Fabrizio Furano: “From IO-less to Networks”

Part 2

- Techniques to higher the I/O performance

A simple idea: pack things together

- Instead of requesting one data chunk at a time and waiting for it:
  - Request two chunks IN THE SAME REQUEST
  - When they arrive, put them in a buffer
  - The application reads twice from this buffer
- In principle by doing this we have cut the total latency by two
  - Because, trivially, we have cut the number of interactions
  - In our million-chunks job this means being idle 10m instead of 20m
  - This mechanism can be put to work for many more chunks
    - And cut the latency by a bigger factor

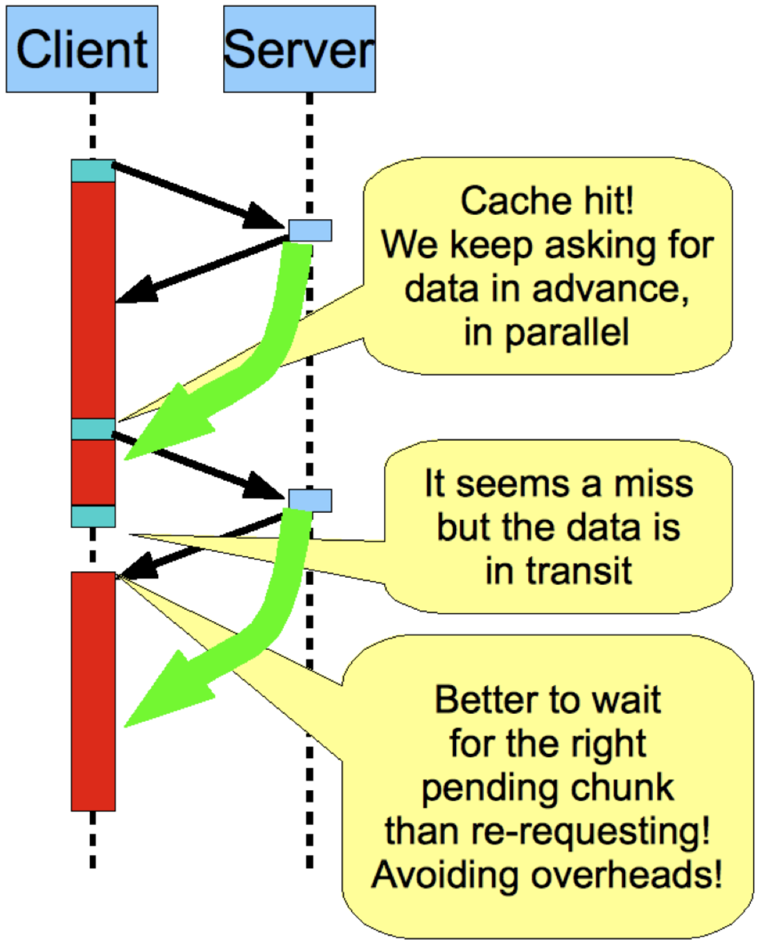
Another idea: work in the background

- Request the (n+1)th chunk just before starting processing the (n)th
  - This implies some form of tricky lower level parallelism
  - While the app computes, the data flows
    - When the app finishes computing the (n)th chunk, it again:
      - Requests the (n+2)th
      - Wait for the (n+1)th to finish its incoming path
        - Eventually it's already available
      - Process the (n+1)th

Two ideas or just one?

- It may look the same idea, but they imply completely different architectures/ implementations
  - 1: the ability of issuing composite requests
  - 2: the ability of handling asynchronous data transfers
- Both techniques deal with ***when*** to issue a data request

Work in the background



How to do it… in principle

- In the real world these can be accomplished in two different ways:
  - A Vectored read primitive (readv)
    - Instead of a single couple (offset, length) we want to aggregate many of them in the same request
      - A single request pays the network latency once
      - Even if it is much heavier
      - All the data chunks will travel together, serialized in a big composite data block
      - The request will be much more demanding for the disk
        - But there is a higher probability that it will be treated efficiently
  - Asynchronous capabilities in the communication
      - The client can send requests without waiting for each response
      - The responses are collected by a parallel thread
      - The app gets the data from internal buffers populated by the requests

About vectored reads

- We are speaking about a concept
  - Aggregating multiple requests into one
  - Send these composite requests through the network to a server which supports them
  - Then, the server forwards these requests to the disk system it is connected to
  - Doing this way:
    - We aggregate requests at the app/network level, cutting the network latency (well, dividing it by a ‘big’ factor)

About vectored reads

- This kind of request must be supported by the data transport protocol
  - e.g. xrootd, dcap, http, etc.
- The server receives such a request
- And then forwards it to its disk system
  - Eventually translating it into normal reads
  - The server builds up its unique (composite) response and sends it back
  - The client unpacks it and puts the individual subchunks into memory buffers for the app to access them
  - Yes, it needs a complicated machinery which also has to be very efficient
  - Better to hide it from the application's perspective

About vectored reads

- What the disk sees is more or less unchanged
  - Still the same stream of requests, eventually sorted
  - So, same number of requests, same number of interrupts
  - It is believed to be somehow more efficient at the disk level
    - But still very controversial
    - Request streams from several clients will interleave, leading to a completely random pattern

- At the OS/disk side there are the primitives to do that:
  - If you do **`man readv`** you can see that that one is not what we are talking about
  - Try instead **`man lio_listio`**
  - So there is no easy way to aggregate requests towards the disks
    - At least, we need to implement one more complicated machinery
    - And the creator of the application should better be shielded against all the technicalities
    - The disks heads will still have to move
    - Hence, in principle, we cannot cut the disk latency, or not too much
    - Nor the disk thrashing due to an excessive load

This is not what we need

```
ssize_t
     readv(int d, const struct iovec *iov, int iovcnt);
(. . .)
```

Readv() performs the same action, but scatters the input data into the iovcnt buffers specified by the members of the iov array: iov[0], iov[1], ..., iov[iovcnt-1].

For **readv**(), the iovec structure is defined as:

```
struct iovec {
        char   *iov_base;  /* Base address. */
        size_t iov_len;    /* Length. */
};
```

Each iovec entry specifies the base address and length of an area in memory where data should be placed.

Readv() will always fill an area completely before proceeding to the next.

An example of the readv we need

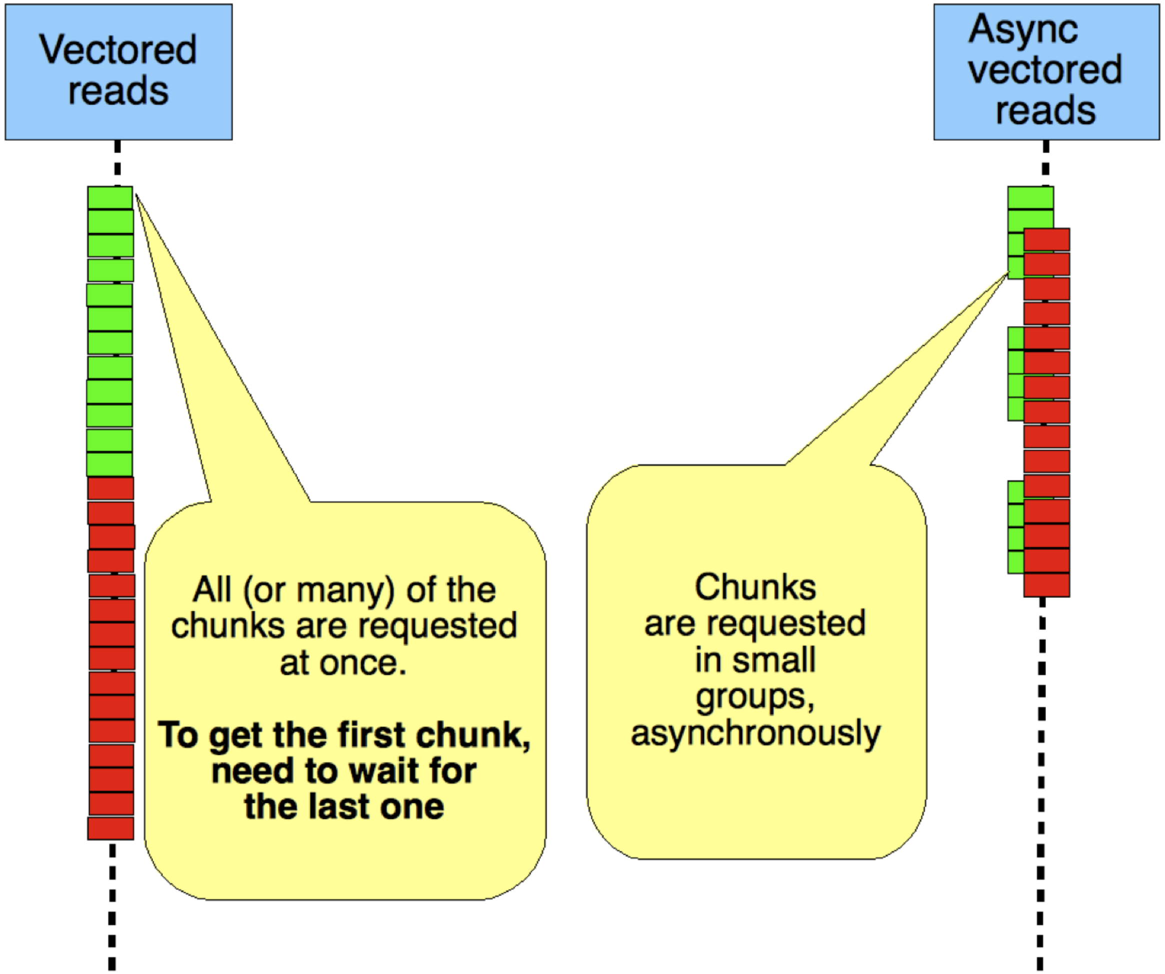
```
// Read multiple blocks of data compressed into a sinle one. It's up
// to the application to do the logistic (having the offset and len to find
// the position of the required buffer given the big one). If no error
// occurs, it returns all the requested bytes.
// NOTE: if buf == 0 then the req will be carried out asynchronously, i.e.
// the result of the request will only populate the internal cache.
// A subsequent read()
// of that chunk will get the data from the cache

kXR_int64 ReadV(char *buf, long long *offsets, int *lens, int nbuf);
```

No compromises

- We want the best of both worlds
- What about sending asynchronous vectored requests which are not too big?
  - Transferred in parallel
    - Hides the overall latency (network+disk)
  - Big enough to cut their network latency by e.g. 512 times
  - Small enough to avoid the serialization problem
      - Having to wait for the last chunk in order to process the first
  - This works
    - Not very easy to exercise seriously
    - This is the way ROOT works if used properly
    - Anybody else could do it in principle

Pure Readv vs. Async



An example

- Let's see how all this behaves in my laptop towards a robust data server over a 1Gb LAN
  - Nobody else using it (which is an optimistic situation)
  - The server has already cached all the data (optimistic situation = no disk latency, only network)
    - This is almost never true in the real world
    - Even better than the performance of SSDs
  - We use this case to see the difference between the various techniques
    - And this difference can be even bigger in the real case!
      - For the reasons we already described.

A practical evaluation: sync reads

```
./bin/TestXrdClient_read root://lxfsrc2802//cfs/fs10/fabrizio/h1huge.root 0 0 0 0 <
~/offsetlen_nurcan2.txt
Read style: Synchronous reads, ev. with read ahead.
.....--- Freeing buffer
Summary ----------------------------
$$$ starttime: 1.25414e+09
$$$ lastopentime: 1.25414e+09
$$$ closetime: 1.25414e+09
$$$ endtime: 1.25414e+09
$$$ open_elapsed: 0.013067
$$$ data_xfer_elapsed: 120.295
$$$ close_elapsed: 0.0252302
$$$ total_elapsed: 120.333
$$$ totalbytesreadperfile: 132851819
$$$ maxbytesreadpersecperfile: 1.10438e+06
$$$ effbytesreadpersecperfile: 1.10403e+06
$$$ readscountperfile: 95651
$$$ openedkofilescount: 1
```

- This tells us that the network latency is ~1.25ms per request
  - 120.333 / 95651 = 0.00125
  - Because we know that we have almost no disk latency here (everything is cached because we wanted it to be that way)

A practical evaluation: async reads

```
>./bin/TestXrdClient_read root://lxfsrc2802//cfs/fs10/fabrizio/h1huge.root 0 50000000 4 0 < ~/
offsetlen_nurcan2.txt
Read style: Asynchronous reads.
.....--- Freeing buffer
Summary ----------------------------
$$$ starttime: 1.25414e+09
$$$ lastopentime: 1.25414e+09
$$$ closetime: 1.25414e+09
$$$ endtime: 1.25414e+09
$$$ open_elapsed: 0.0132041
$$$ data_xfer_elapsed: 5.6057
$$$ close_elapsed: 0.0121732
$$$ total_elapsed: 5.63108
$$$ totalbytesreadperfile: 132851819
$$$ maxbytesreadpersecperfile: 2.36994e+07
$$$ effbytesreadpersecperfile: 2.35926e+07
$$$ readscountperfile: 95651
$$$ openedkofilescount: 1
```

- Looks interesting… from 120s to 5.6s doing the same sequence of reads
  - >20 times faster. Seems a very good optimization!
  - The latency has been “hidden” in this case

A practical evaluation: sync readv

```
>./bin/TestXrdClient_read root://lxfsrc2802//cfs/fs10/fabrizio/h1huge.root 0 50000000 1 0 < ~/
offsetlen_nurcan2.txt
Read style: Synchronous readv
<snip>
--- Freeing buffer
Summary ----------------------------
$$$ starttime: 1.25414e+09
$$$ lastopentime: 1.25414e+09
$$$ closetime: 1.25414e+09
$$$ endtime: 1.25414e+09
$$$ open_elapsed: 0.0133462
$$$ data_xfer_elapsed: 2.13707
$$$ close_elapsed: 0.00490284
$$$ total_elapsed: 2.15532
$$$ totalbytesreadperfile: 132851819
$$$ maxbytesreadpersecperfile: 6.21653e+07
$$$ effbytesreadpersecperfile: 6.16389e+07
$$$ readscountperfile: 95651
$$$ openedkofilescount: 1
```

- What? 120s to 2.1s doing the same thing?
  - 60 times faster!
  - In this case the latency has been “cut” by aggregating reads

A practical evaluation: Async readv

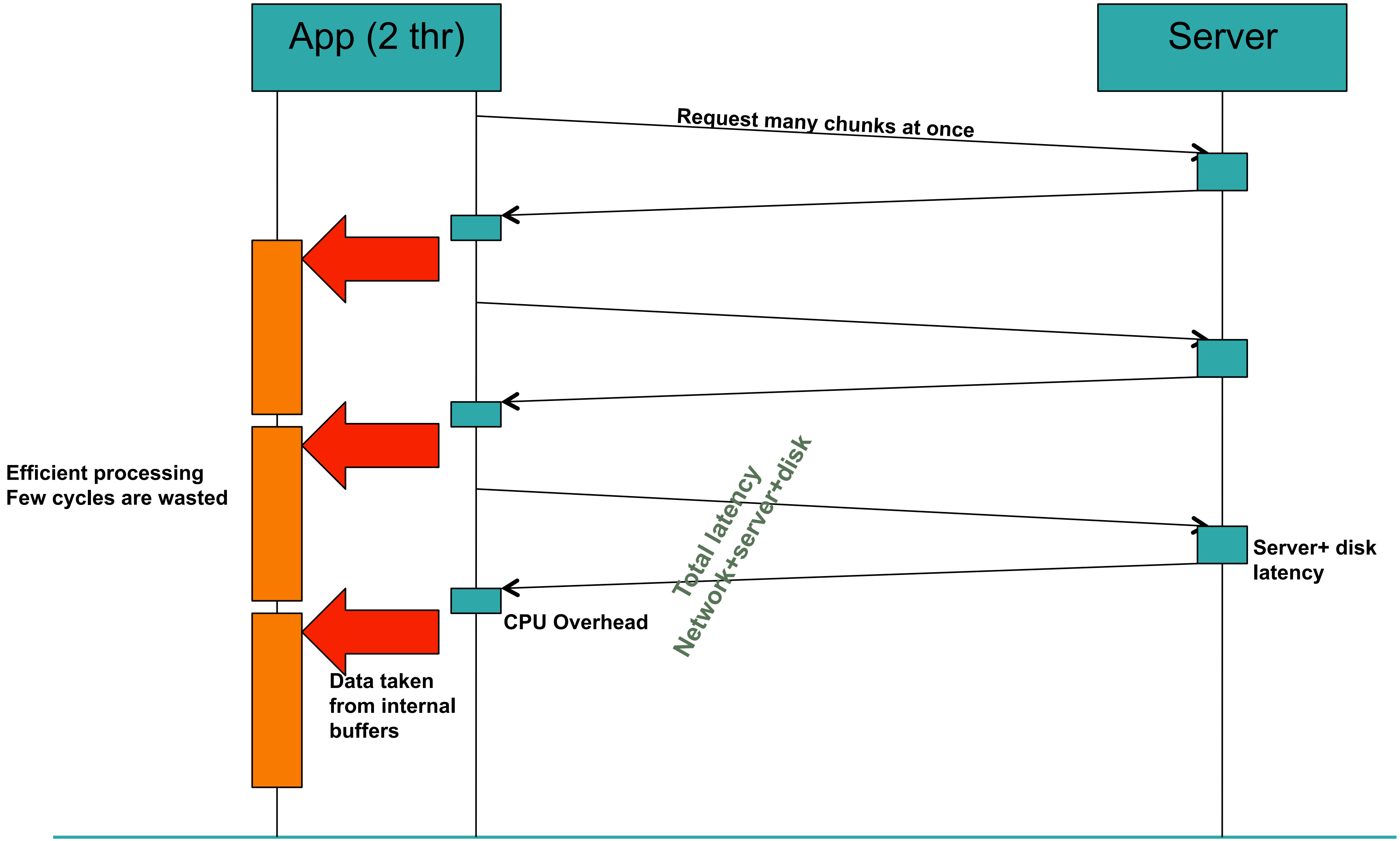
```
>./bin/TestXrdClient_read root://lxfsrc2802//cfs/fs10/fabrizio/h1huge.root 0 50000000 3 0 < ~/
offsetlen_nurcan2.txt
Read style: Asynchronous readv.
<snip>
--- Freeing buffer
Summary ----------------------------
$$$ starttime: 1.25414e+09
$$$ lastopentime: 1.25414e+09
$$$ closetime: 1.25414e+09
$$$ endtime: 1.25414e+09
$$$ open_elapsed: 0.0133181
$$$ data_xfer_elapsed: 2.6645
$$$ close_elapsed: 0.00909686
$$$ total_elapsed: 2.68691
$$$ totalbytesreadperfile: 132851819
$$$ maxbytesreadpersecperfile: 4.986e+07
$$$ effbytesreadpersecperfile: 4.9444e+07
$$$ readscountperfile: 95651
$$$ openedkofilescount: 1
```

- Apparently just a bit slower then the previous.
  - 2.7s. In this case the latency has been “cut” and then “hidden”
    - Hiding it needs a little more CPU
    - Remember that here we only read, no time is spent in processing anything
    - Hence, there is no place to hide the latency under
  - Here there is an advantage which we like.
    - And we are going to discuss it.

A practical evaluation: sync vs async

- Let's make the appication "think" and process the data (still in my laptop, reading from a robust server)
  - i.e. using CPU cycles between reads, e.g. 10ms every 100 reads
  - And the scenario becomes more clear:
    - Silly sync reads: 153s , CPU usage=30% (please remember that this technique is what 3 of the 4 LHC experiments use)
    - Sync readv: 12.7 seconds, CPU usage>100%
    - Async readv: 10.5 seconds (2.2 secs less), CPU usage>100%
      - So, why is this now faster? Before it was a bit slower.
      - Because it:
        - Cuts the latency by a factor by aggregating reads, then,
        - Transfers the next bunch of chunks while the app is crunching numbers
        - So it also can use CPU cycles from another CPU core.

What's happening



A practical evaluation: sync vs async

- One more side effect is that this works amazingly well also in WAN if there is enough bandwidth
- Yes, because now, finally the only limiting factor left is bandwidth. For that, we need only:
  - A sane network design
  - Very powerful disk systems to give enough throughput
  - A lot of money
- But now we can use it productively, in both LAN and WAN.
- And, YES, when we are here, all the other CPU-based optimizations start making a lot of sense
  - All that effort is worth the time. Do it.

Last considerations

- Doing asynchronous things generates in general more CPU overhead
  - If this is shorter (in time) than a latency hit then we gain anyway
- A pure readv is very efficient
  - But to process the first requested chunk we must wait for all of them to come
    - And they come serially (very fast, however)
- Very often, for very sparse patterns readv is a very good choice
  - For less sparse patterns often not quite
  - But typically analysis applications generate extremely sparse pseudo-random patterns

Last considerations

- When multiple users hit the same disk, that disk ‘sees’ a truly random pattern.
  - Hence its performance decreases
  - It can decrease really a lot
  - There’s not much that we can do for now
    - Buy better disks
    - Avoid RAIDs if you can (they move more heads/drives to do the same job)

But… Wait… Is that possible?

- In both cases we are requested to know the future
  - In the form of the data chunks which will be needed
  - The client API and the servers must support the techniques of course
  - But… Is knowing the future a realistic thing?
  - How can an app which needs the (n)th chunk also suggest that it will need the (n+X)th ?

Feeding the communication pipe

- Basically there are two techniques to make it possible:
  - Guessing the future
    - Applying statistics-based ideas
      - Typically Read Ahead/Prefetching. We read data in advance, hoping that it will be useful to a sufficient extent for the next requests.
  - Knowing the future exactly
    - A list of (offset, length) which will be needed
    - This is what any cp-like program can do (read everything!)
    - This is what ROOT can do (TTree/TTreeCache)
    - But not every app uses ROOT, and sometimes, if they use it, they do not use it in that way ☹

Feeding the communication pipe

- We don't need necessarily complicated things
- For example, we know what our forward reader app needs
  - 1KB every 10KB
  - So we might, in principle
    - Produce this list of chunks at the beginning
    - Fire it to the disk
    - Loop on the results
  - But the devil is in the details
    - The sw machinery to do this is not so simple, e.g. we must remember that from a 10 lines program we might go to 100 with only such a simple scenario

You also need the right API

- In the POSIX calls there is a way to "suggest" actions to be done in parallel
  - O_NONBLOCK
  - The app must deal directly with that complexity
    - And explicitly keep track of what's pending, what arrived, what will be requested
- Here we are using the Xrootd client, which was built to do that
  - And the complexity was hidden in it
  - I am not aware of other similar APIs (probably my fault)
    - Even if the principles are 20 years old

Read ahead

- In simple words it means:
  - "Read something which will likely be useful in the near future, and store it somewhere for fast lookup"

- There's a caveat:
  - We rely on statistics, not on exact knowledge
  - We could read a lot of useless data.
  - We could miss what the app really needs
  - To be statistically significant, we need a lot of memory to keep it, except in the trivial cases (e.g. purely sequential)
  - These are the limits of this technique in its common implementations.

Read ahead: the simplest strategy

- Given a read request to satisfy, trim it to a bigger block and store the whole result
  - This is what typically the OS does for disks.
    - The request is enlarged in order to cover the minimum number of pages which contain the needed data
    - The request can also be enlarged much more, typically forward (even tenths of megabytes)
- It's an algorithm like many others
  - Which should be smart enough to AVOID requesting the same data more than once
    - Not very easy

Read ahead: another strategy

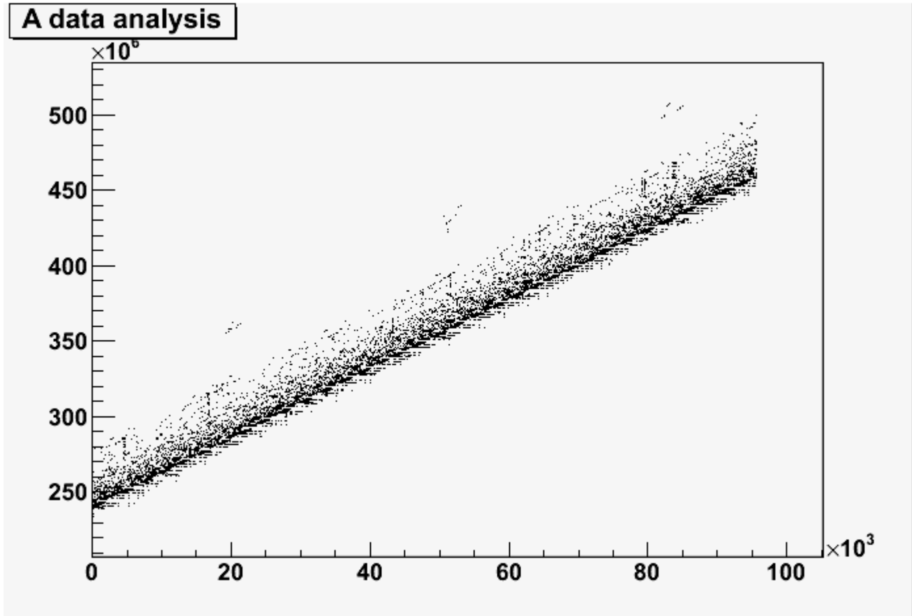
- Given a read request to satisfy, make sure that the data in the internal buffers arrives up to the location offset+N
  - Eventually requesting what's missing as an unique big block
    - And purging something else
- We can call this also "look ahead"
  - The last byte requested in advance is always at offset +N
- Very efficient for sequential access
  - The data stream can never stop

Read ahead: one more flavour

- We compute the average offset of the last accesses
- We try to keep in memory a “window” of data around this average
  - Hoping that the next accesses will hit inside it
- The window slides forward with the average offset
  - Allowing some accesses to be outside it
  - Reading (ahead) in steps of 1MB
  - Dropping the block with the least offset
- Good for not so sparse patterns which slowly proceed through the file length

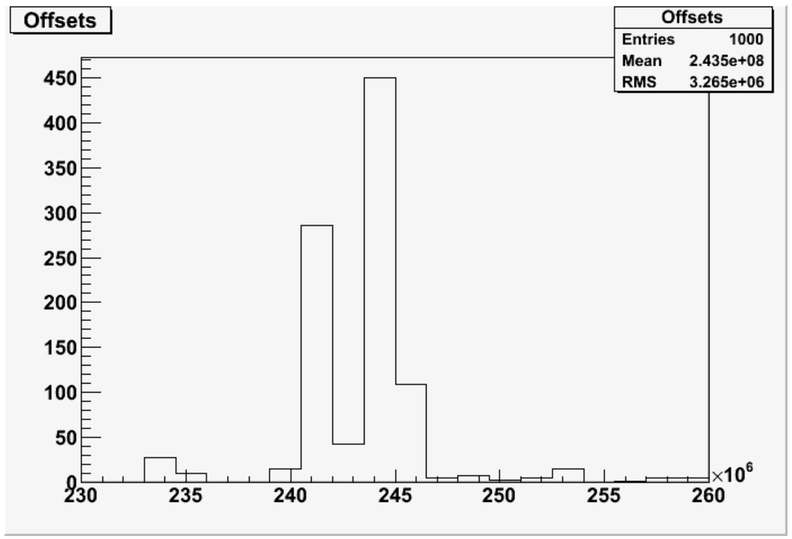
A snippet of a data analysis (ATLAS AOD)

- Index of the read on the X, offset on the Y
- It's "random", but not quite
  - Even by looking at it we can almost predict where it goes



A snippet of a data analysis (ATLAS AOD)

- A histogram of the first 1000 offsets is even more suspect
  - With a buffer holding data from 235M to 255M we can accommodate the majority of the (very small) first 1000 reads



Do they work in practice?

- Sometimes yes, sometimes no
  - They can gain a lot or loose, depending on the case or on the class of applications
  - We measure their efficiency like a cache:
    - Miss rate: the ratio between the number of times a chunk is correctly prefetched with the number of times it has to be requested
    - Byte overhead: how many useless bytes are read
  - For a copy-like sequential read
    - Missrate=0 and overhead=0
  - For a generic application it depends
    - A hit saves one interaction (hence, one latency hit also in the disk)
    - The byte overhead (given the maximum throughput) must be lighter than the time (and resources) saving due to the hits
    - The application consumes more CPU, because of the overhead due to the internal bookkeeping and calculations
    - Keeping track of what's outstanding is not cheap