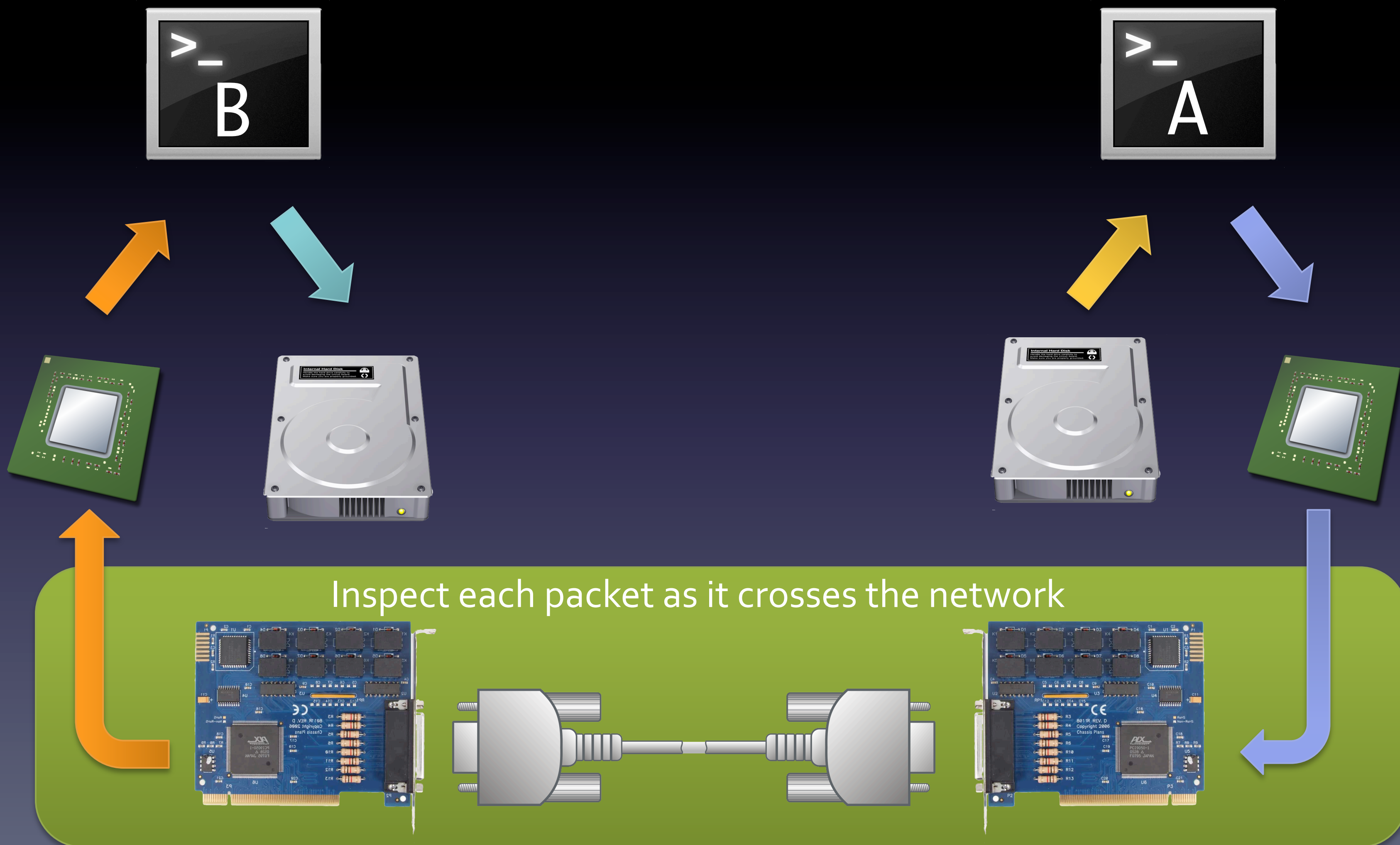
Check at the endpoints



The application endpoint is best suited to verify the data. It knows how the data is used, and how to check that the operation succeeded



Verify data in low level systems



Verifying each packet

- Encapsulate a 20KB file transfer using the XMODEM protocol and transfer at 9,600 bps

XMODEM Packet Structure:

SOH	Frame #	$\overline{\text{Frame \#}}$	Byte 1	Byte 2	• • •	Byte 128	CRC	CRC

Determine total size of data including container

$$20\text{KB} \times \frac{\text{frame}}{128\text{B}} = 160\text{frames}$$

$$160\text{frames} \times \frac{5\text{B}}{\text{frame}} = 800\text{B}$$

$$20\text{KB} + 800\text{B} = 20.78125\text{KB}$$

→

XMODEM transfer time

$$20.78125\text{KB} \times \frac{8\text{b}}{\text{B}} \times \frac{1\text{s}}{9,600\text{b}} = 17.7\overline{3}\text{s}$$

Raw data transfer time

$$20\text{KB} \times \frac{8\text{b}}{\text{B}} \times \frac{1\text{s}}{9,600\text{b}} = 17.0\overline{6}\text{s}$$

→

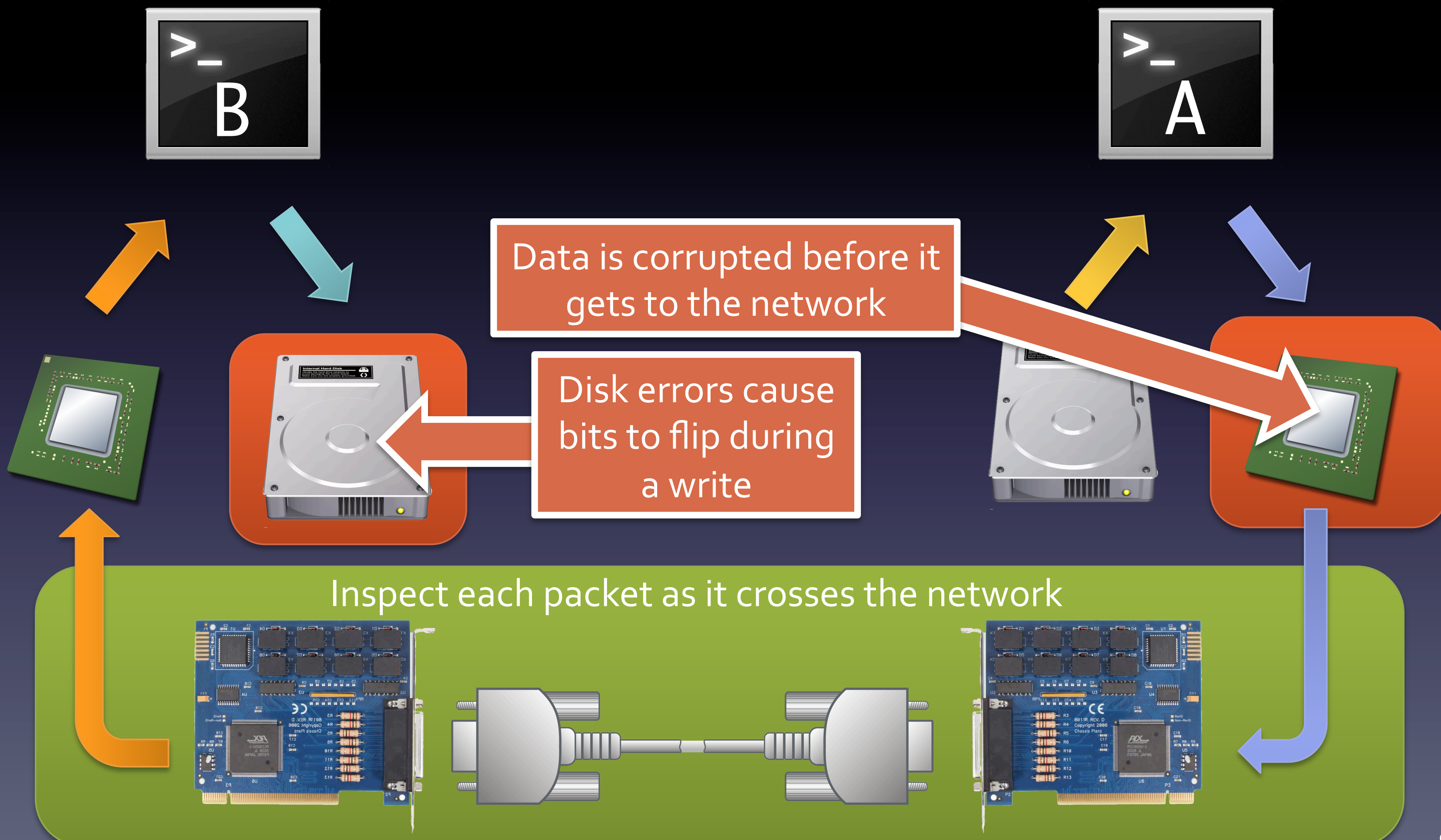
XMODEM Overhead

$$\frac{17.7\overline{3}\text{s}}{17.0\overline{6}\text{s}} - 1 = 0.0390625 \approx$$

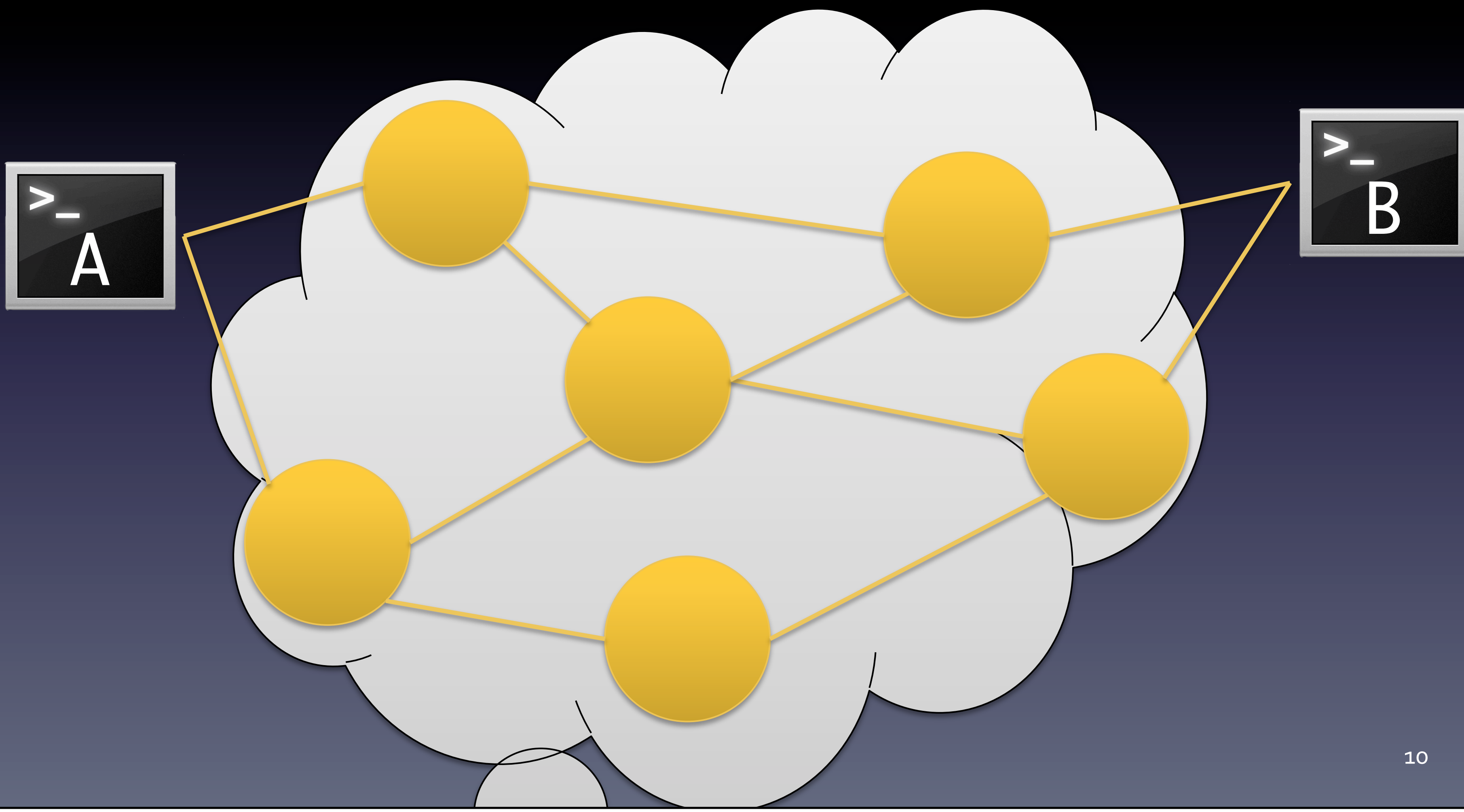
4%

The overhead imposed by checking each packet *seems* modest…

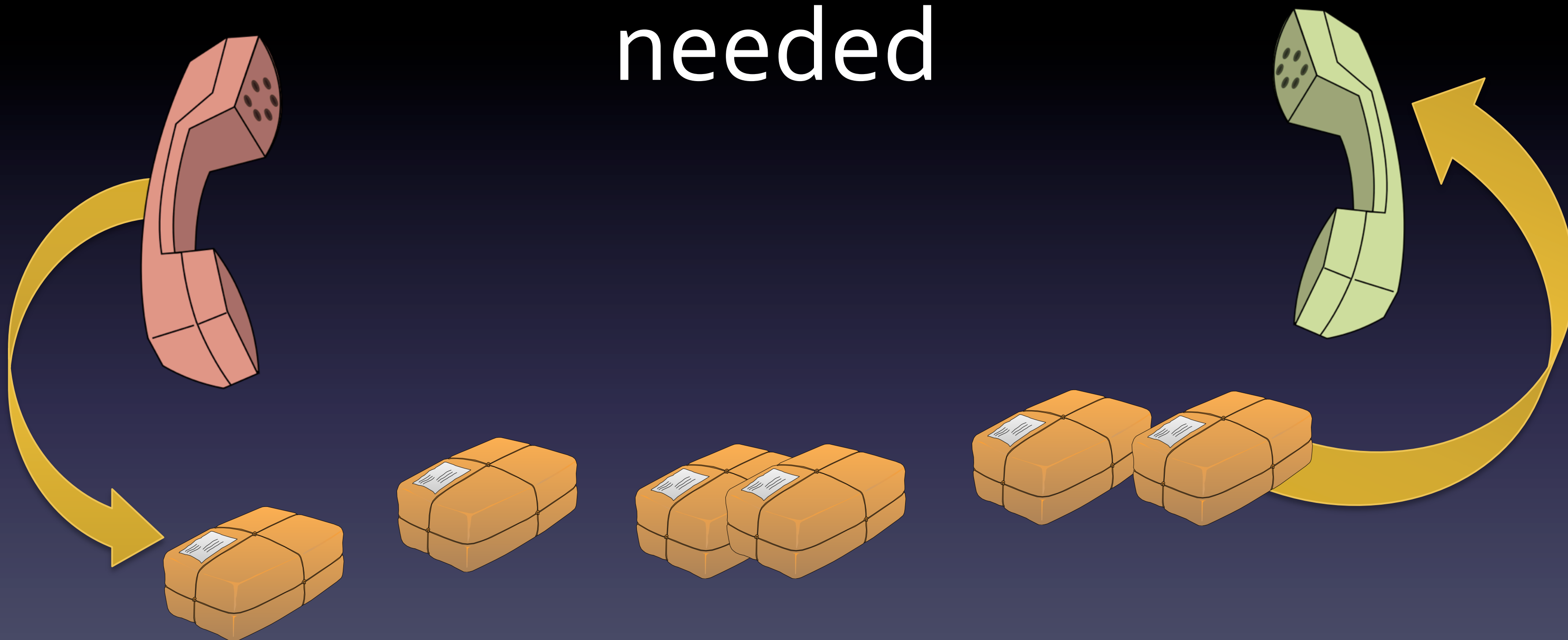
Other errors can defeat packet inspections



Checking packets at each hop



Let the application decide what error checking and recovery is needed

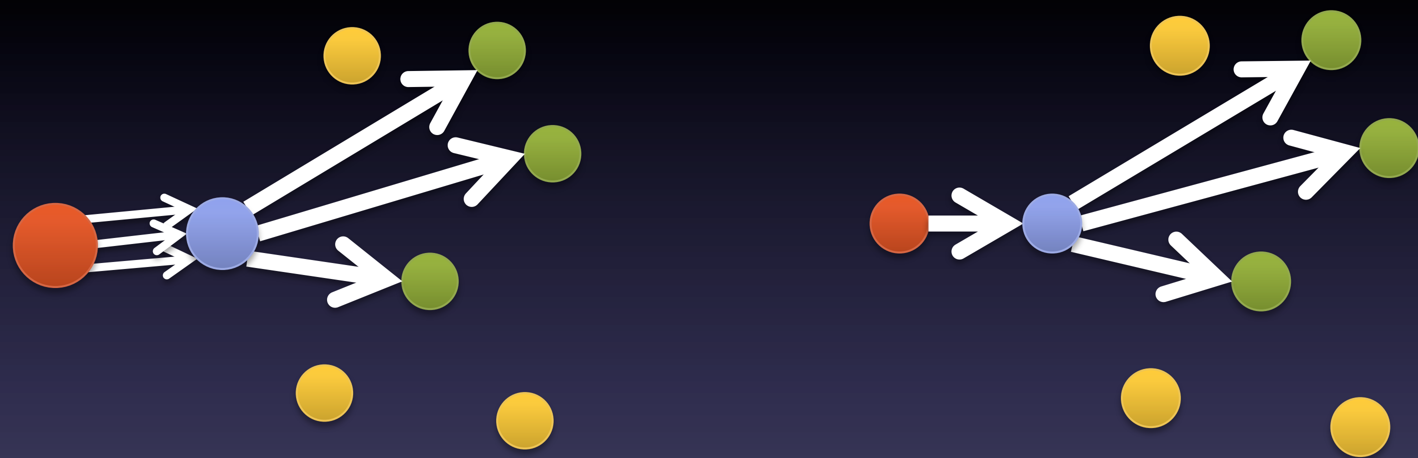


Error checking and recovery is not preferred in *every* situation

What do you think?

Multicast

- Unicast requires a dedicated message for each client
- Can be bandwidth intensive, since identical content may be sent across the network to different users



- Multicast allows users to register to receive messages from a particular source
- Allows the sender to send one message, which is duplicated at a node with multiple interested clients attached

This task cannot be performed in the ends, but clients are not forced to use multicast

Source: T Moore. **A critical review of "End-to-end arguments in system design".** *In IEEE ICC 2002*. vol 2. pp 1214-1219. August 2002.
http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=997043

Sorting Libraries

```
void qsort ( void * base,
             size_t num,
             size_t size,
             int ( * comparator ) ( const void *, const void * ) );
```

- The endpoint is an application which requires sorted data sets
- The sort function doesn't have enough information about how to compare items
  - It's a partial implementation
- The calling application provides the rest of the implementation using a comparator

The sorting library does not restrict callers to use a particular value system, yet provides a library implementation of quicksort

Questions?

Time, Clocks, and the Ordering of Events in a Distributed System

by L. Lamport

CS 5204 Operating Systems

Vladimir Glina

Fall 2005

Overview

- Key Points
- Background
- Partial Ordering
- Extension for Total Ordering
- Further Work
- Key Points Reiteration
- Evaluation
- Discussion

Key Points

1. The “happens before” relation on the system event set
2. The events partial ordering on the base of the relation
3. The distributed algorithm for logical clock synchronization
4. The algorithm extension to the case of total events ordering
5. The algorithm application for physical clock synchronization

Background: Distributed System Features

- Spatially separated processes
- Processes communicate through messages
- Message delays are considerable
- Absence of the single timer leads to synchronization problems
  - Example: totally ordered multicast

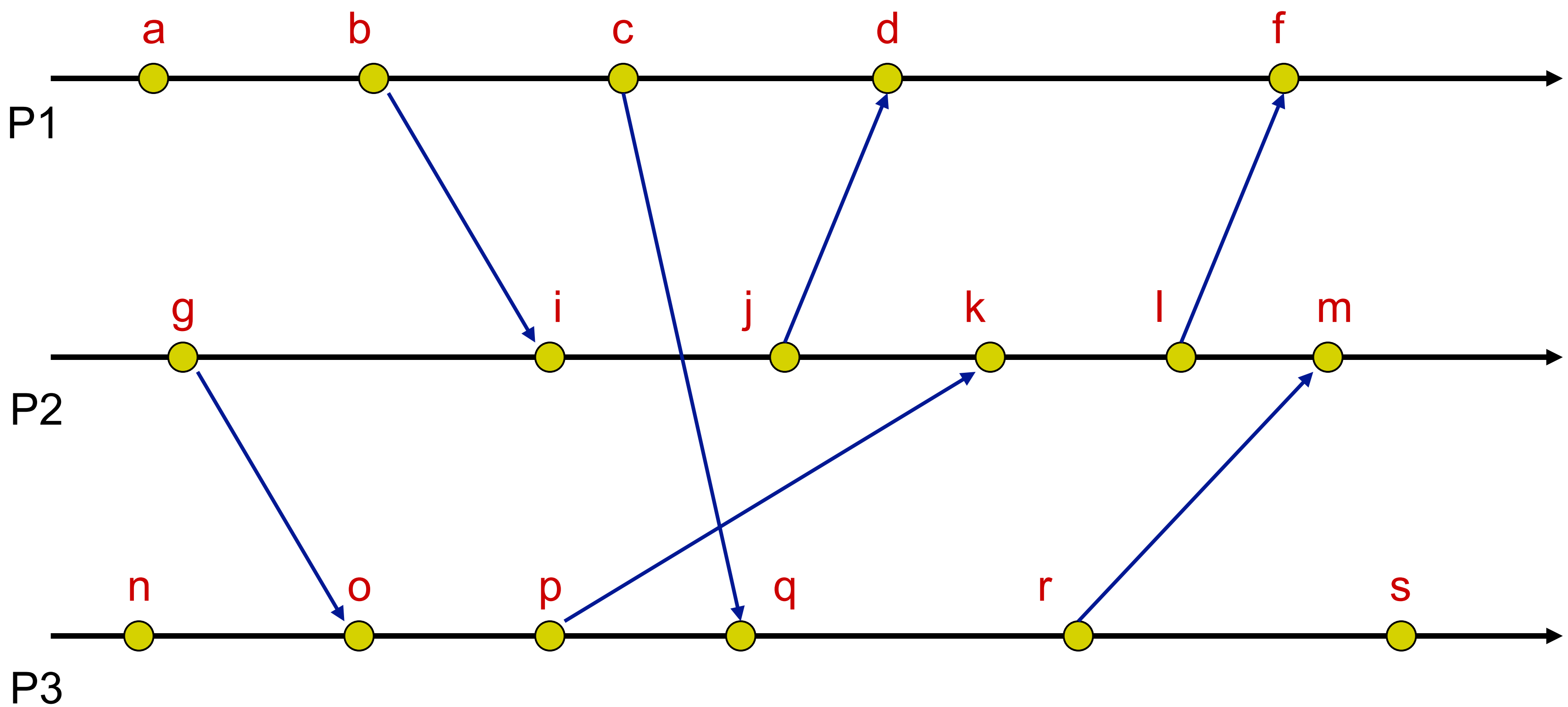
Background: Synchronization Approaches

- Physical Clock Adjustment
  - All clocks show the same actual time
  - Problems:
    - **Most important**: backward time flow possible
    - Sophisticated time services (i.e. WWV); or
    - Reliance on a human operator
- Logical Clock Adjustment
  - Consistency is important, not actual time

Partial Ordering: Basics

- A system is a set of processes P_i
- A process is a set of events *a, b, …*
  with total ordering
- "Happened before" ($\rightarrow$) relation:
  - $(a \in P)$ && $(b \in P)$ && (a comes before b) $\Rightarrow a \rightarrow b$
  - (P_1 sends a to P_2) && (b is the receipt of P_2 for a) $\Rightarrow a \rightarrow b$
  - $(a \rightarrow b)$ && $(b \rightarrow c) \Rightarrow a \rightarrow c$
- $!(a \rightarrow b)$ && $!(b \rightarrow a) \Rightarrow$ *a* and *b* are concurrent
- $!(a \rightarrow a)\ \forall\ a$, so "happened before" is an irreflexive partial ordering on the set of all the system events

Partial Ordering: Example



a → f   b → s   c → m   d || s   i || q   k || r

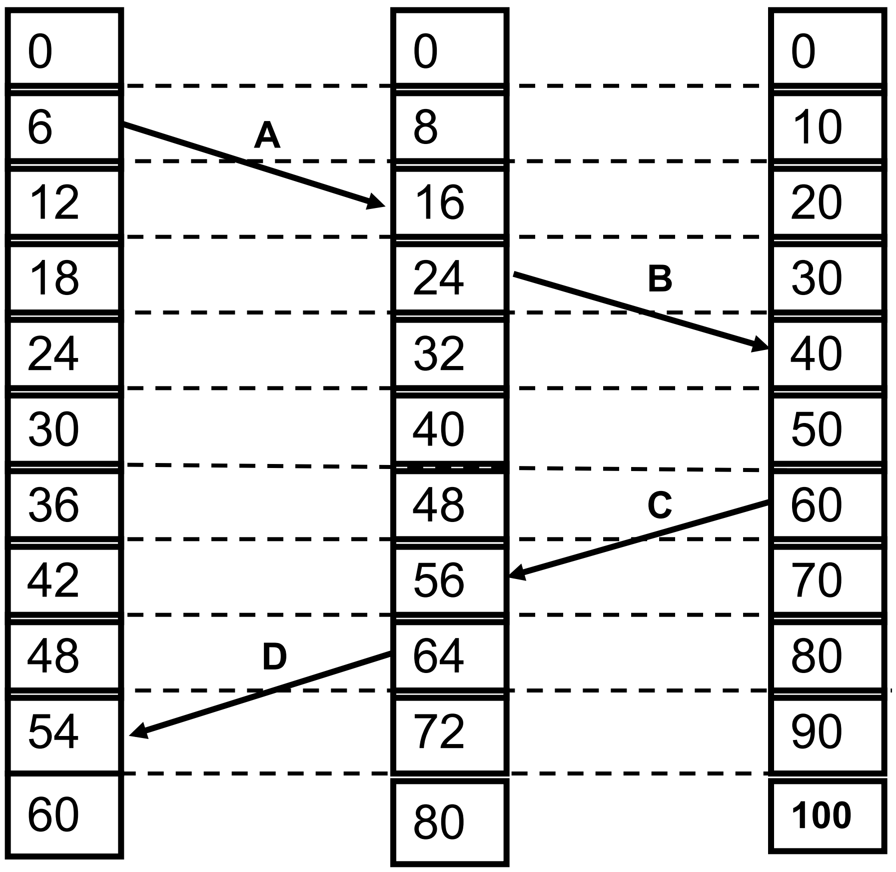
Partial Ordering: Synchronization

- Logical clock: $C\langle a \rangle = C_j\langle a \rangle$ if $a \in P_j$
- ***Check condition:*** for $\forall$ *a, b*
  $a \rightarrow b \Rightarrow C\langle a \rangle < C\langle b \rangle$ **(not vice versa)**
- The check condition is satisfied if
  - **C1.** $(a, b \in P_i)$ && (a comes before b)
    $\Rightarrow C_i\langle a \rangle < C_i\langle b \rangle$
  - **C2.** (P_i sends a to P_j) && (b is the receipt of P_j *to* a)
    $\Rightarrow C_i\langle a \rangle < C_j\langle b \rangle$
- **C never decreases!**

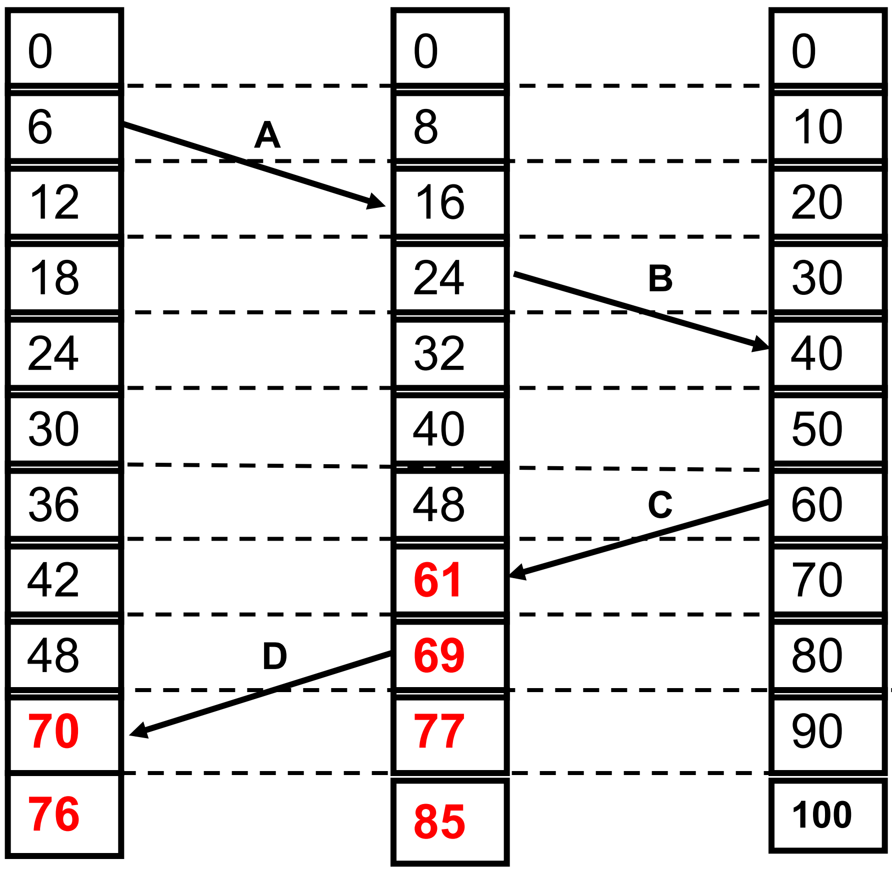
Partial Ordering: Implementation Rules

- **IR1.** Each P_i increments C_i between any two successive events.
- **IR2.**
  a) If *a* is the sending of a message *m* by P_i, then m contains a timestamp $T_m = C_j\langle a\rangle$; **and**
  b) Upon receiving *m*, P_i sets C_j greater than or equal to its present value and greater than T_m

Partial Ordering: Unregulated Clocks



Partial Ordering: Corrected Clocks



Total Ordering: Definition

- ‹ is an **arbitrary** total ordering of processes
- "Happen before" for total ordering ($\Longrightarrow$):
  $(a \in P_i)$ && $(b \in P_j) \Rightarrow a \Longrightarrow b$ iff
  - $C_i\langle a\rangle < C_j\langle b\rangle$, ***or***
  - P_i ‹ P_j
- The total ordering depends on C_i and is not unique

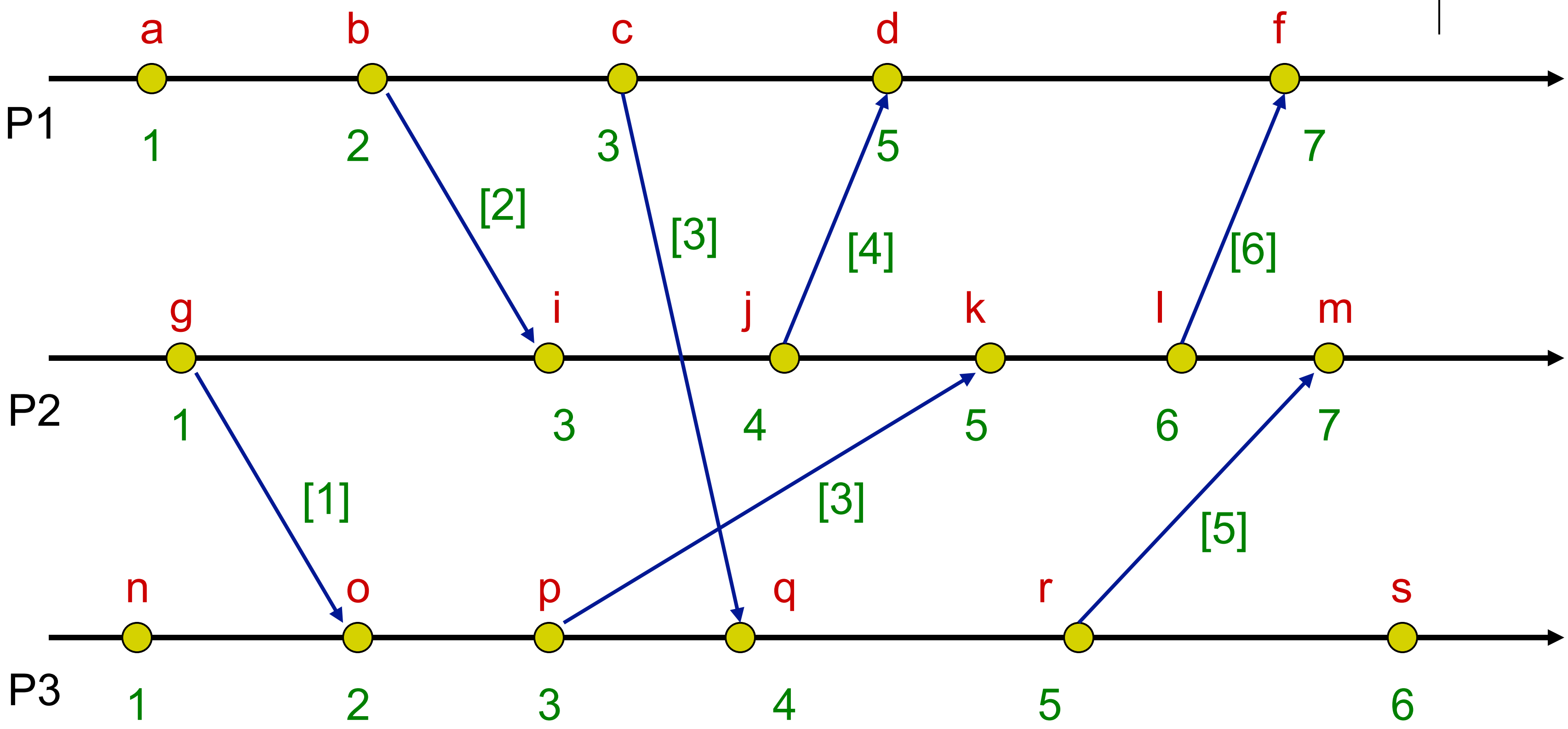
Total Ordering: Synchronization

1. P_i broadcasts the message $T_m:P_i$ (request resource) and puts it on its request queue.
2. When P_j receives $T_m:P_i$, it puts the message on its request queue and sends the acknowledgment to P_i.
3. To release the resource, P_i removes $T_m:P_i$ from its queue, broadcasts a timestamped release message.
4. When P_j receives the release message, it removes $T_m:P_i$ from its queue.
5. P_i is granted the resource when
   1) It has $T_m:P_i$ in its queue ordered before any other request in the queue by the relation $\Longrightarrow$; and
   2) P_i has received a message from every other process timestamped later than T_m.

Further Work: Vector Timestamps

- Lamport clock is:
  - Consistent: $a \rightarrow b \Rightarrow C\langle a\rangle < C\langle b\rangle$
  - **Not**: $C\langle a\rangle < C\langle b\rangle \Leftrightarrow a \rightarrow b$ (not strongly consistent)
- Vector timestamps (VT) are strongly consistent
- VT address **potential** causality
  - Allow to say if *a* **happened** before *b*, but not if *a* **caused** *b*
- VT say how many events have occurred so far at all processes
- VT solve the totally-ordered multicasting problem

Lack of Strong Consistency

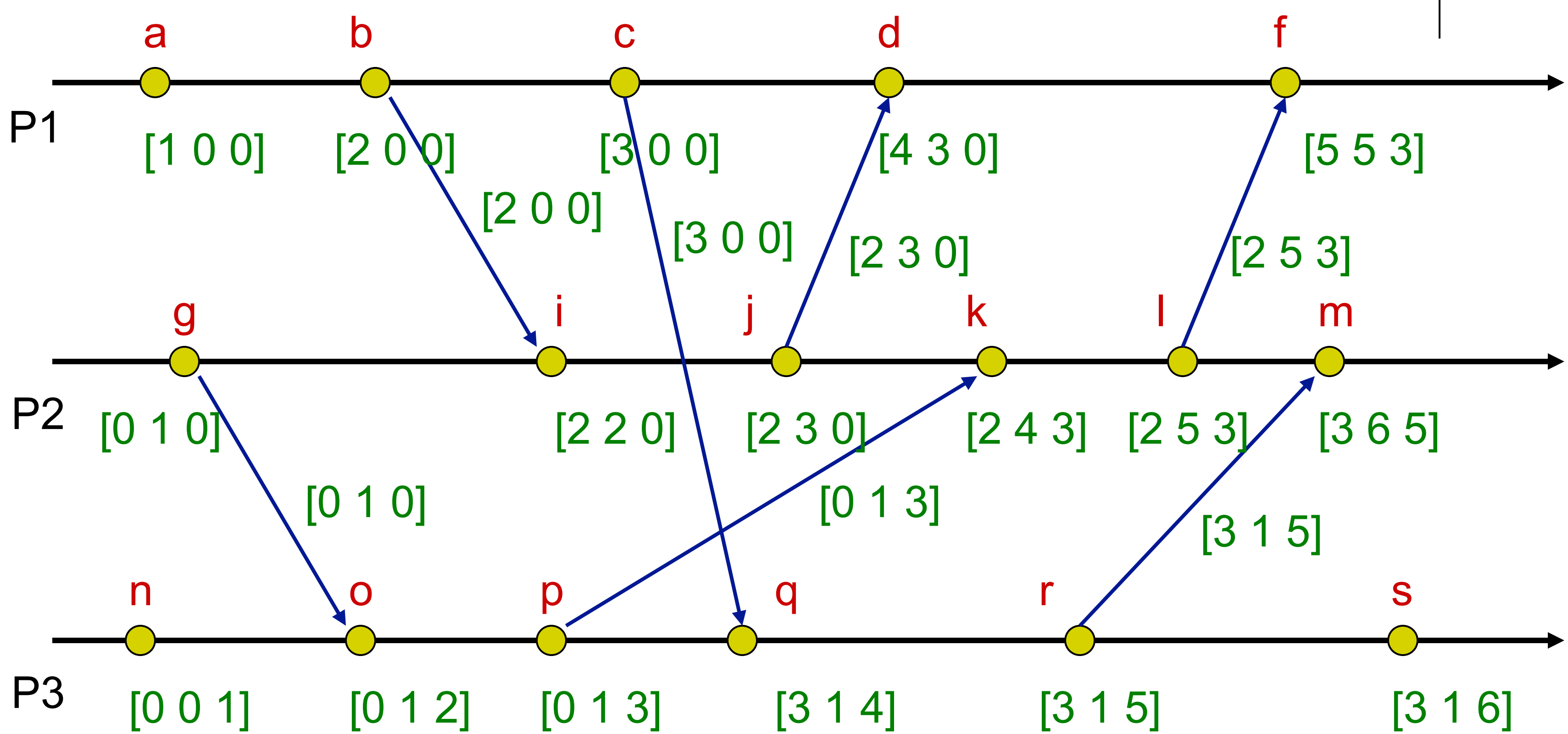


d || s
5 < 6

q || i
4 > 3

k || r
5 = 5

Vector Clocks (1)

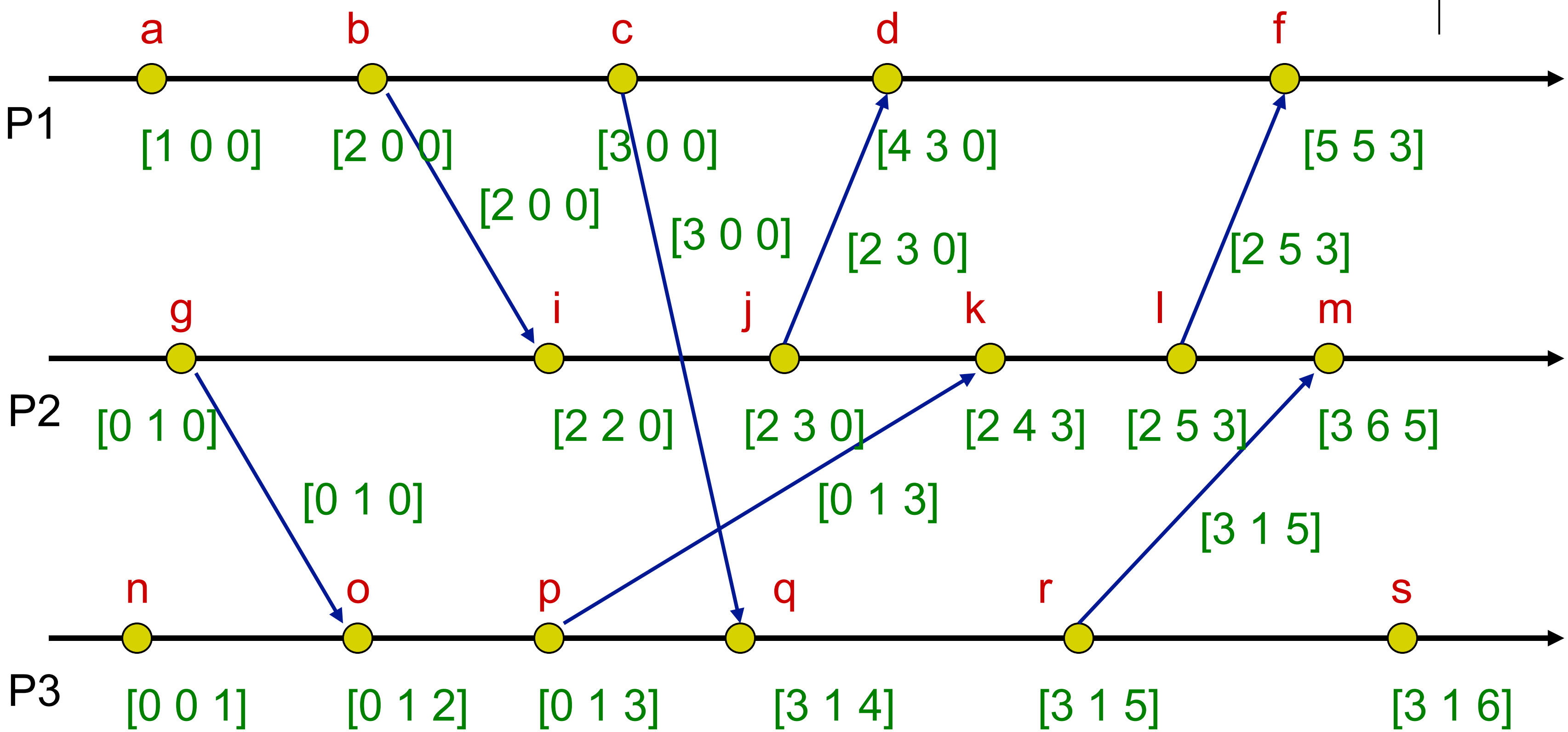


a → f
[1 0 0] < [5 5 3]

b → s
[2 0 0] < [3 1 6]

c → m
[3 0 0] < [3 6 5]

Vector Clocks (2)



d || s
[4 3 0] < [3 1 6]

q || i
[3 1 4] < [2 2 0]

k || r
[2 4 3] < [3 1 5]

Key Points Reiteration

1. The "happens before" relation on the system event set
2. The events partial ordering on the base of the relation
3. The distributed algorithm for logical clock synchronization
4. The algorithm extension to a case of total events ordering
5. The algorithm application for physical clock synchronization

Evaluation

- The logical clocks idea is very appealing
- Virtually no revision on previous work
- Nice to have more mathematically strict extension on total ordering, if possible

Discussion

Thank you!

Any questions?