Data access performance through parallelization and vectored access

Some results

Fabrizio Furano
INFN – Istituto Nazionale di Fisica Nucleare
furano@pd.infn.it
Andrew Hanushevsky
SLAC – Stanford Linear Accelerator Center
abh@slac.stanford.edu

Motivation

- We want to make WAN data analysis convenient
  - The typical way in which HEP data is processed is (or can be) often known in advance
    - TTreeCache does an amazing job for this
  - *xrootd*: 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 in many cases

A simple job

- An extreme 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.

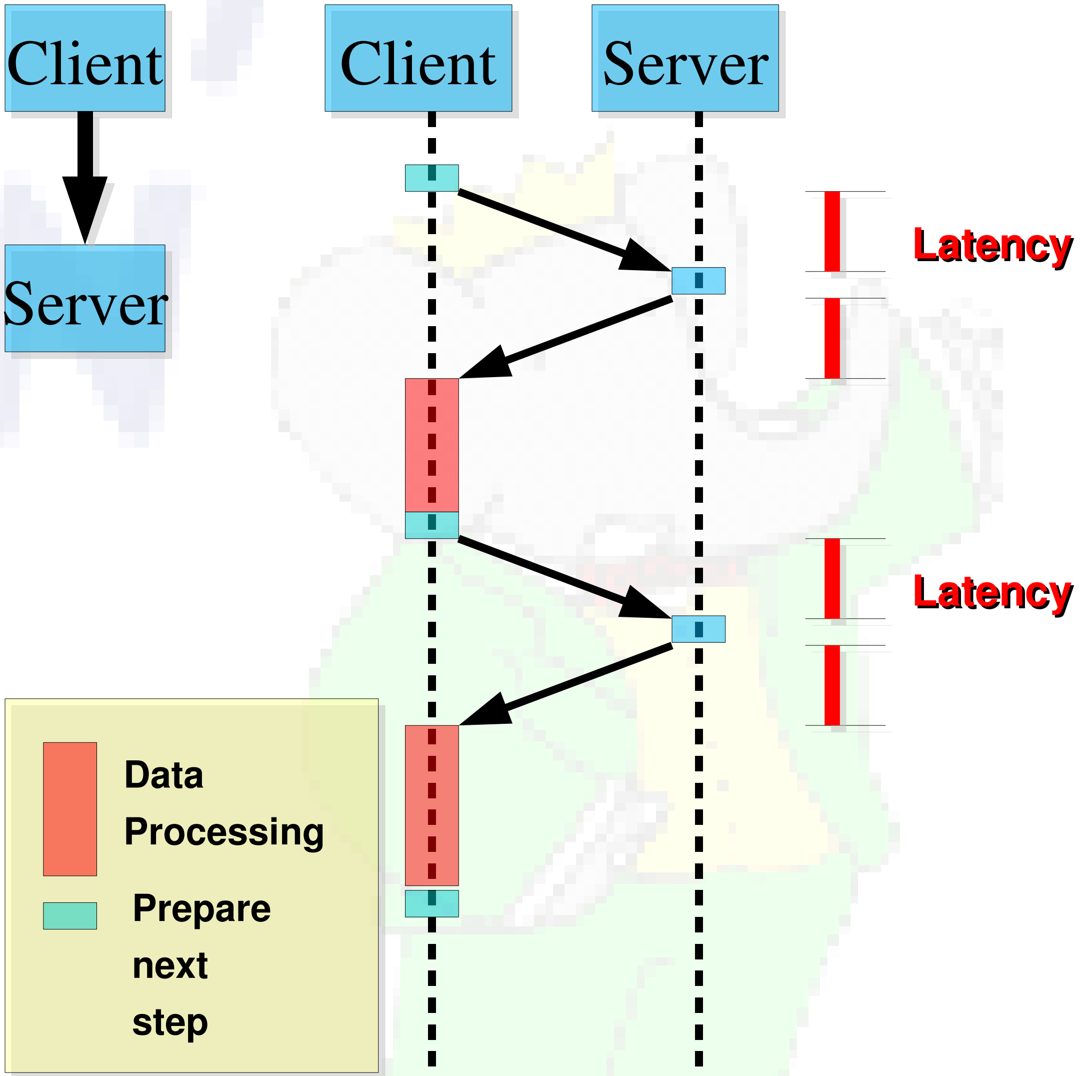
Hypothesis: the data are not local to the client machine

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 ½ 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
- The worst enemy for WAN data access.
    - Achievable throughput comes second
- Can we do something for this?

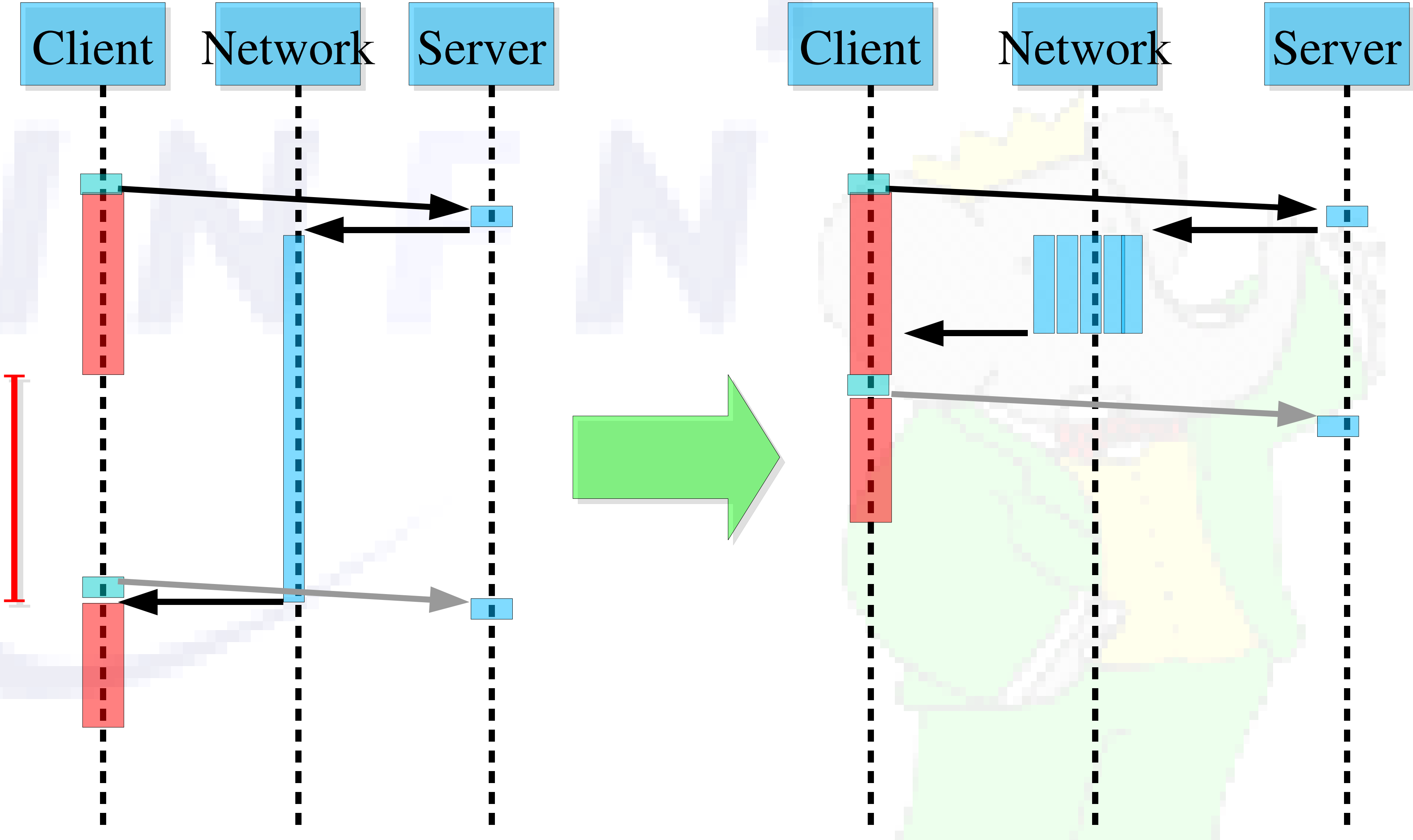
Latency

- Two ways to attack this issue:
  - Hiding it
    - Parallelism: request/send many chunks at the same time
    - If the app computes between chunks, we can xfer data while the app is computing
  - Gluing chunks together
    - Divides by a const factor the number of times we pay the latency
    - Huge readaheads (e.g. for cp-like apps which read everything)
    - Vectored reads
    - Multiple single overlapped async requests (reordered by the client)
- In any case we need to know something about the future.
    - Statistically (the caching + read ahead case, leads to the “hit rate” concept)
    - Exactly (the ROOT `TTreeCache` is fine for that)

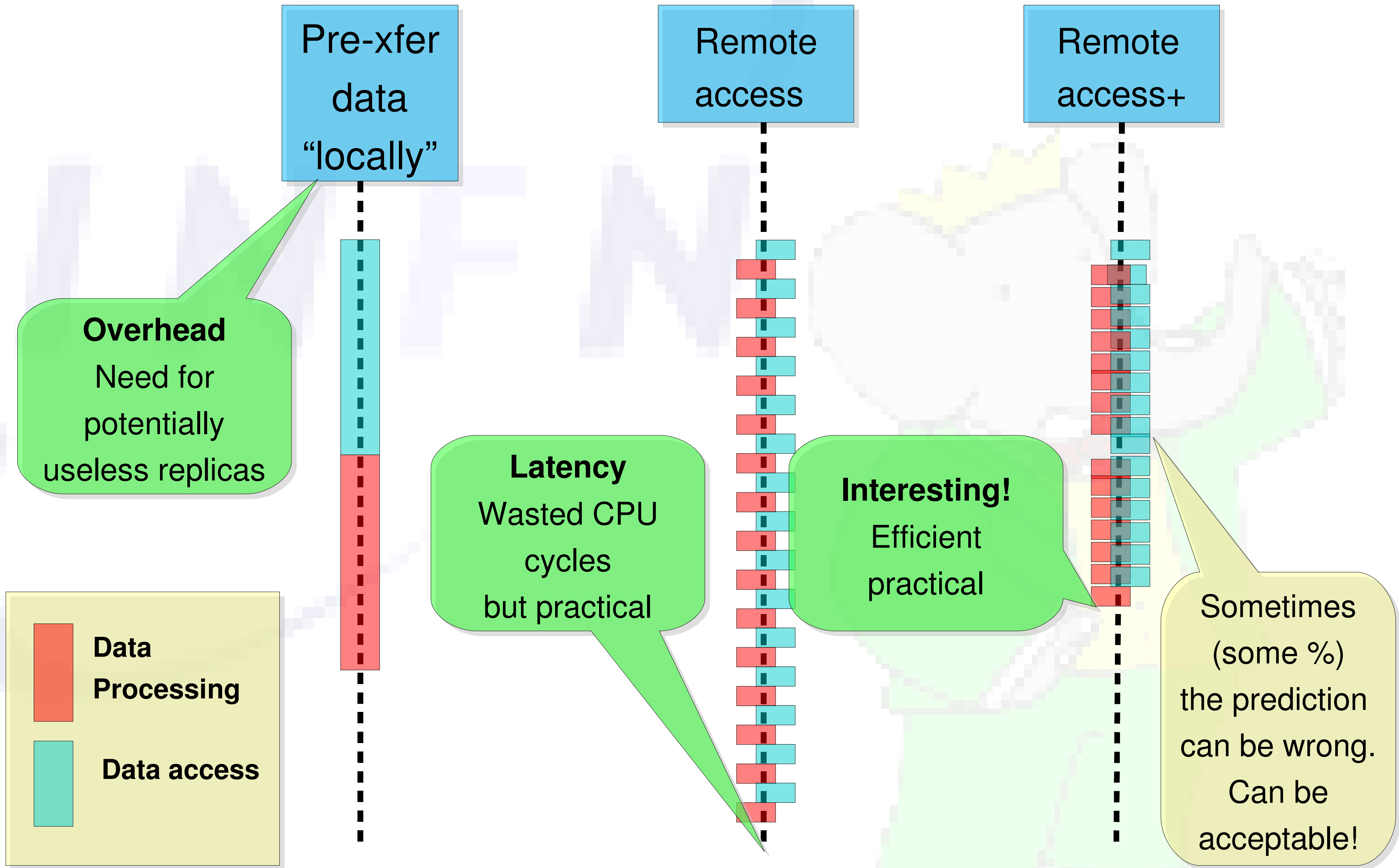
Throughput

- 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 TCP 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 and USING the bandwidth

Throughput via multiple streams

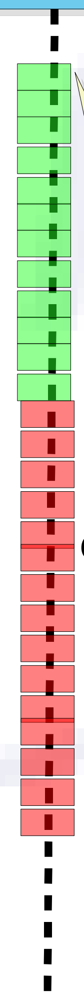


Intuitive scenarios (1/2)



Intuitive scenarios (2/2)

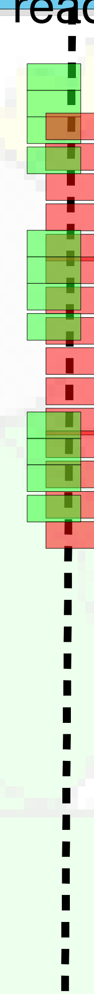
Vectored reads



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)

First evaluation

Processing at CERN (MacBook Duo), data at SLAC (160ms RTT)

	Open File	Draw Branch	Total
Local File	0.218123	6.76074	6.978863
sync readv	2.39714	21.2345	23.63164
async (1 str.)	2.96605	14.4711	17.43715
async (5 str.)	4.99053	11.9273	16.91783
async (10 str.)	7.46079	12.5978	20.05859
async (15 str.)	9.87976	14.921	24.80076

Courtesy of Leandro Franco and the ROOT team

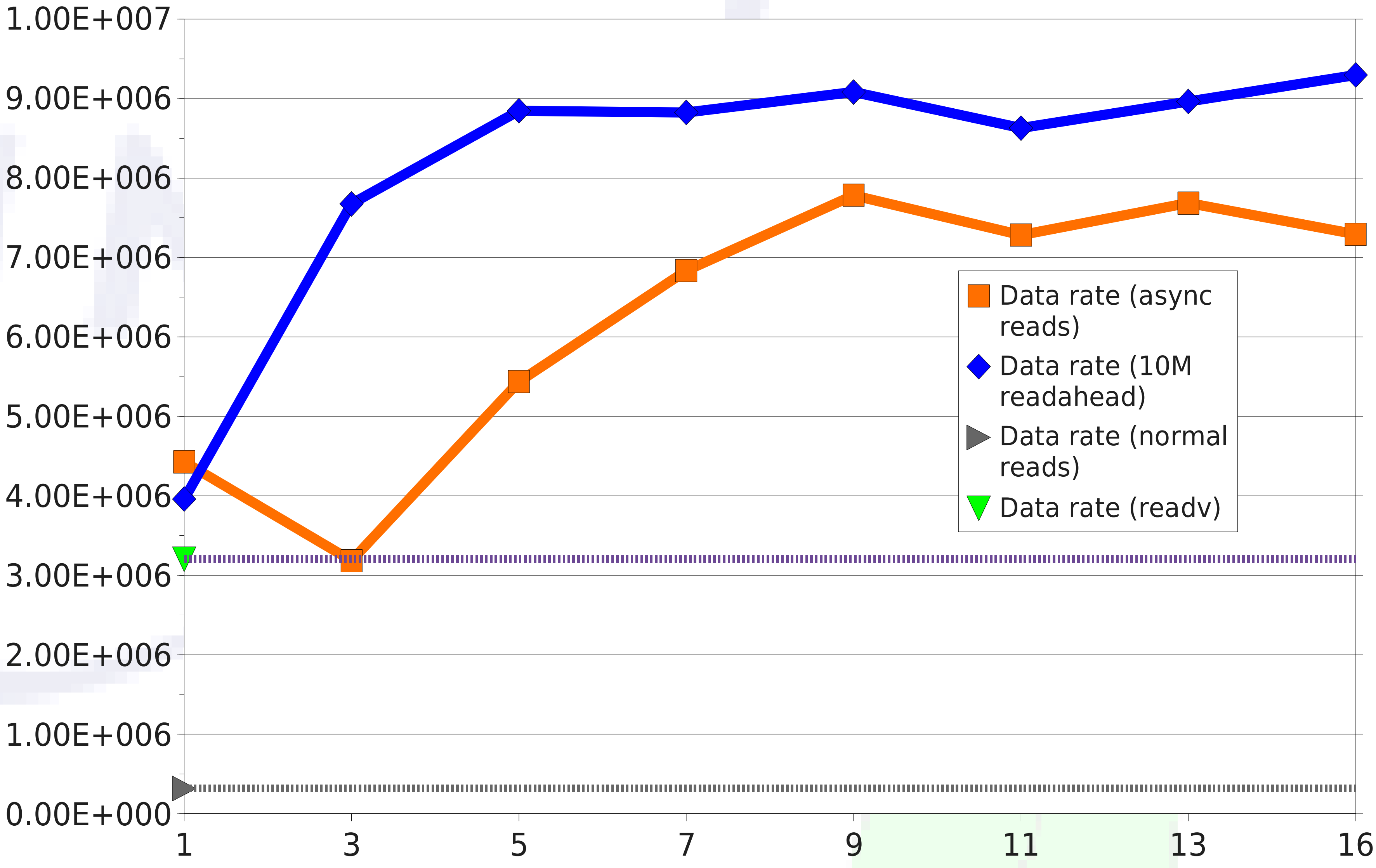
First evaluation

- Confirmation of the huge improvement
    - Simple strategy: >30min of wasted time!
  - With some surprises
    - Reading async the single small chunks (some KB) seemed faster than reading them as a unique readv
    - 15 streams show a data rate worse than 10 streams
    - Opening streams towards SLAC takes 0.6 secs per stream
      - Maybe we can reduce this to <1 sec in total
      - Anyway not a dramatic problem
        - Latency paid only once when contacting a server
        - The connections are shared between multiple clients/files through the XrdClient connection multiplexing
- We wanted to decide where it is worth optimising
    - And what to do and what to expect

A simple benchmark using WANs

- One server in Padova, hosting a 500MB file
- A benchmark client app
  - At SLAC (noric) Ping time: about 160ms
  - At IN2P3 (ccali) Ping time: about 23ms
- We want to read 50000 chunks of 10KB each
    - legacy reads, async reads in advance (1024), huge readaheads , readV (up to 20480 subchunks)
  - Spanning all the file
  - The sorting and the disks performance are of secondary importance here
    - If compared to the latency of the network
    - Only the huge readaheads need chunks to be sequential

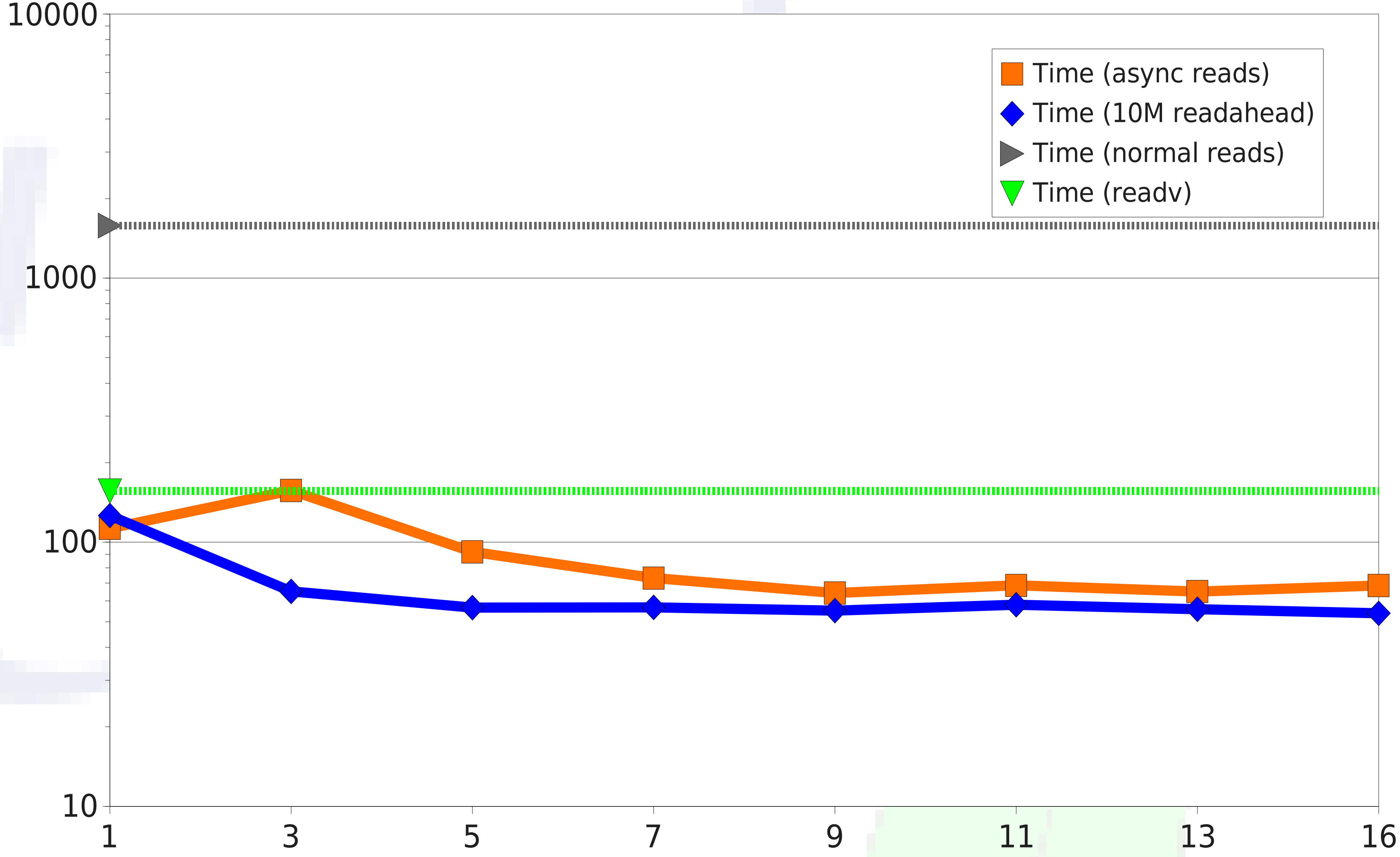
Data rate vs stream count: IN2P3<--PD



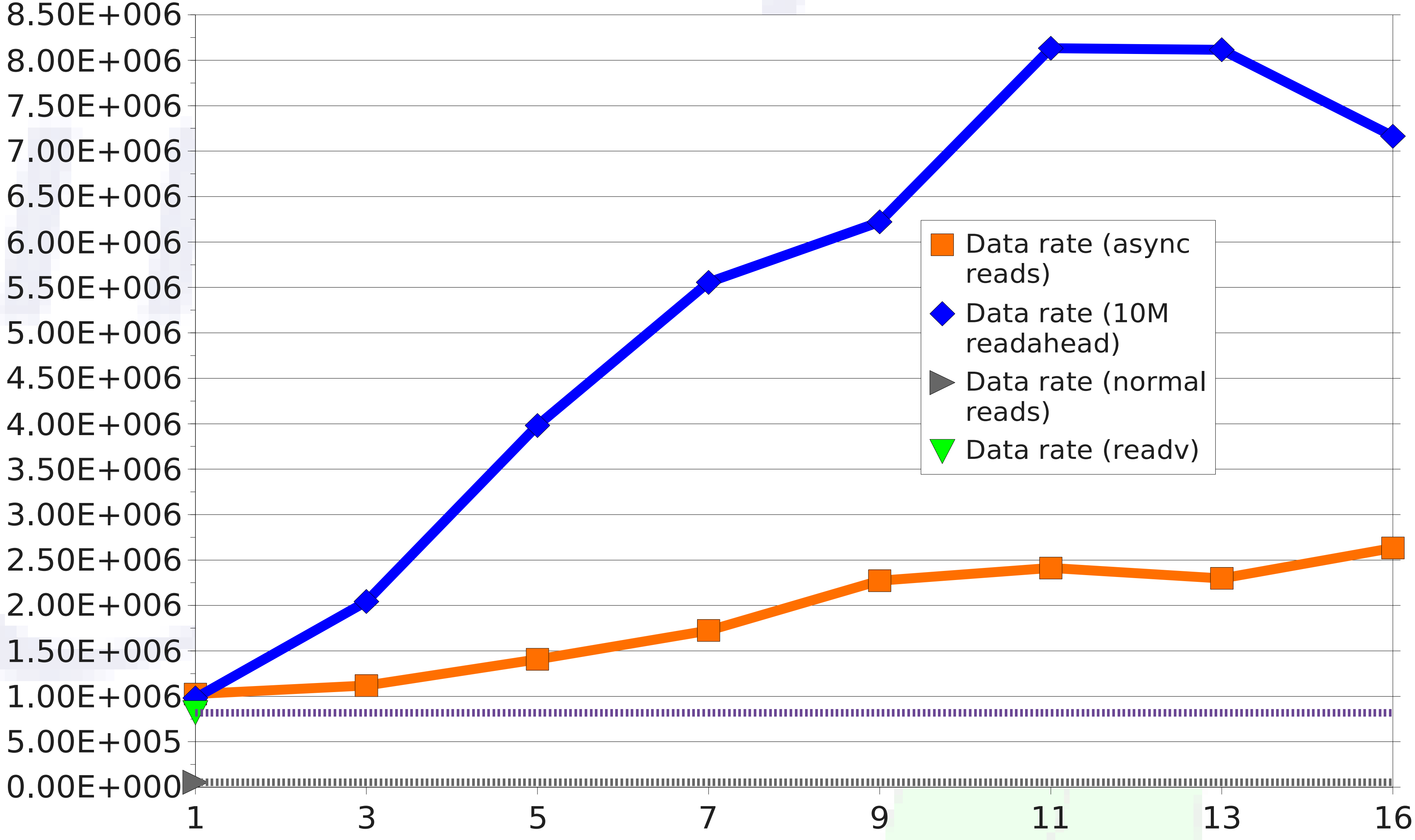
IN2P3<--PD test case

- no additional streams: async chunk xfer slightly faster than sync readv
  - We also remember that async chunk xfer makes the server able to overlap the operations
  - the network bandwidth tends to saturate with few streams
  - the max data rates are comparable (9MB/s vs 7.5MB/s)
    - But there is still room for improvement
- We get a nearly 24X performance boost
  - Reaching about 80% of the higher one
  - And we did not have a processing phase to hide latencies behind
  - Heading for 28X

Time elapsed vs stream count: IN2P3<--PD



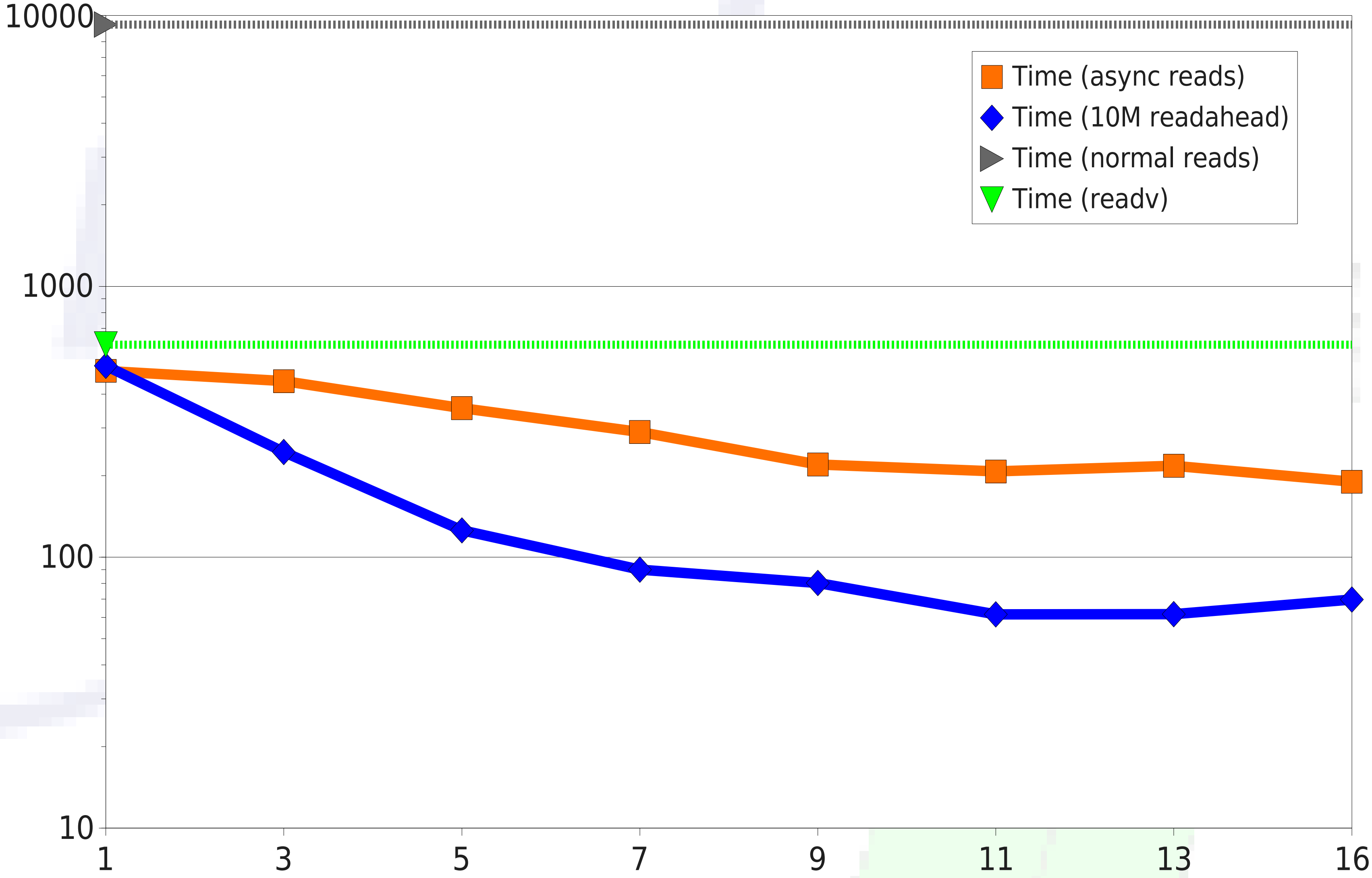
Data rate vs stream count: SLAC<--PD



SLAC<--PD test case

- no additional streams: async chunk xfer still performs better than sync readv
  - With sync readv the app can proceed only if the whole block is complete. A delayed TCP packet puts in stall the whole super-chunk.
- with the small chunks via multistreaming, async chunk xfer gets a boost not better than 3X
  - With large chunks we reach a plateau of 9-10X (>8MB/s)
  - Hence, there is room for improvement in the many streams-small chunks case
  - As the sub-chunk size decreases (< 700KB), performance decreases (see 15 add. streams with the huge readahead)
- 44X boost over the "legacy" case (heading for 150X!)

Time elapsed vs stream count: SLAC<--PD

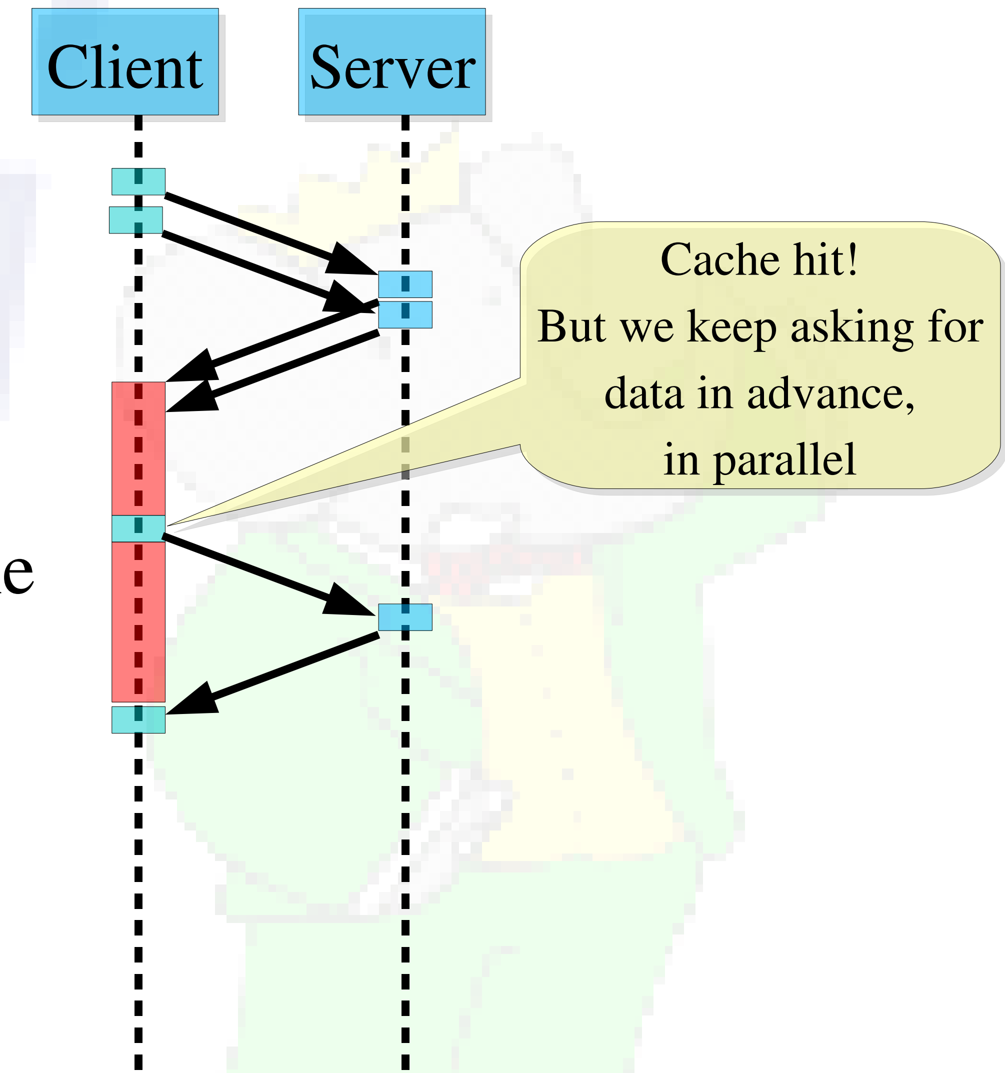


Conclusion

- Implementation platform: **xrootd**
    - `readV`, `readV_async`, `read_async` are OK
    - `readV_async` is not multistream aware yet
        - We expect to fill with this the performance gap for small chunks
    - ROOT TTrees have the TTreeCache, the async xfers are under advanced dev/test
    - Add to this the fault tolerance and the Parallel Open
  - Purpose (other than raising performance):
    - To be able to give alternative choices for data or replica placement, eventually for fallbacks
        - Let people choose why one should need replicas (more reliability, more performance, willing to 'own' a copy of the data, ...)
        - Without forcing everybody to get (and keep track of!) potentially useless ones.

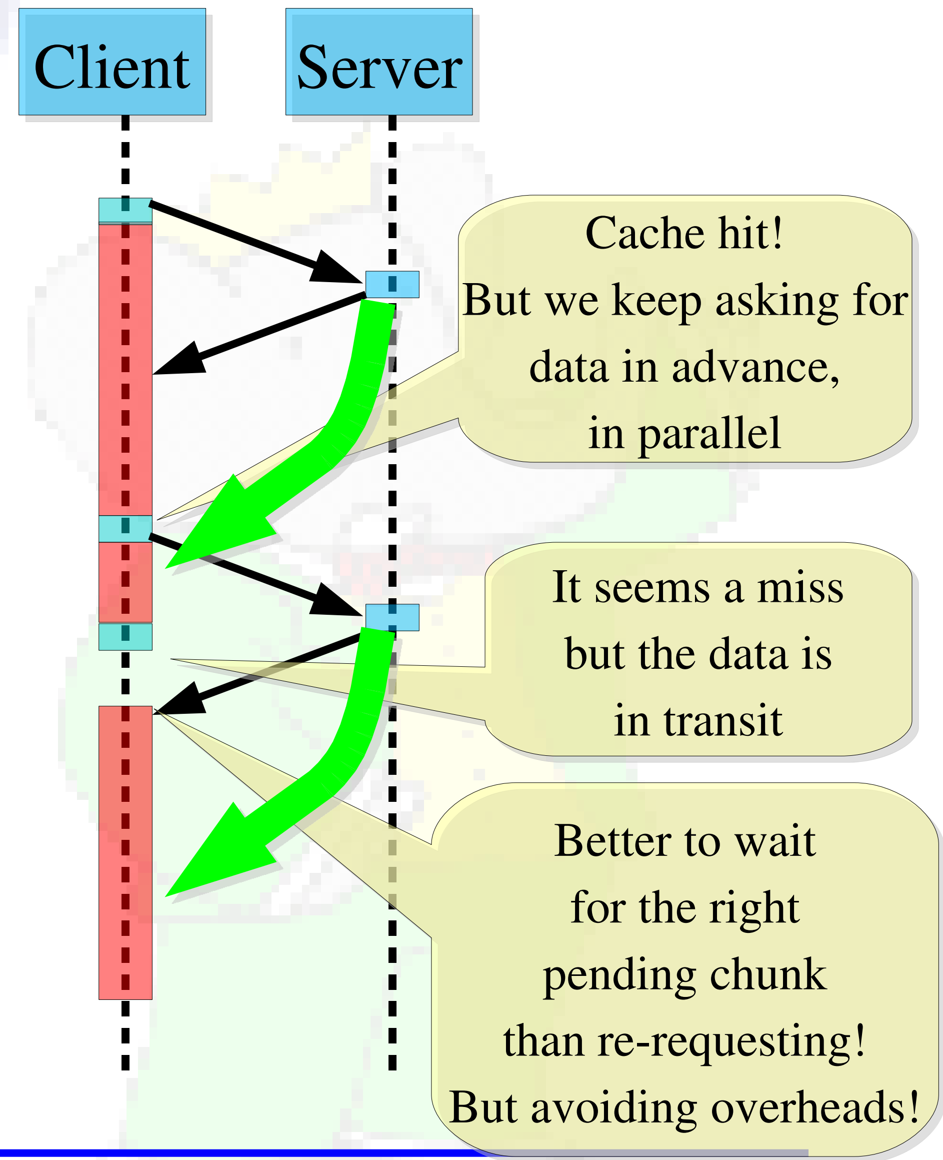
Prefetching

- 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)



How to do it ? (1/4)

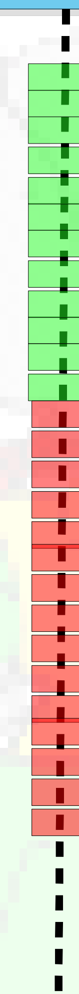
```
while not finished
  d = read(next_data_piece)
  process_data(d)
end
```

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
```

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

All this can be transparent to ROOT Ttree users via TtreeCache, which now is used to feed HUGE sync vectored reads

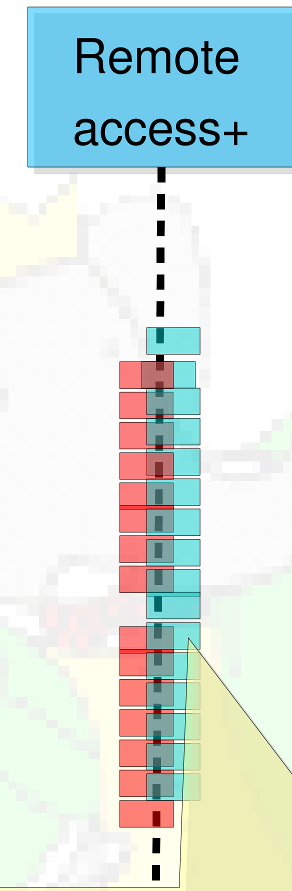
How to do it ? (3/4)

XrdClient keeps the data as it arrives, in parallel!

```
while not finished
  if (few_chunks_travelling)
    enumerate_next_M_chunks()
    foreach chunk
      Read_async(chunk)

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

Many data chunks are coming
while this one is processed.
We can also look more ahead (M=2...1024 ...)



While the app processes one, we want to keep many travelling chunks, as many as the network pipe can contain, to "fill" the latency.

How to do it ? (4/4)

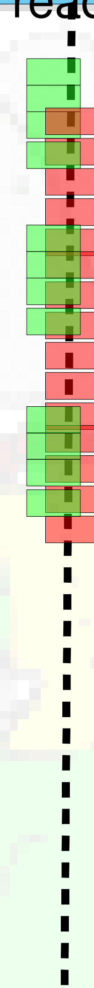
XrdClient keeps the data as it arrives, in parallel!

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

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

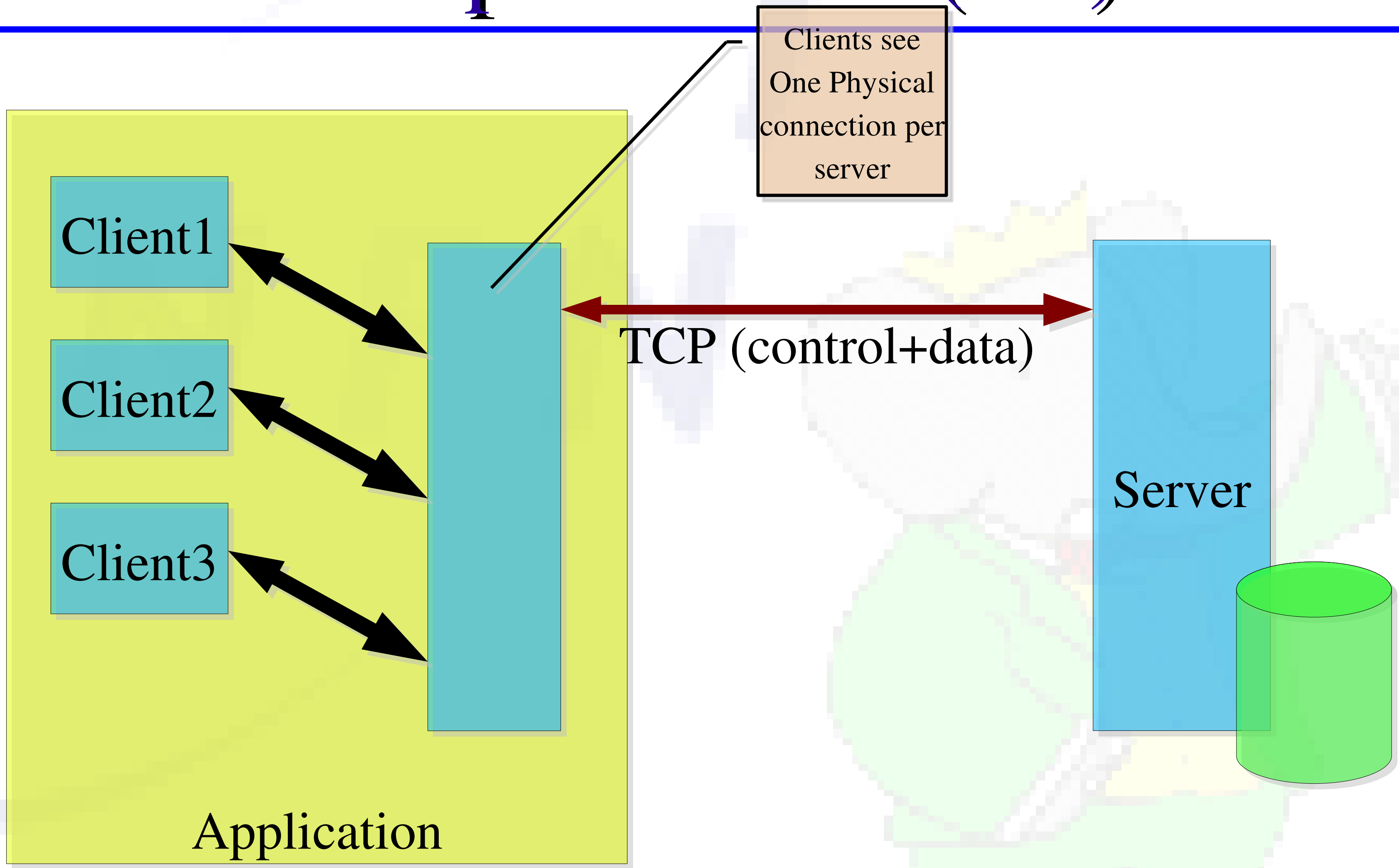
Many future data chunks are coming
while the nth is processed. Unfortunately the multistream
xfer is not yet ready for the async readVs :-(

Async vectored reads

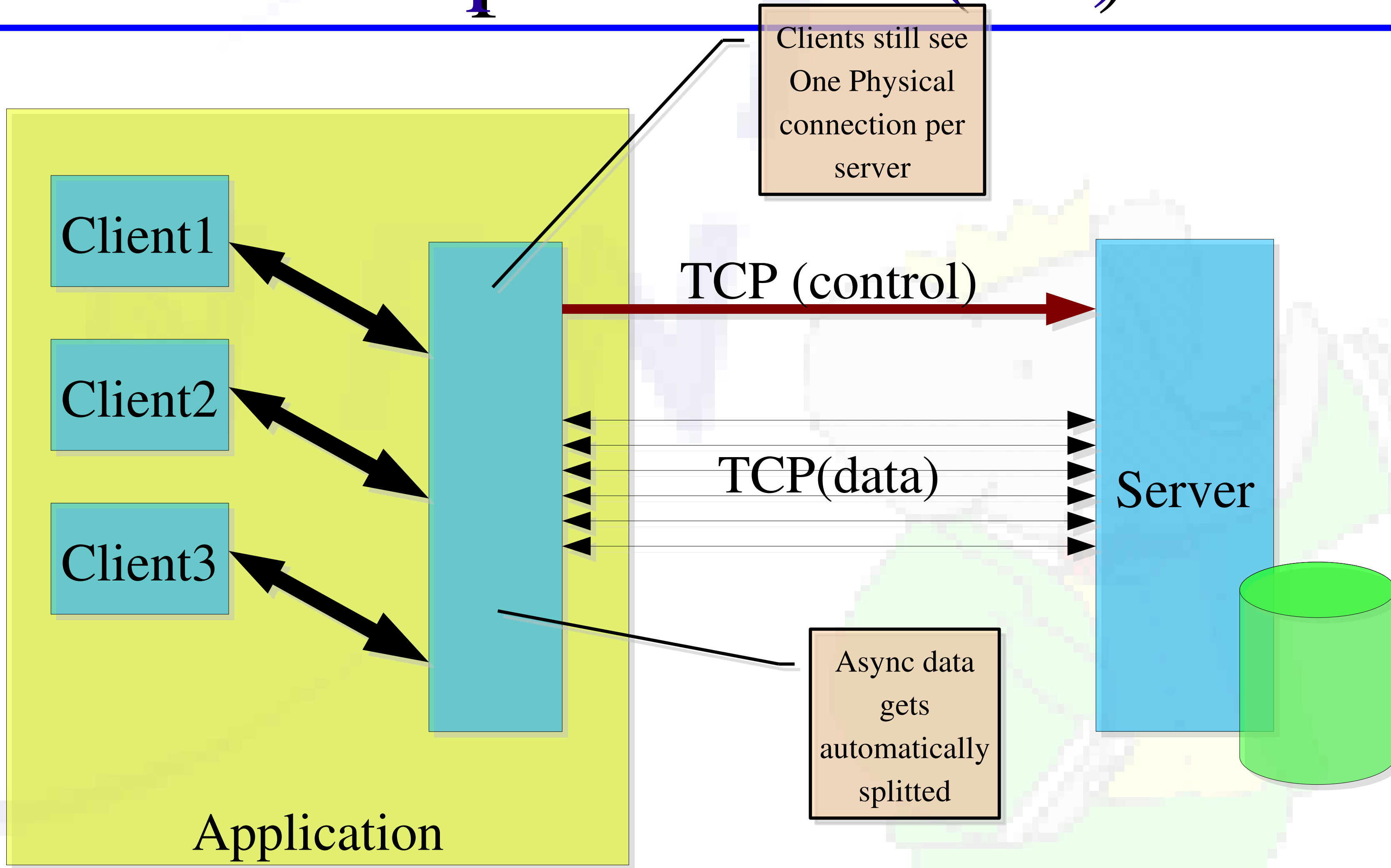


Inform that some chunks are needed. Processing starts normally, waits only if hitting a “travelling” one. In principle, this can be made very unlikely. (by asking chunks more in advance)

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.
    - Not easy to keep the fault tolerance
- Client and server automatically agree on the TCP windowsize

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