Latencies and data access. Boosting the performance of distributed applications.

Fabrizio Furano* Peter Elmer† Andrew Hanushevsky‡ Gerardo Ganis§

Abstract

The latencies induced by network communication often play a big role in reducing the performance of systems which access big amounts of data in a distributed environment. The problem is present in Local Area Networks, but in Wide Area Networks is much more evident. It is generally perceived as a critical problem which makes very difficult to get access to remote data. However, a more detailed analysis on the access pattern of the involved applications can be used to understand the characteristics of the stream of the data requests, and develop techniques to optimize it. This work started from the analysis of the access patterns of the BaBar experiment's physics analysis data, but the methods and the results can be applied in other computing environments as well. We show how the exploit of caching and asynchronous prefetching techniques is able to enhance the performance of such kind of applications in Local Area Networks, and is able to lower the total latencies for Wide Area Networks data access of an order of magnitude. Moreover, the ability to process file open requests in parallel can be a very interesting performance enhancement for applications which need access to many files at once. Such general techniques have been implemented in the client side of the xrootd data access system, which, for its performance and its fault tolerant architecture, showed itself as an ideal testbed for such a kind of enhancements.

INTRODUCTION

The problem we address in this paper is related to the performance of data analysis jobs which access an external data repository. The typical situation of data centers offering computing services for HEP experiments usually comprehends computing facilities and storage facilities. In this environment, a simple application which has to analyze the content of some files, typically will:

- open the files it has to access
- cycle through the stored data structures, performing calculations and updating the results
- output in some way the final results

From here, we assume that those data access phases are executed sequentially, as it is in most data analysis applications. These considerations refer to the concept of file-based data store, which is a much used paradigm in the HEP computing; however, other data access methods can be affected by the same performance issues. The key issues we try to address are:

- A single file open can be a long operation, expecially (but not only) if it involves the file staging from tape to disk. Hence, how can an application deal with the fact that the long wait would be multiplied by the number of the files to open before starting computing? As a simple example, if we consider an application opening 1000 files, each producing a two second delay, the application will start its computing phase in 2000 seconds if no optimizations are implemented.
- The computing phase will be composed by a huge number of interactions with the external (or internal) data store. Hence, how can an application deal with the fact that even a very short mean latency (e.g. 0.1 milliseconds) would be multiplied by the huge number of interactions (e.g. 10^7). With the data given as an example, the involved CPU will waste over 1000 seconds in waiting for data to come.

The latter topic can also be considered a serious issue which makes impossible for data analysis applications to access remote repositories. However, there are situations where this is not true. A trivial example of such a situation is when the application does not need to read all the content of the files it opens. Another example will be addressed in the subsequent paragraphs, where we discuss a way to enhance the performance of data reads even in presence of high network latencies. The former topic (the multiple application-side serialized file opens) will also be shortly addressed, to present the results of the most recent implementations of algorithms which parallelize the open requests without having to change the application code, which is supposed to:

- open all the files in sequence,
- access the data only when all the files are opened.

READ REQUESTS AND LATENCY

Figure 1 shows that, in a sequence of read requests from a client to a server, the transmission latency affects the requests twice, and is located before and after the server side

* INFN sez. di Padova, furano@pd.infn.it
† CERN, peter.elmer@cern.ch
‡ Stanford Linear Accelerator Center, abh@slac.stanford.edu
§ CERN, ganis@cern.ch

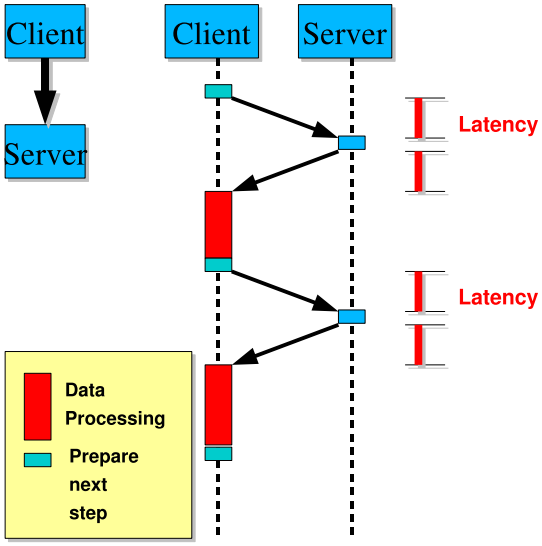


Figure 1: Role of the latency in read operations.

computation. If we suppose that the latency is due to the network, and that the repository is not local to the computing client, the value of the latency can be even greater than 70-80 milliseconds. With a latency of 80 milliseconds, an application in Padova requesting data from a server at SLAC, for example, will have to wait 160 seconds just to issue 1000 data requests. However, the work done at INFN Padova addresses the problem in a different way. If we can get some knowledge about the pattern (or the full sequence) of the data request issued by the application, the client side communication library can, in general, request in advance the data for the future requests, and store portions of it in memory. Moreover, the future data requests can be issued in parallel with respect to the present ones, getting the advantage of having, in any phase of the computation, outstanding data which has a high probability of containing the responses of the future data reads.

ACCESS PATTERNS

Figure 2 shows a summary of the data access pattern of a BaBar analysis job which generated about 10^7 reads. For each read request generated by the analysis application, we calculated the displacement (in bytes) from the offset of the previously issued read request. The plot in Figure 2 shows the frequency of the so calculated byte displacements. What is more evident is that:

- a sequential access would have been a pure spike at the zero coordinate, hence the access pattern is not purely sequential;
- a random pattern would have been equally distributed, instead we see that the majority of the displacements lie in the [-500000, 500000] interval.

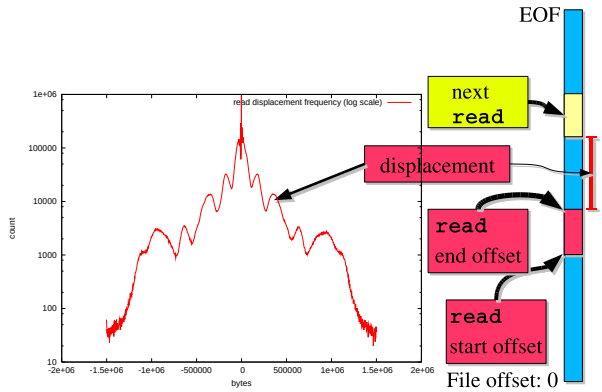


Figure 2: Frequency of the read displacements in a BaBar analysis job.

EXPLOITING THE ACCESS PATTERNS KNOWLEDGE

The main result of the simple previous data pattern analysis is that it is very likely that a caching mechanism would be very effective in reducing the number of the data requests which need to cross the network to reach the server. However, we decided also to exploit the predictability of the data requests, by implementing a read ahead schema. Hence, the basic idea is that, to lower the total latency, at the client side communication library (*XrdClient* or *TXNetFile*) we exploited these techniques:

- a memory cache of the read data blocks. The default size of the cache is 3 Megabytes, with a block size of 128 KBytes;
- the client is able to issue data requests for subsequent blocks, which will be carried out in parallel
- the client exposes an interface for the application to be able to inform it about its future data requests
- the client keeps track of the outstanding requests, in order not to request twice data which is in transit.

Figure 3 shows a simple example of how such a mechanisn works. The client side communication library can speculate about the future data needs, and *read ahead* some data at any moment. The caching mechanism drastically reduces the number of requests which have to be forwarded to the server, thus reducing the overall data transfer latency. For more information about caching and prefetching, see [2].

Figure 4 shows the three typical scenarios for remote data access. The most obvious one is the one at the left, where the needed files are transferred locally before starting the computation. The middle one refers to the application paying for the network latency for every data requests. This makes the computation very inefficient, as discussed. The rightmost part, instead, shows that, if the data

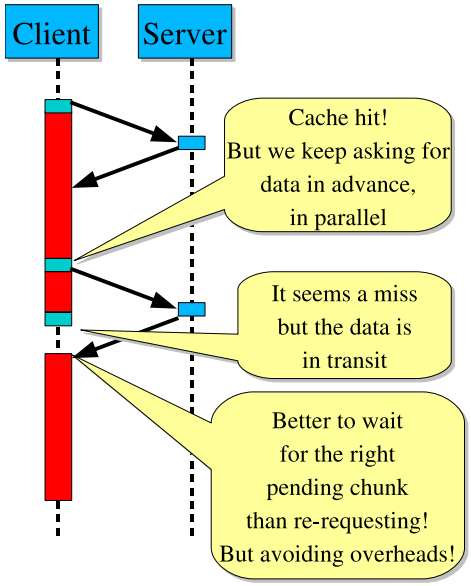


Figure 3: Prefetching and outstanding data requests.

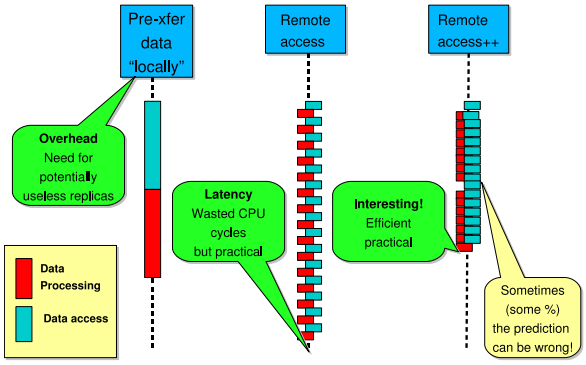


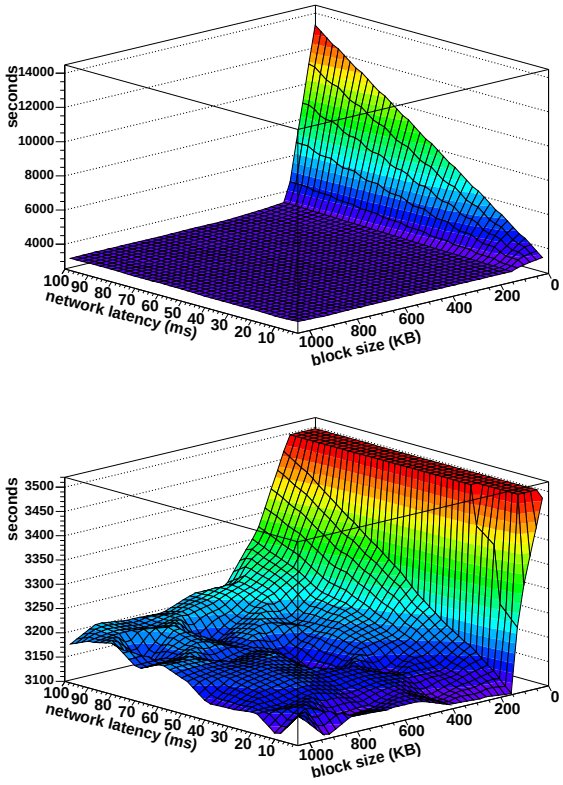
Figure 4: Three scenarios for remote data access.

caching/prefetching is able to keep the cache miss ratio at a very low level, the achievable result is to keep the data transfer go in parallel with the computation, but a little in advance with respect to the actual data needs. Some measurements and real world tests showed that, for that kind of BaBar job, the miss rate can be even between 1% and 5%.

TOWARDS A REAL WORLD EVALUATION

Before starting implementing the complexities of the caching, prefetching and cache placeholders techniques in the client side of the *xrootd* system, we decided to evaluate the achievable performance using a simulation based approach. Some results are visible in Figure 5, which shows the duration of a BaBar analysis job depending on the network latency and the cache block size (where 0 for the block size means no caching/prefetching at all).

In Figure 5 we can see that, with all the optimizations



Cache size: 3M

Figure 5: Simulated duration for a BaBar analysis job, depending on network latency and cache block size.

switched on, there is not a huge difference between the simulated job duration in the case of a LAN (latency near to 0) and a WAN (latency up to 100ms), provided that they can sustain the minimum needed throughput for this analysis job, which was of 350KBytes/sec. Instead, we can see that the time needed to analyze the data (including the CPU time) drops quite sharply from 12000-14000 seconds down to 3000-3300 for a cache block size (and read ahead block) greater than 200KBytes and a network latency of 80-100ms. From this, we concluded that the said techniques were worth implementing, and now are available in the *xrootd* UNIX reference client *XrdClient* and in its ROOT 5 embedding, named *TXNetFile*.

Figure 6 instead shows the result of a test of the algorithm named *Parallel Open*, implemented in both *XrdClient* and *TXNetFile*. The test has been performed towards the Kanga cluster at the SLAC (Stanford Linear Accelerator Center) laboratory. The test application performs the following actions:

- **open** 1576 files by invoking the Open request for each file to open,
- **read** three small data chunk from each file,
- **close** all the files.

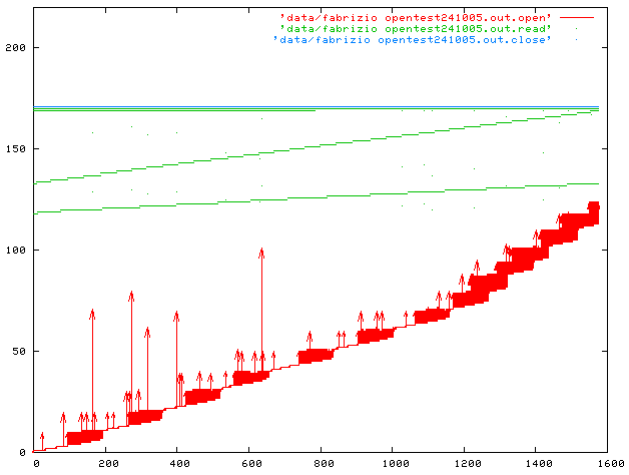


Figure 6: Behavior of an application opening 1576 files at once, with the Parallel Open feature.

The diagram in Figure 6 shows in the Y axis the time when each action started and ended. Every X coordinate instead refers to a single file. The typical simple implementation of an Open-like primitive would have been synchronous, i.e. the (i+1)th file open request starts when the i-th file open request has finished. From the diagram instead we can easily see that the system carries out many open request in parallel. This is visible because, in a given moment (i.e. at a given Y coordinate) many Open request are in progress (the vertical red lines), and the total time needed to open 1576 files was around 110 seconds, comprehending the time needed to stage many of them. This gives much better response times for applications in the need of opening many files at once before starting their computation.

REFERENCES

[1] A. Hanushevsky, A. Dorigo, P. Elmer, and F. Furano. The next generation ROOT file server. In CHEP'04 - Computing for High Energy Physics. CERN, 2004.

[2] Patterson R. H., Gibson G. A., E. Ginting, Stodolsky D., and Zelenka J. Informed prefetching and caching. In Proceedings of the 15th ACM Symposium on Operating Systems Principles., 1995.

[3] Andrew Hanushevsky. Hyper-scaling data access: understanding XROOTD data clusters. In ROOT Workshop 2005. CERN, September 2005.

[4] Fabrizio Furano. Large Scale Data Access: Architectures and Performance. Ph.D. thesis TD-2006-1, Universita' Ca' Foscari Venezia, 2006.