Fabrizio Furano: "From IO-less to Networks"

- The most common pitfall: "Everything is just the same"
  - A little provocation
- Optimizing code is good, often not enough
- Latency
  - Latency in local IO
  - Path of the data
  - Latency in networked IO
  - Path of the data
- Techniques to higher the I/O performance

Focus of the lecture

- The mid-high level aspects of I/O
  - As seen by a competent user (= programmer of the final app)
- Generic issues with I/O
  - Which are much heavier with networks
- Then switch to networked I/O
  - Why is it particular

- We will not treat the low-level specifics
  - E.g. system calls for TCP/IP, exotic flags, etc.
  - 99% of the times these are encapsulated by some product which we use to exchange data

- The focus is how to write generic applications which will perform well
  - By using some sane encapsulation of the TCP-related things
  - And becoming able to see what's really wrong to do

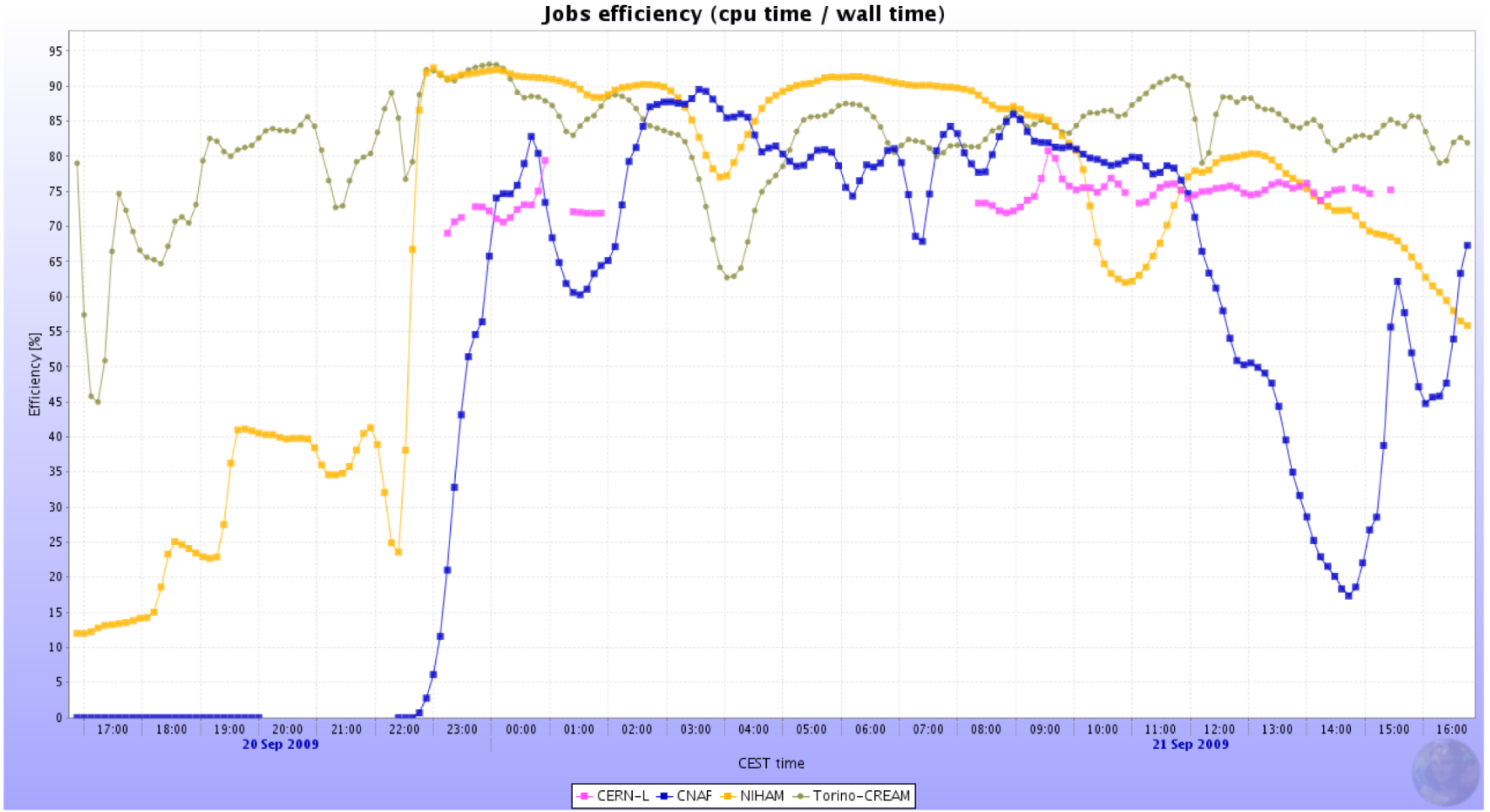
A little provocation

- Suppose you are now a “Jedi code writer”
  - And your code is optimized to perfection
    - Multicore – SSEx etc
  - Will it run as fast as the CPU(s) can?
    - It depends on what it does
  - Most of the HEP tasks can be classified as “I/O intensive”
    - The softwares read and write a lot of data
    - No processing can take place without input
    - Also, some output has to written

A little provocation

- There is really the chance that your application wastes more time in waiting than in computing
  - Simple measure of it: the CPU time / Wall time
  - Let's have a look at a simple real case
    - Which is absolutely good, but still inspiring.
    - It's very difficult to perform better at a large scale.

A little provocation



A little provocation

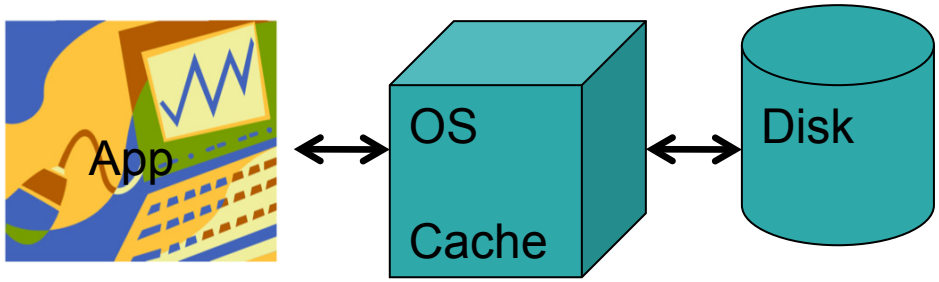
- A small selection of ALICE sites
- The ALICE computing by now is very efficient
  - This means that other's situations can be much worse
  - We see nice peaks of 90-95% efficiency
  - But also lower values
- A delusion?
  - We spent months optimizing the code and now it runs at 70-80% in average
  - Don't be frustrated enough, with other apps and other environments you may find 40% and even much less
    - Just look around

What can we do

- First: “Be prepared”
  - In order not to get frustrated
- Know very well the specifics of I/O performance
  - Throughput, latency and transaction rate
  - Disk I/O and networked I/O as well
- Design your app with this in mind
  - Or be prepared to (painfully) fix it

What's behind the scenes

- An application doing simple I/O (r/w)
- Your favorite OS
- Your local disk



Better to start from this extreme simplification, just to see where we are going

The application

- We can consider it as a producer of requests
  - File opens
  - And then a sequence of couples
    - (offset, length) [+data if it wants to write]
    - For each file it accesses
  - Every request asks the storage to do something
    - Typically it's composed by disks
    - It can be reading or writing
  - And typically waits for the response because it needs it to proceed with the computation

The disk in 30 seconds

- We know how a disk works
  - See the previous lecture by A.Hanushevsky
  - Some time is spent to find the data
  - Some time is spent to send (or write)
- It can be an ultra-fast Solid State Disk
- Or a cheap floppy disk
- It will always work like that
  - The difference is in how these times are related to the computing phase of the app

Reading and writing

- In general, reading data is a bit harder than writing data
  - In a simple case (rules of thumb):
    - Write case: the app has to produce buffers as fast as it can
      - The bigger the buffer, the faster it will be
      - We can also accumulate many buffers and flush them later (delayed write, very well known also for USB pendrives)
    - Read case: if the app needs a chunk of data, it will wait until it has come. No option for the ingenuous programmer.

A key factor: the OS cache

- Modern OSs are very smart
  - For each read request they remember the result
  - They also remember what's in the proximity
- So, they can just fool us
  - By making us believe that our sw is very efficient
    - Just because we execute it more than once in our disk
    - Let's have an inspiring quick try
- Believing that our app is efficient (while it is not) is the first big mistake we can make

An example: the forward reader

```
int f = open("/tmp/bigfile.dat", O_RDONLY);
 if (f < 0) {
  printf("Error: %s", strerror(errno));
  exit(-1);
 }

 long long offs = 0;
 struct stat st;
 if (fstat(f, &st)) {
  printf("Error: %s", strerror(errno));
  exit(-1);
 }

 long long filelen = st.st_size;
 printf("File length: %lld\n", filelen);
 char buf[1024];
 while (offs < filelen) {
   int n = pread(f, &buf, 1024, offs);
   if (n <= 0)  {
  printf("Error: %s", strerror(errno));
  exit(-1);
   }

    offs += (10240-1024);
 }
```

An example: the forward reader

- It reads 1KB every 10KB
- For a 2GB file it reads 200MB
  - Somebody might think that the disk is able to get 20-50MB/s, hence it should take 10s
  - Instead it takes 5 minutes (in my machine)
  - It takes a few seconds only at the second run (and not always)

An example: the forward reader

- It runs very fast, yes… apparently
- Unless you clear the OS cache with the given tool ‘clearcache’
  - And then it has (in my laptop) an efficiency of 4%
  - To make it fast (the second time) the OS uses a lot of memory
    - The OS uses for that the unallocated memory
    - It can cache gigabytes in common hardware
  - If there are many users or the app consumes it all, it will not run so fast
  - This program is very inefficient, but you might think it’s not
  - Conclusion (for now): don’t be fooled

Even worse

- In the real life your app is almost never alone
- Disk manufacturers declare “very” low times to execute a (read) transaction
  - But often between two transactions there are many others (from other users/processes)

Let’s suppose that it takes 5ms for a disk to pick up the requested chunk

In those 5ms the disk bus could read AT LEAST 5ms*266Mbit = >1GByte
Instead it does nothing.
In those 5ms a cheap hard disk could read 5ms*50MB/s = 250KB
Instead our test app is idle in order to read 1KB each time

Another example: the backward reader

- We can be amazed by how many things in OSs and disk hardware/firmware are optimized for increasing offsets
  - What happens if we read the same data chunks backwards?
  - The performance (with and without) OS caching gets much lower. Try!
  - Conclusion (for now): this is one more way to make our tiny program even more inefficient.
    - At least, we are starting wondering some of the things to avoid, and the true goal of this lecture

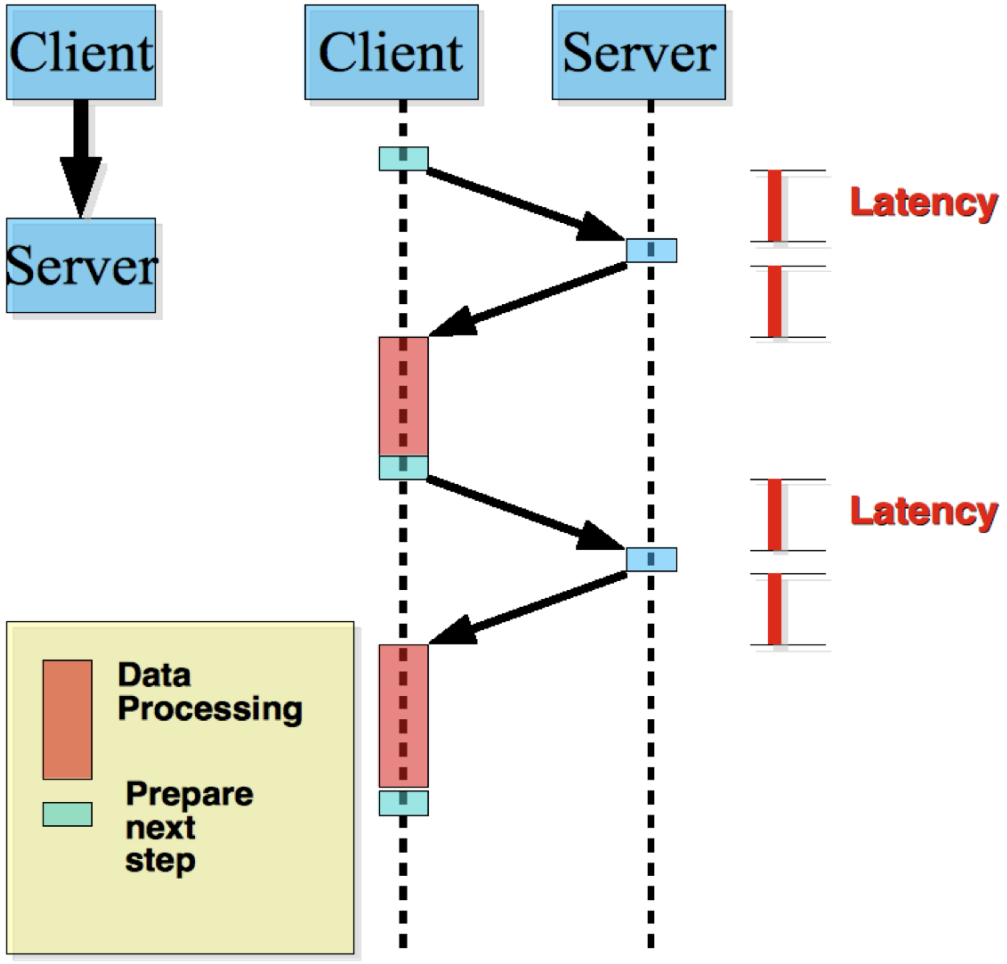
Conclusion (for now)

- There are several ways to do the same I/O operations
  - We are focusing on reads: often more ‘difficult’ than writes
  - E.g. getting some chunks of data to process
  - Some ways are much more efficient than others
  - To avoid having highly inefficient applications we have to:
    - Know very well the details of the technology (hw and sw) we use
    - Exploit it in order to always choose the best opportunities
      - And know the possibilities we have
  - Hence, some important choices are:
    - The sequence of the operations (e.g. the lengths@offsets to read)
    - The moment in which a request is issued.

The enemy: Latency

- Simply speaking: The time it takes to get the response to a data request
  - More precisely: the time it takes to start getting it
  - Typically the throughput is very high
- It is present and measurable also in our very simple examples
- It can DOMINATE your computation also in simple cases (e.g. 1 app, 1 user with 1 disk)
  - Like the forward/backward reader
  - It will do it much more in the multiprocess case
- In all the cases we saw, we never reached the maximum available data throughput from disk
  - So, for now that's not a problem, we don't need a faster disk
  - It may become later, but that's problem #2

Latency: A sequence graph



Latency

- “Latency” can be anything which makes the client wait
  - Network latency
  - Time to move the disks heads
  - Server congestion
    - Which makes it process the requests slowly
  - <put your reason here>

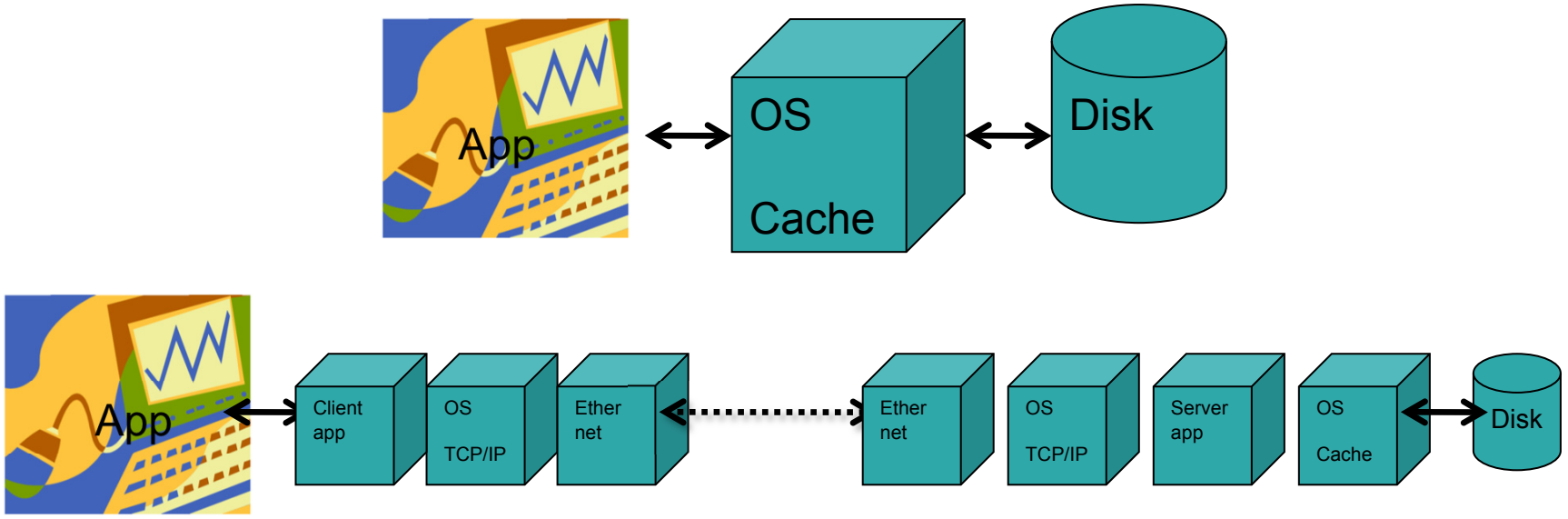
Closer to HEP apps. An example.

- An app can read 1M chunks of 2KB each
  - 2GByte, a typical HEP file
  - If it takes 1ms of latency per chunk (highly optimistic!) the app will do nothing for 1000 seconds (~20min)
  - If, in average, the app computes 1ms per read chunk the efficiency will be ~50%
    - Very common case
    - Remember that we are still speaking of local disks
      - **Supposed** to be the easy case
      - Instead we might have been just fooled by the OS
    - This makes us not very confident in our super-optimized application anymore
      - Again, that was NOT wasted time

Where's the network?

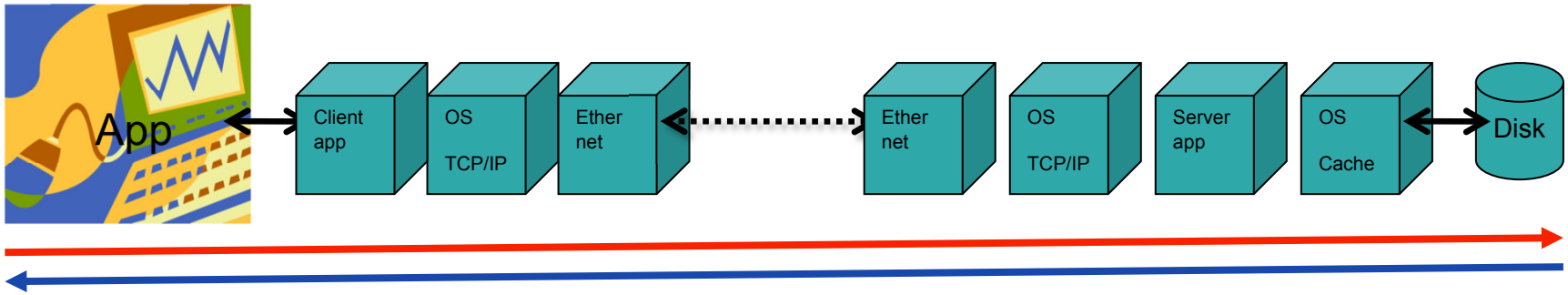
- We are able to successfully measure the (in)efficiency of our app even in a simple case, i.e. a local disk.
- With networks the things can become more problematic
  - Because latency plays an even greater role
  - Because we do not have the hope that a new technology will save us. With networks we are fighting against the speed of light.
    - New technologies will higher the throughput, but here we saw that the worst enemy is latency
- Let's see why and how.
  - This will give us a basic additional insight. Later on, we will look at a few techniques to reduce such a heavy negative impact
    - And get our performance

The data flows: local and network



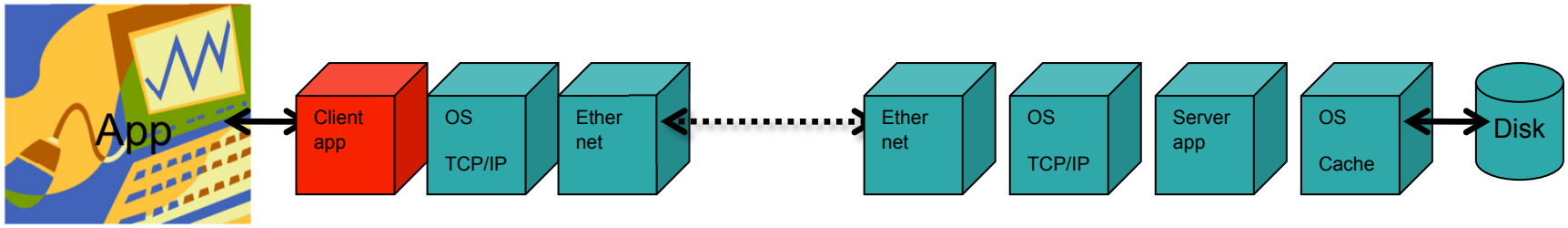
- Just with a quick look:
  - Many steps = Many places for latency
  - Still space for low throughputs as well
  - Some of the steps can be really problematic
  - Here, we suppose that the software quality is at its best
    - Which, unfortunately, often is not the case

Networked case: information flow



- Potentially every step can cause a slowdown
- The base mechanism in the app is still the same
  - Send request + get response
  - Both request and response follow the same steps (reversed)
  - Need to know the characteristics of each step
    - To have an idea about its impact
    - With respect to the used technologies
    - Let's have a deeper look

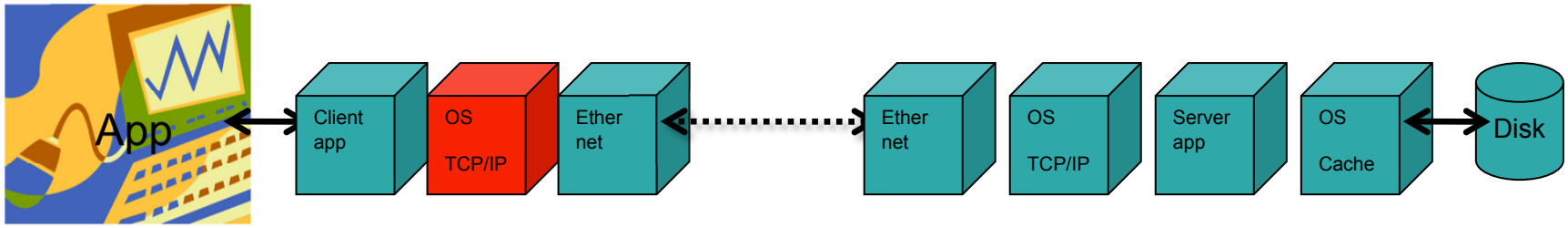
Networked case: information flow



- The application asks for a chunk to read
- It asks for it to its client
  - Can be the client of NFS/AFS/GPFS/XROOTD/ RFIO/ORACLE/MYSQL etc.
  - Common clients immediately forward it (but not necessarily) by invoking the proper OS primitives

Possible source of slowdowns (beside a slow app):
•The client takes too much to translate the request and to pass it forward

Networked case: information flow



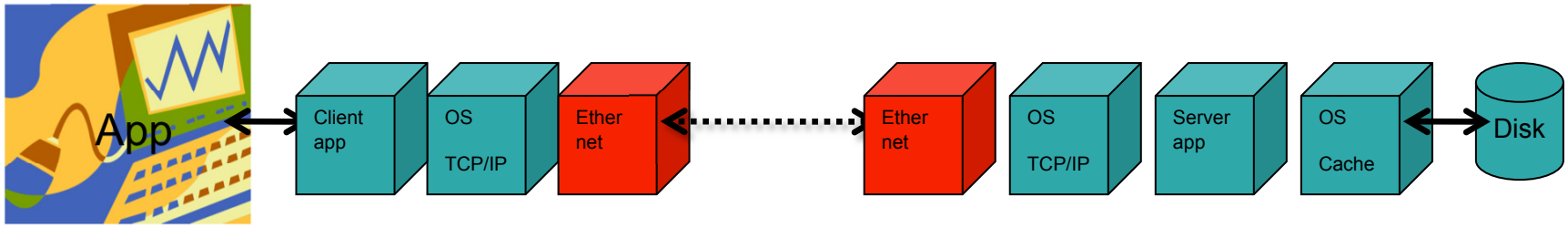
- The OS forwards the request's parts to the network part
  - Depending on the settings, this can be delayed
    - Ref: the TCP_NODELAY socket option for example
  - The quality of the implementation of such client may be important
    - But typically it is not up to the user… good and bad clients come bundled with something else.

Possible source of slowdowns:

•A single request can be unreasonably big or demanding for the OS to treat it

•The client app translates simple requests into super-complicated interactions

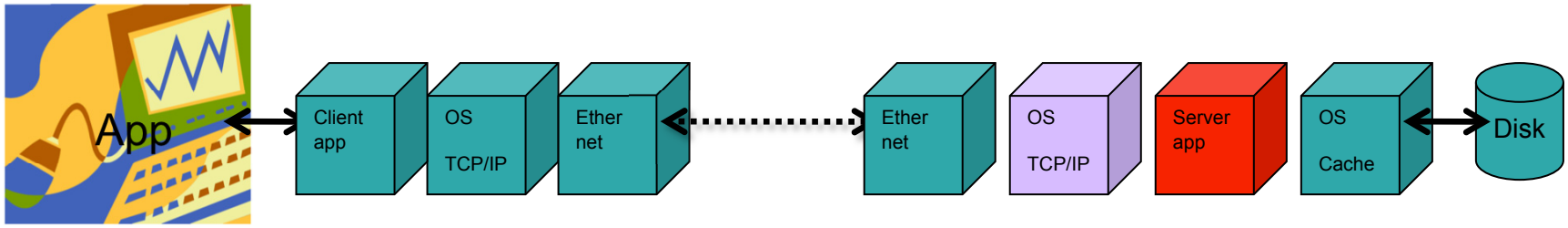
Networked case: information flow



- The network can be:
  - High latency by itself (e.g. 0.1ms for a LAN up to a 300ms RTT WAN)
  - Too slow (takes time just to send the bytes composing the request)
  - Very loaded (collisions make the OS/Ethernet retry/wait)

Possible source of slowdowns:
•Size of the data or number of chunks composing the request
•Characteristics of the network
•The latency here can be EXTREMELY variable (0.1ms -> 150ms)

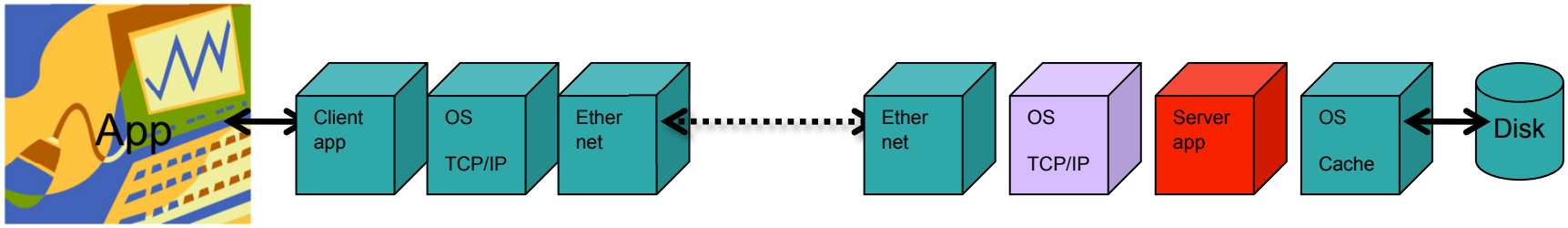
Networked case: information flow



- The OS can be:
  - Incorrectly tuned (happens very often for WANs)
  - Can suffer from the excessive fragmentation of the request

Possible source of slowdowns:
•Latency+throughput: typically bad TCP settings in the OS

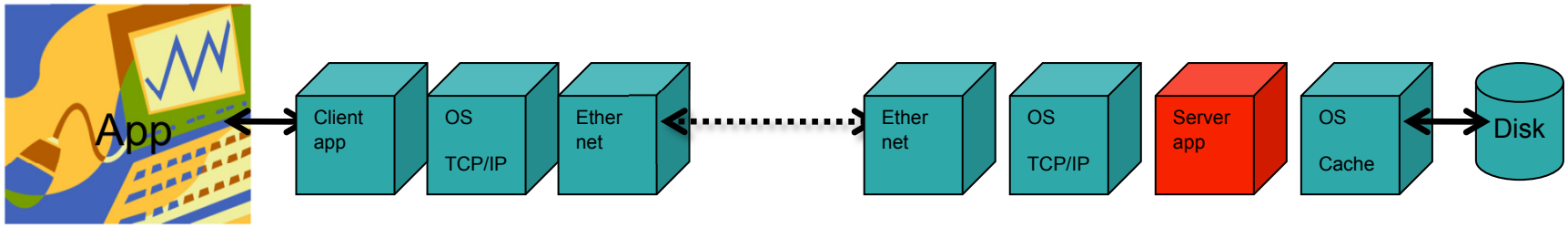
Networked case: information flow



- The server (in the TCP side) can be a problem.
  - The software quality here plays a major role and offers a very broad range of inefficiencies
  - These can be linked with latency, throughput and stability
    - Yes, restarting everything manually is a form of latency as well!

Possible source of slowdowns:
•Ingenuous, scholastic programming in the server

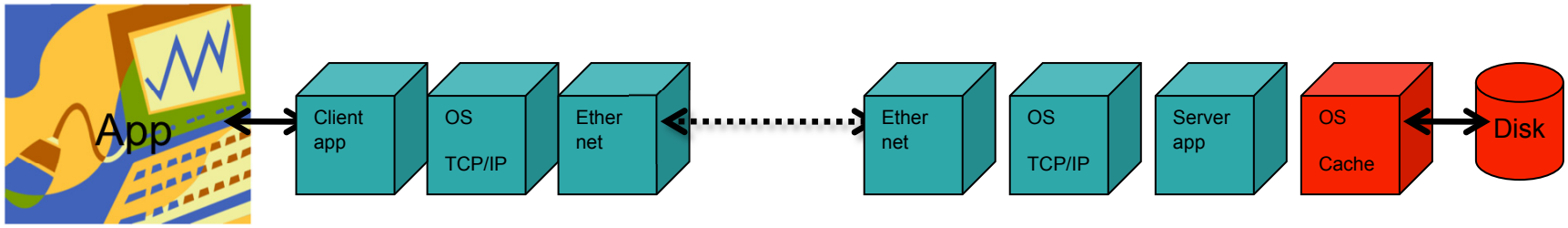
Networked case: information flow



- When there is a server we are never alone using it
  - There can be many other connections ($O(10^3)$)
  - Not all the implementations/products are equal
    - In fact we are surrounded by so many of them
  - Also here software quality makes the difference

Possible source of slowdowns:
•Ingenuous, scholastic programming in the server

Networked case: information flow



- OS+Disks: we already spoke about this
  - Taken alone it can cause unacceptable inefficiencies in the data flow
  - They are just one among others now…

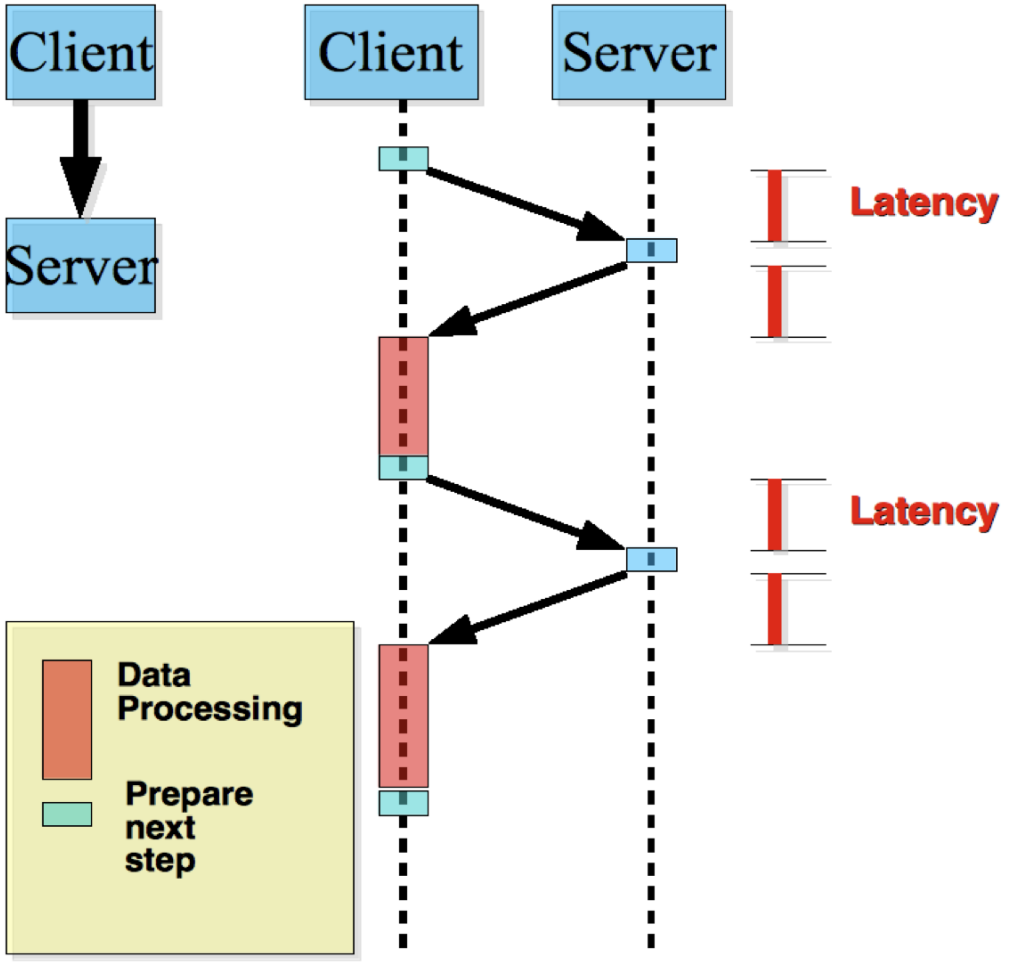
Possible source of slowdowns:
The stream of requests is not compatible with disk access, which becomes very inefficient

Let's simplify

- From this point on we suppose that:
  - All the softwares of the blocks in the prev. slide are "perfect software"
  - Perfectly tuned
  - No slowdowns due to poor sw quality
- Note that this is a very strong assumption
  - We already have enough troubles
  - Anyway we'll have a better insight useful to spot bad softwares in the future

Latency again

- This time we know that the latency may be a sum of many things
  - The result does not change, this is what the client (and the app) see



What to do?

- We don't have to get depressed
- We don't have to stop (or avoid) optimizing our code
- There are some ways to deal with latency
  - They must be implemented in the data access framework (e.g. ROOT+xrootd)
  - But the app must be a bit aware of that
    - In order to exploit them
  - The programmers must know how it works
- In the next part we will explore what's possible
  - And how it works