
Hide latencies, increase throughput

Exploiting some
new features in the xrootd data access system

Fabrizio Furano

INFN – Istituto Nazionale di Fisica Nucleare
furano@pd.infn.it

Andrew Hanushevsky

SLAC – Stanford Linear Accelerator Center
abh@slac.stanford.edu

Motivation

- The typical way in which HEP data is processed is (or can be) often known in advance
 - Fast and scalable server side
 - Makes things run quite smooth
 - Gives room for improvement at the client side
 - About WHEN transferring the data
 - There might be better moments to trigger a chunk xfer
 - with respect to the moment it is needed
 - Better if the app has not to pause while it receives data
 - Many chunks together
 - Also raise the throughput
-

A simple job

- A simplification of the problem
 - A single-source job
 - Processes data “sequentially”
 - i.e, the sequentiality is in the basic idea, but not necessarily in the produced data requests
 - e.g. scanning a ROOT persistent data structure

```
while not finished
  d = read(next_data_piece)
  process_data(d)
end
```

Simplified schema

We can make some simple hypotheses.

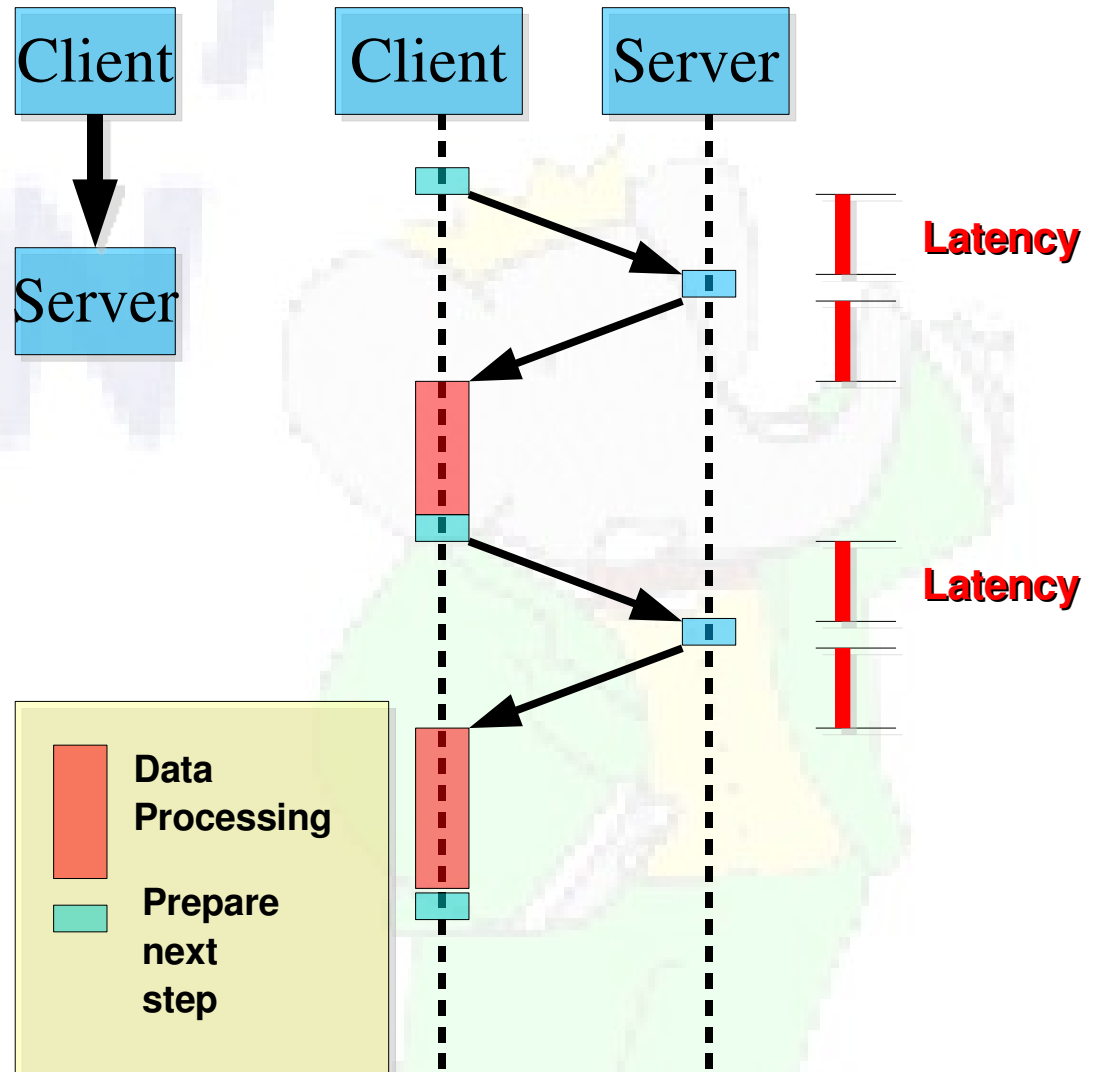
Hypothesis: the data are not local to the client machine

Hypothesis: there is a server which provides data chunks

1 client

1 server

A job consisting of several interactions



Latency

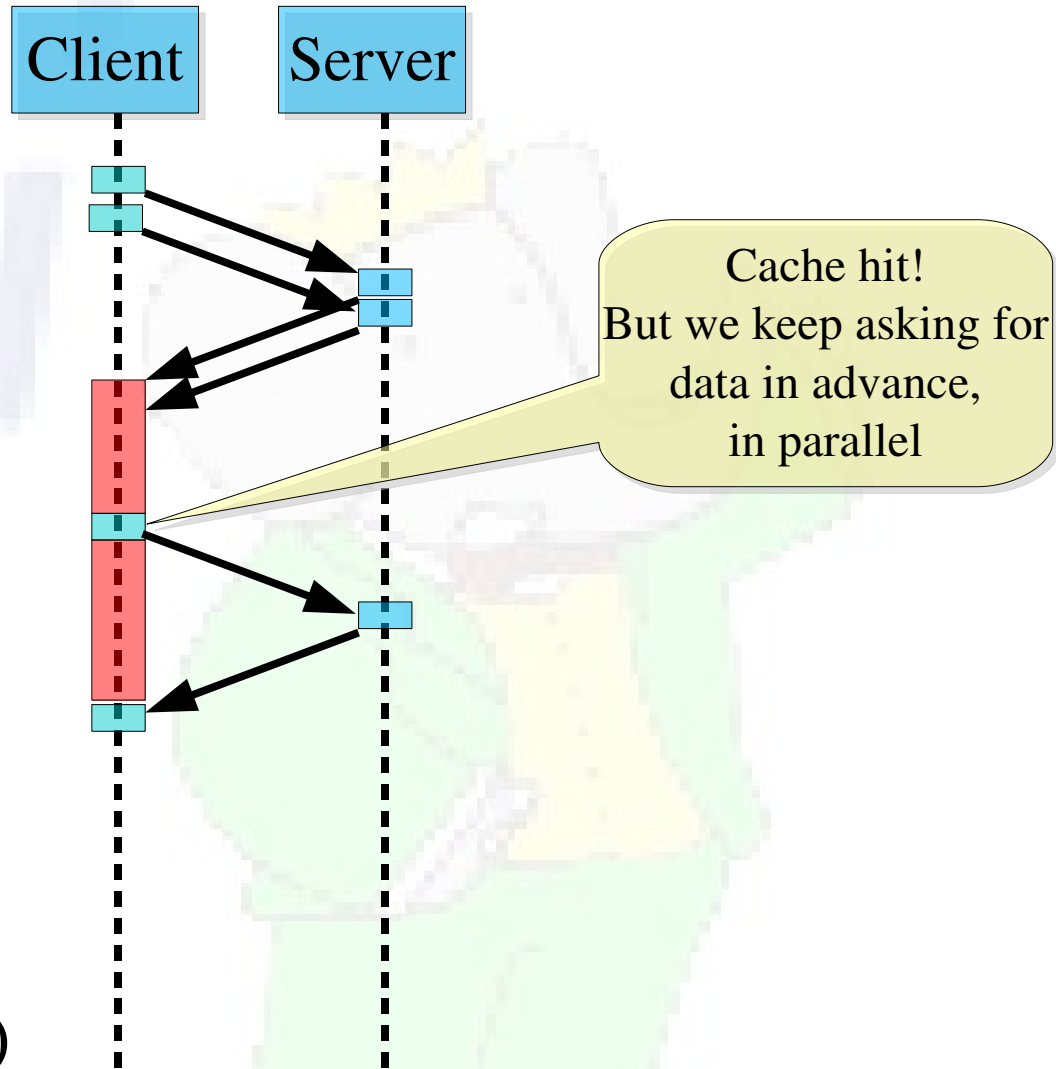
- The transmission latency (TCP) may vary
 - network load
 - geographical distance (WAN)
 - e.g. Min. latency PD-SLAC is around 80ms (about $\frac{1}{2}$ of the *ping* time)
 - e.g, if our job consists of 1E6 transactions
 - 160ms * 1E6 makes our job last 160.000 seconds more than the ideal “no latency” one
 - 160.000 seconds (2 days!) of wasted time
 - Can we do something for this?
-

Throughput

- The throughput is a measure of the actual data xfer speed
 - Data xfer can be a non negligible part of the delays
 - Throughput varies depending on the network
 - Capacity and distance
 - For WANs it is very difficult to use the available bandwidth
 - e.g. A *scp* app PD-SLAC does not score better than 350-400KB/sec, much less than the available bandwidth
 - Even more important:
 - We want to access the data files where they are
 - By annihilating all the latencies
-

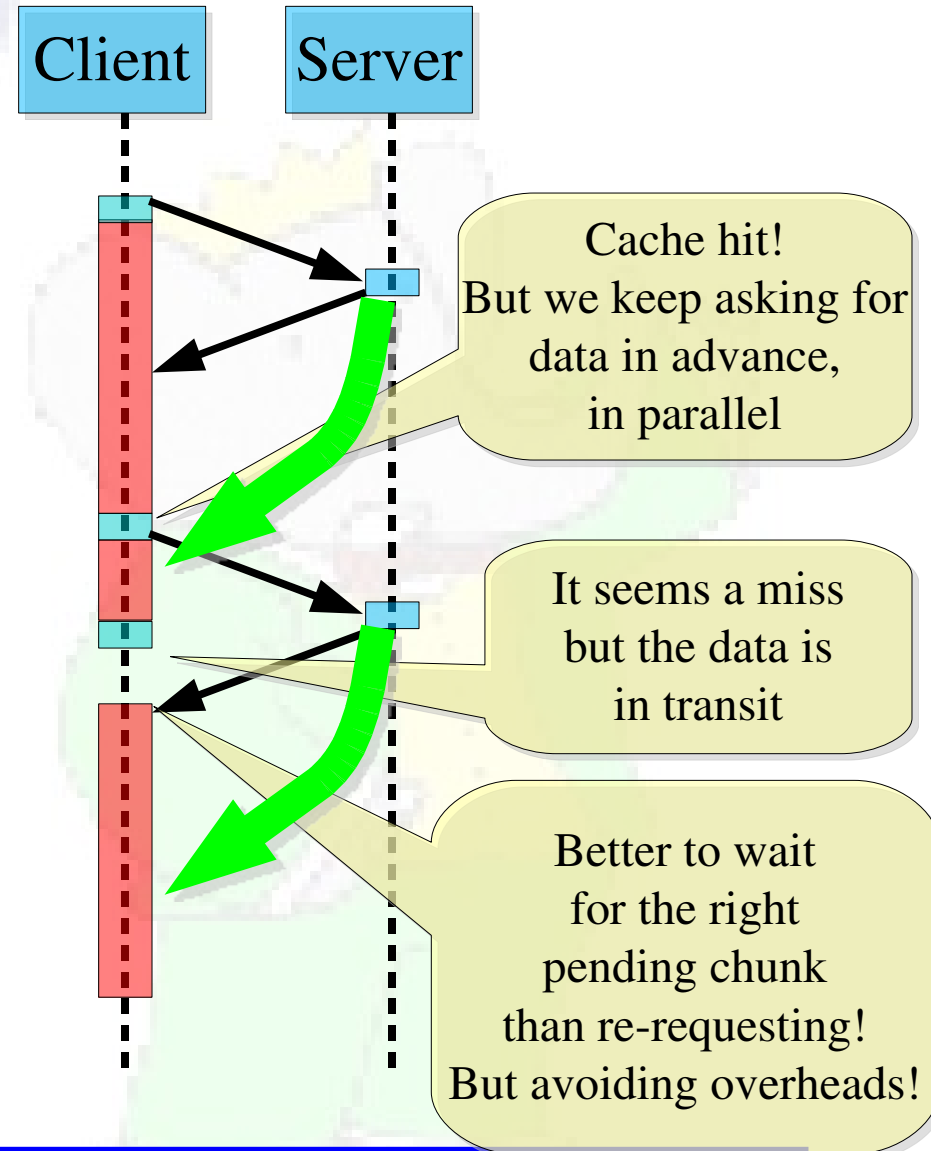
Prefetching

- Read ahead
 - the communication library asks for data in advance, sequentially
- Informed prefetching
 - the **application** informs the comm library about its near-future data requests
 - One/few at each request (async read) or in big bunches (vectored reads)

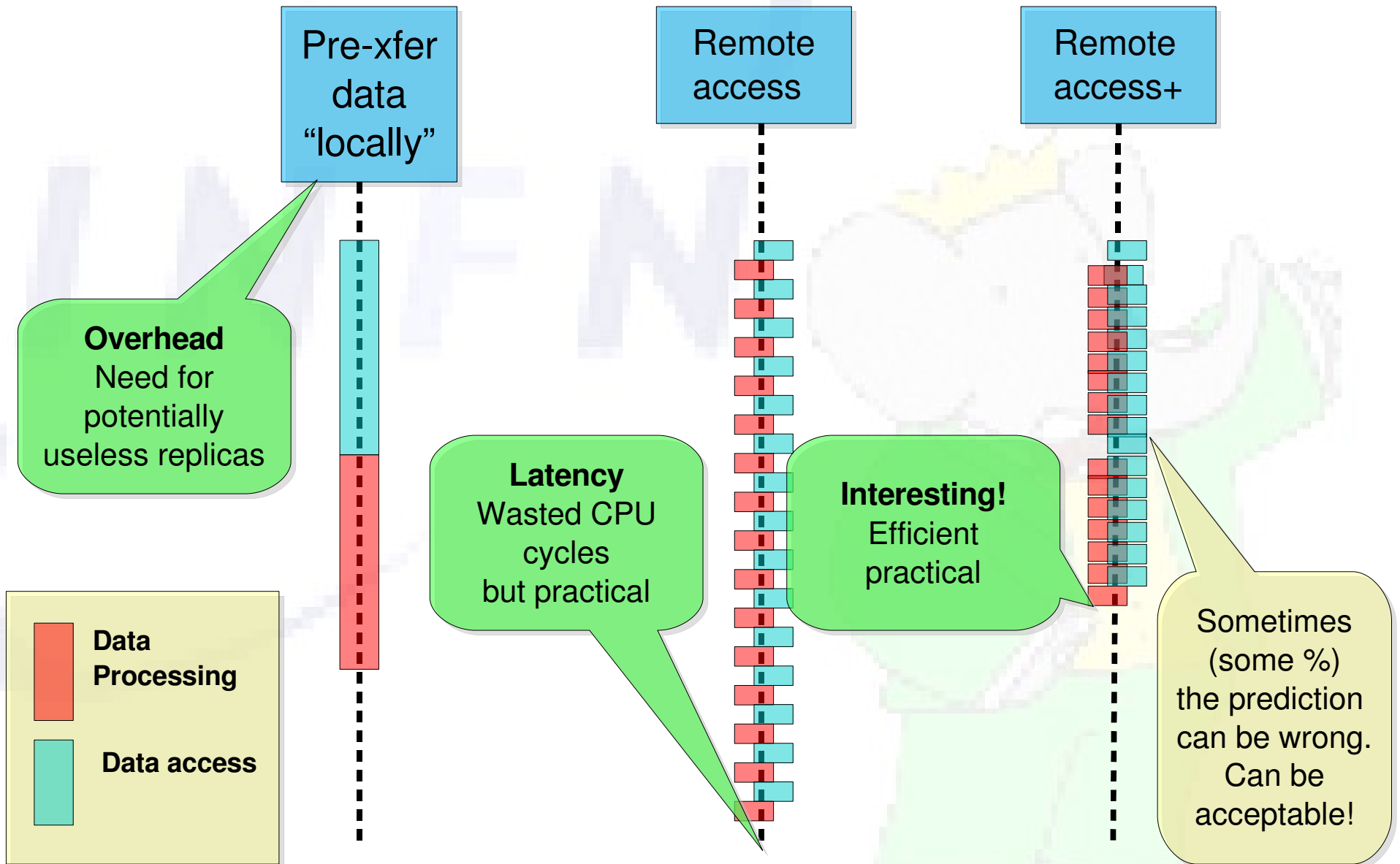


More than a cache

- Trouble: data could be requested several times
- The client has to keep track of pending xfers
- Originally named “Cache placeholders” (Patterson)

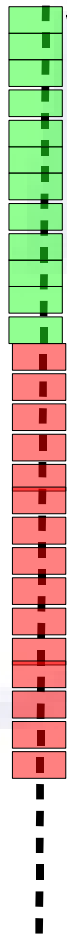


Intuitive scenarios (1/2)



Intuitive scenarios (2/2)

Vectored reads



Efficient
No data xfer overhead.
Only used chunks get prefetched

All (or many) of the chunks are requested at once.
Pays latency only once if no probability of mistakes.
But pays the actual data xfer

Async vectored reads



Inform that some chunks are needed. Processing starts normally, waits only if hitting a "travelling" one. In principle, this can be made very unlikely. (by asking chunks more in advance)

How to do it ? (1/4)

```
while not finished
  d = read(next_data_piece)
  process_data(d)
end
```

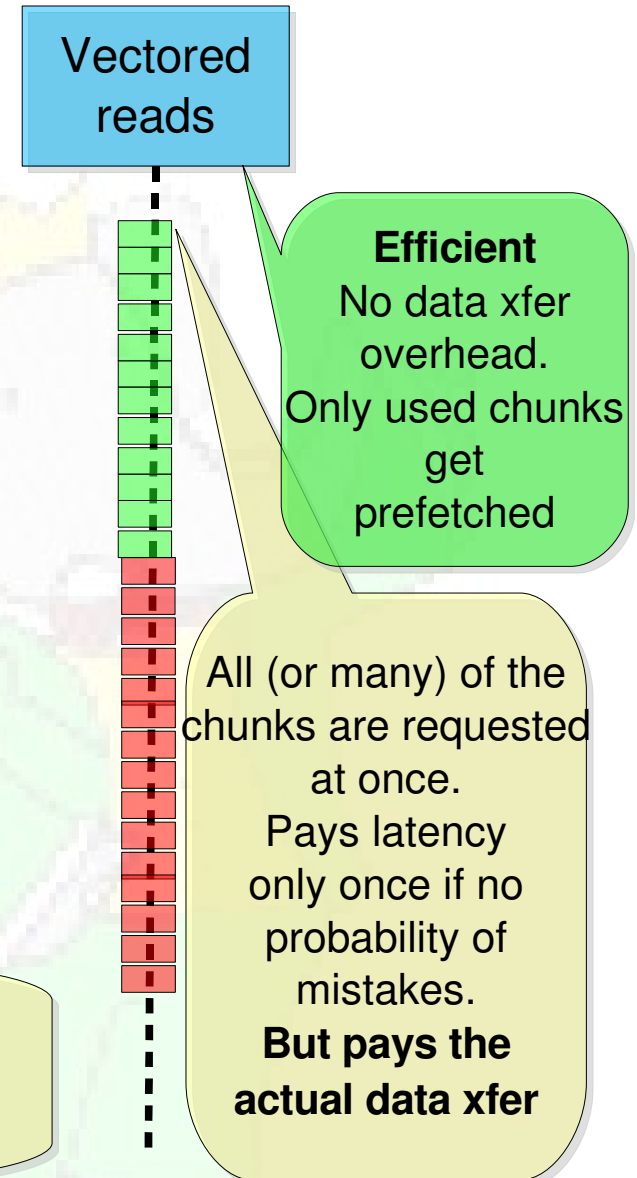


How to do it ? (2/4)

```
enumerate_needed_chunks()  
readV(needed_chunks)  
populate_buffer(needed_chunks)
```

```
while not finished  
    d = get_next_chunk_frombuffer()  
    process_data(d)  
end
```

All this can be transparent to ROOT
Ttree users via TtreeCache, implemented
using HUGE sync vectored reads



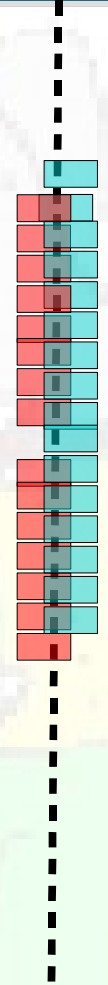
How to do it ? (3/4)

XrdClient keeps the data as it arrives, in parallel!

```
while not finished
  Read_async(n+1th_chunk)
  d = read(nth_chunk)
  process_data(d)
end
```

One data chunk is coming
while the preceding is processed.
We can also look more ahead (n+2, n+10, ...)

Remote
access+



The diagram shows a vertical dashed line representing a data stream. A blue box labeled 'Remote access+' is at the top. Below it, a series of overlapping rectangular blocks are arranged vertically. Each block is split horizontally into a red top half and a cyan bottom half. The blocks are staggered such that the cyan part of one block overlaps with the red part of the block below it. This illustrates parallel processing of data chunks. The blocks are contained within a light green, irregularly shaped area that represents the client's memory or processing window.

How to do it ? (4/4)

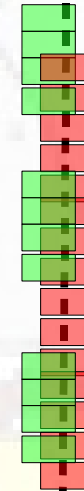
XrdClient keeps the data as it arrives, in parallel!

```
while not finished
  if (few_chunks_unprocd)
    enumerate_next_M_chunks ()
    ReadV_async (M_chunks)

    d = read(nth_chunk)
    process_data(d)
  end
```

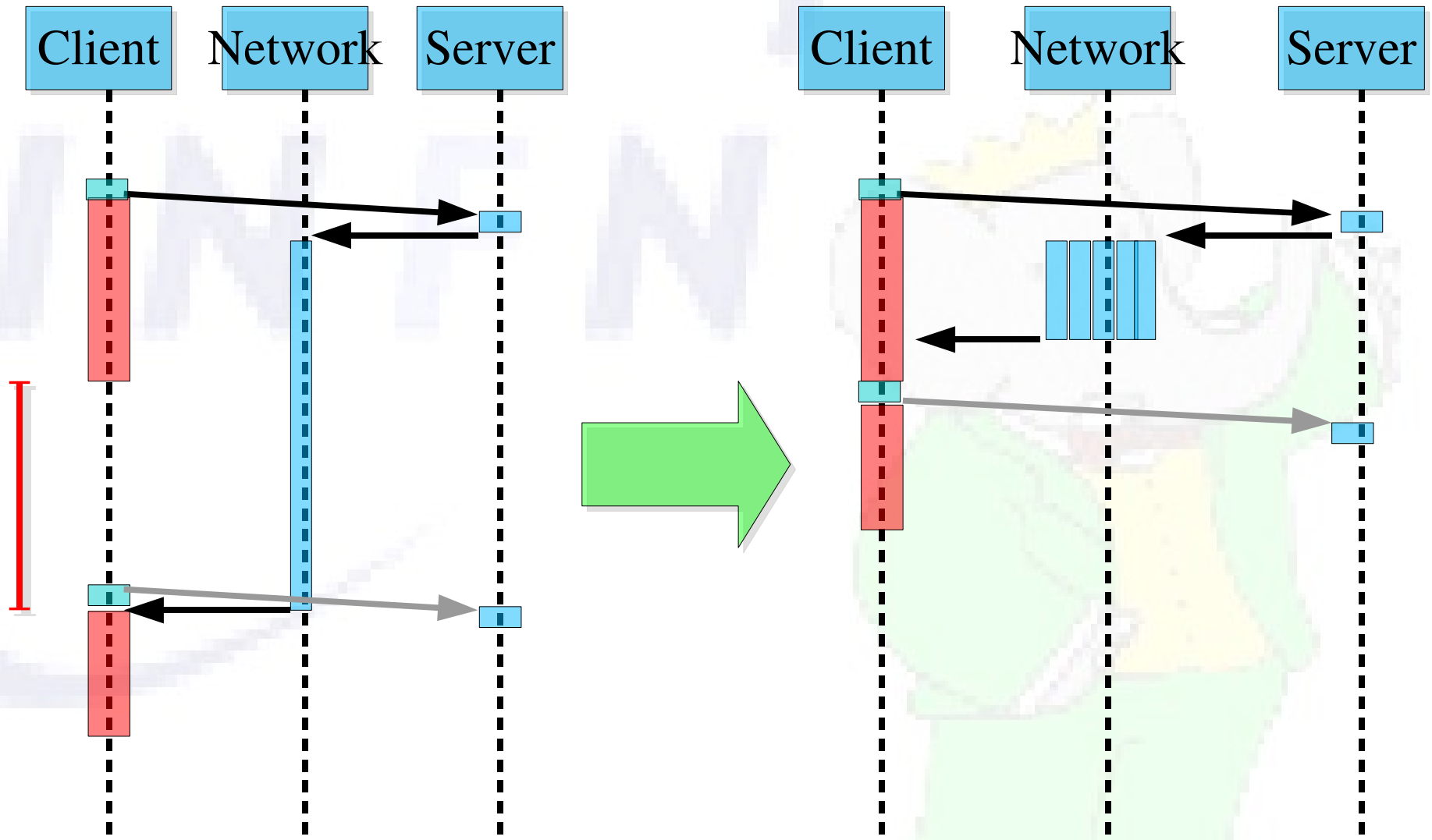
Many future data chunks are coming while the nth is processed. Generally little difference if the ReadV_async becomes a loop of Read_async. It depends!

Async
vectored
reads

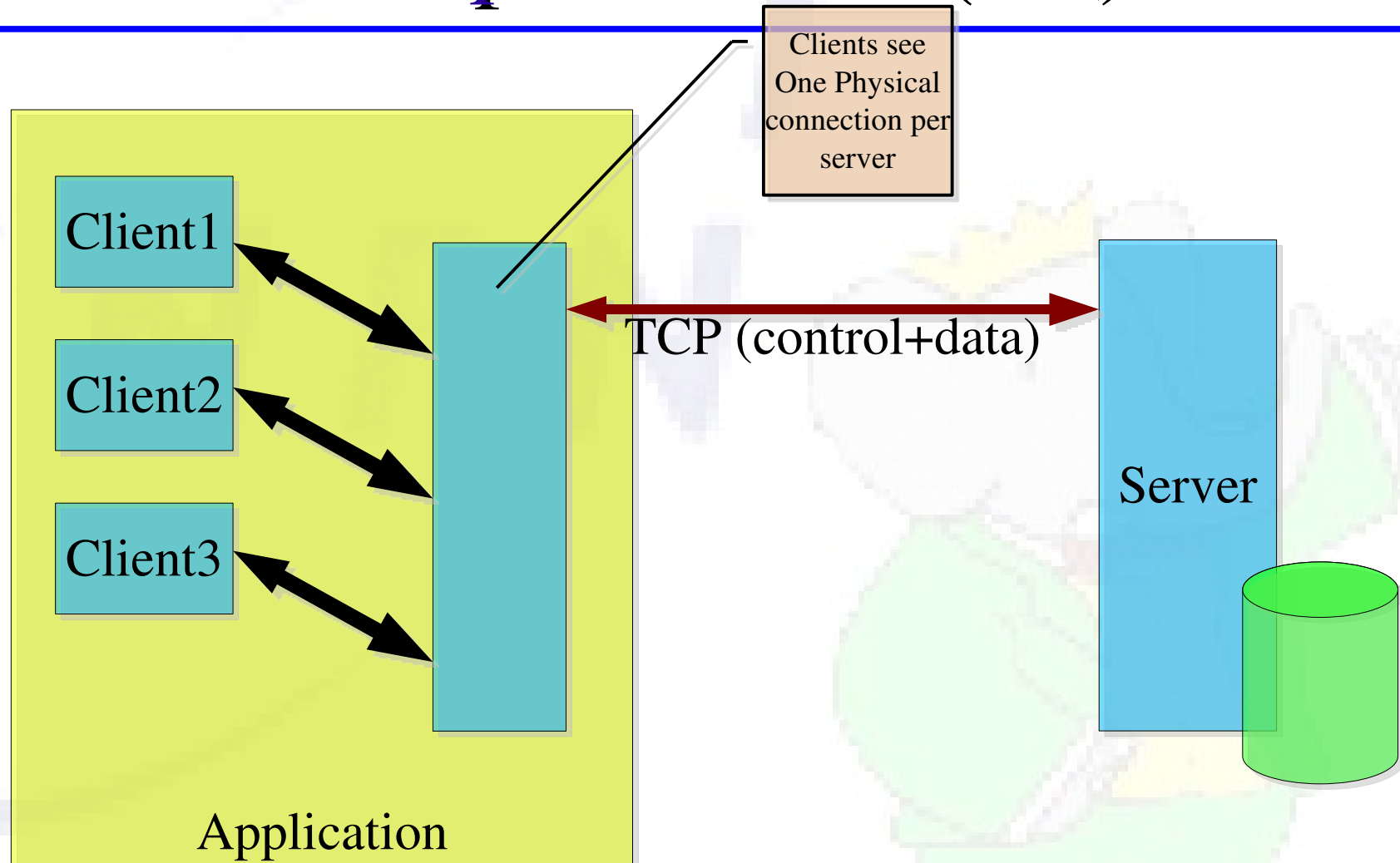


Inform that some chunks are needed. Processing starts normally, waits only if hitting a “travelling” one. In principle, this can be made very unlikely. (by asking chunks more in advance)

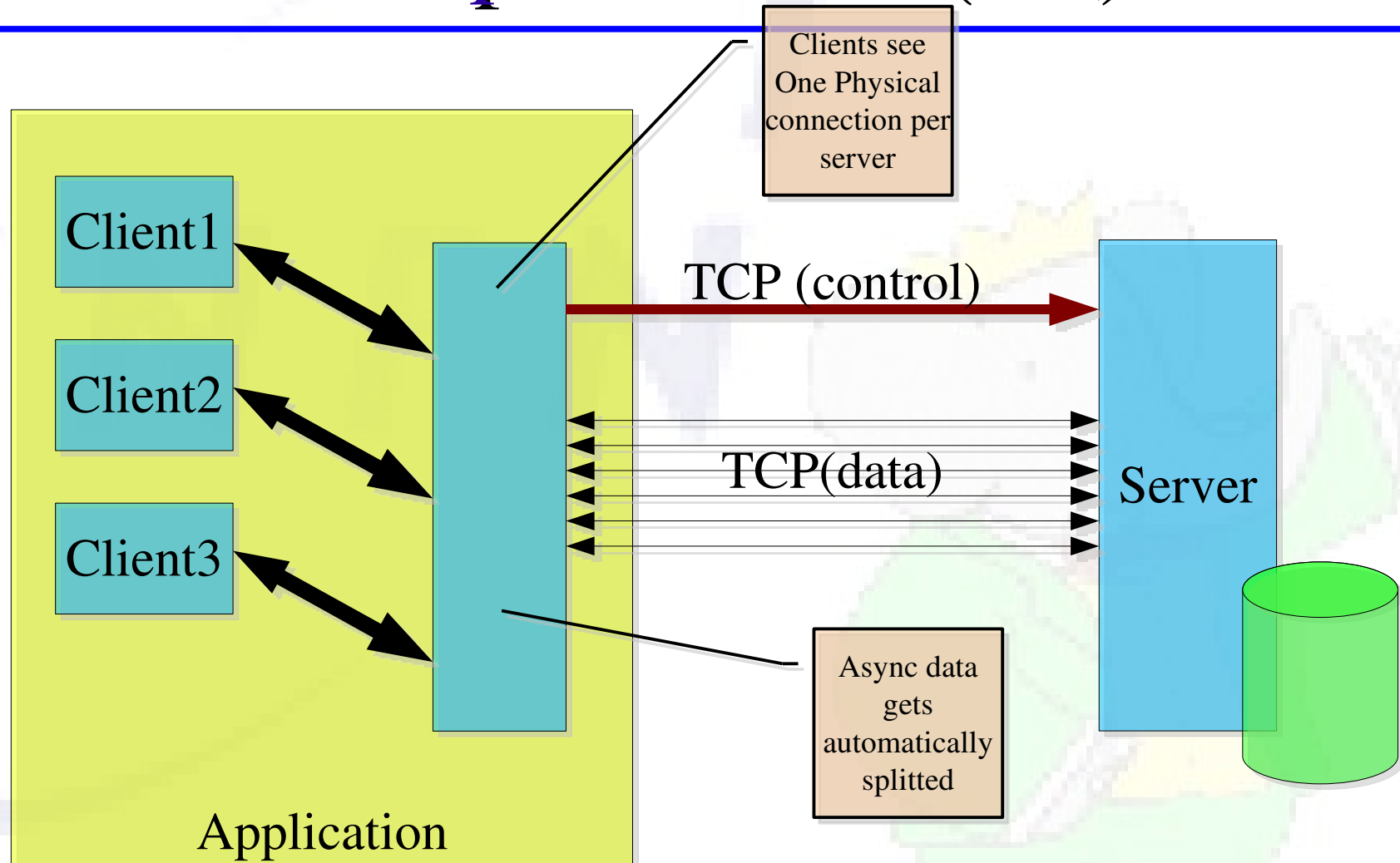
Throughput?



Multiple streams (1/3)



Multiple streams (2/3)



Multiple streams (3/3)

- It is not a copy-only tool to move data
 - Can be used to speed up access to remote repos
 - Totally transparent to apps making use of `*_async` reqs
 - *xrdcp* uses it (-S option)
 - results comparable to other cp-like tools
 - For now only reads are implemented
 - Writes are on the way
 - Automatic client-server WAN mode agreement
 - Very soon!
-

Resource access: open

- An application could need to open many files before starting the computation
- We cannot rely on an optimized struct of the application, hence we assume:

```
for (i = 0; i < 1000; i++)  
    open file i
```

```
Process_data()
```

```
Cleanup and exit
```

Problem: a mean of only 5 seconds for locating/opening a file will take 5000 seconds for the app to start processing

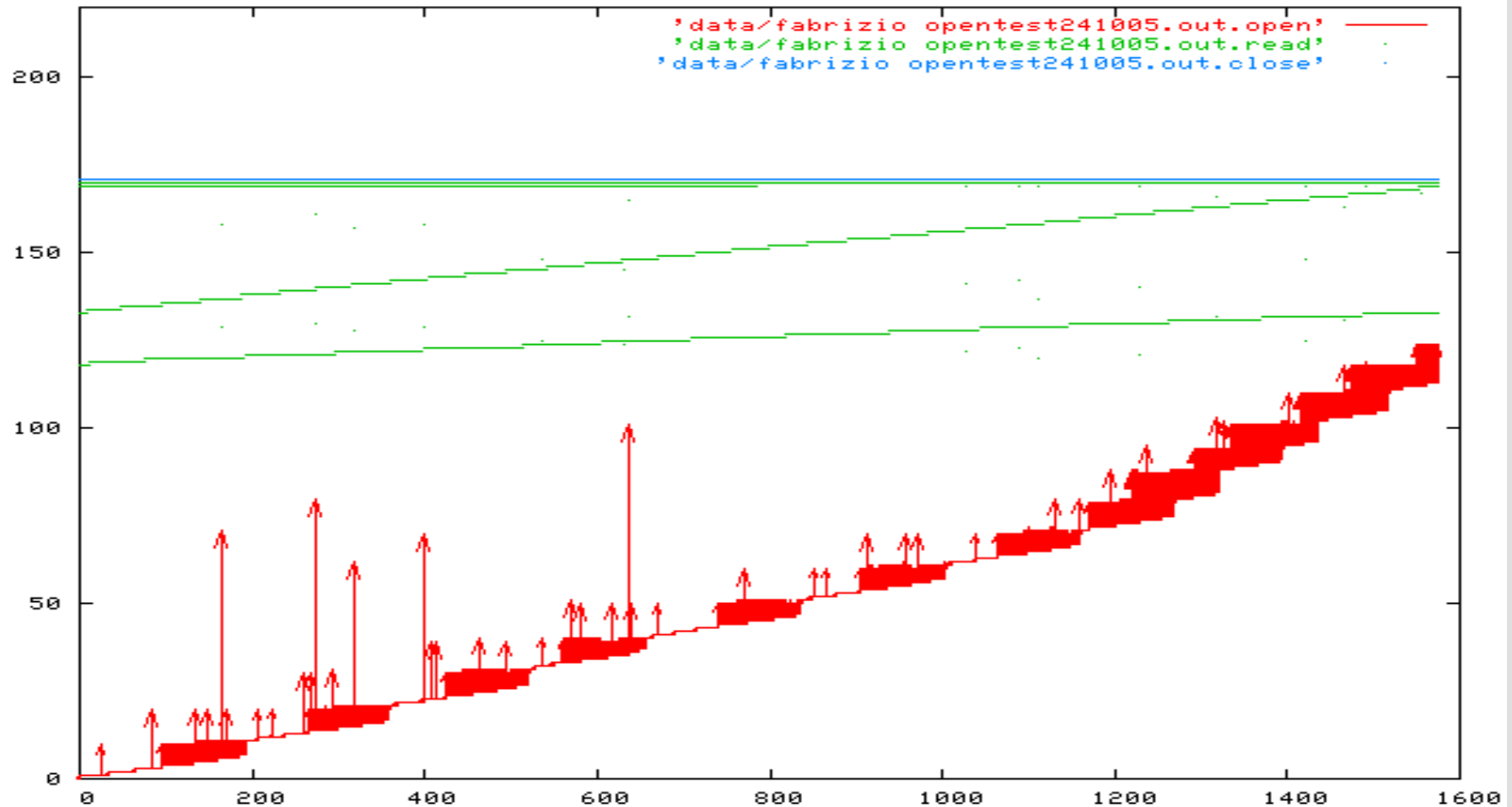
Parallel open

- Usage (XrdClient):
 - No strange things, no code changes, nothing required
 - An open() call returns immediately
 - The request is treated as pending
 - Threads takes care of the pending opens
 - The opener is put to wait ONLY when it tries to access the data (but still all the open requests are in progress)
- Usage (ROOT 5.xx):
 - The ROOT primitives (e.g. Map()) semantically rely on the completeness of an open() call
 - The parallel open has to be requested through

```
TFile::AsyncOpenRequest(filename)
```

Parallel open results

Test performed at SLAC towards the “Kanga cluster”
1576 sequential open requests are parallelized transparently



Conclusion

- Implementation platform: **xrootd**
 - `readV`, `readV_async`, `read_async` are OK
 - ROOT TTrees can use `readV`
 - Parallel Open is there too
 - Multistream transfers are OK
 - used by *xrdcp*, usable for `readV_async` and `read_async`
 - Purpose (other than raising performance):
 - To be able to give alternative choices for data or replica placement
 - Let people choose why one should need replicas (more reliability, more performance, willing to 'own' a copy of the data, ...)
 - To move towards computing models where some phases can rely on remote data
 - Ev. for backup or fault tolerance purposes (if the nearest location breaks, fallback to another one with little or no overhead)
-