Benchmarks Study for OpenLSD

Author: Frédéric Brégier under LGPL

Website: http://openlsd.free.fr/en/OpenLSD.html

Performance benchmarks

Several benchmarks were done on OpenLSD version 0.9.4 pre-version of 1.0 and the final 1.0 version.

All benchmarks were executed using the following hardware:

Function	Hardware	CPU	Memory	Disk	OS	Software
Application Server	PSeries Power5 IBM	8 CPU	24 GB	1 TB SATA (slow) in 2 Gbit SAN	AIX 5.3 64B	JDK 1.5 IBM SR6 64B
Database Server	PSeries Power5 IBM	6 CPU	24 GB	200 GB FC (fast) in 2 Gbit SAN	AIX 5.3 64B	Oracle 10G 64B
Web Servers	Blade Center Intel Xeon	2 CPU	8 GB	72 GB internal	Suse SLES 10 64B	JDK 1.5 IBM SR6 64B

Three kinds of benchmarks were done:

- Massive import of documents (injection of documents into the OpenLSD Server) (0.9.4 and 1.0 versions)
- Consistency check of documents already in the OpenLSD Server
- Massive web retrieve of documents (from Web applications using the OpenLSD service from OpenLSD Server)

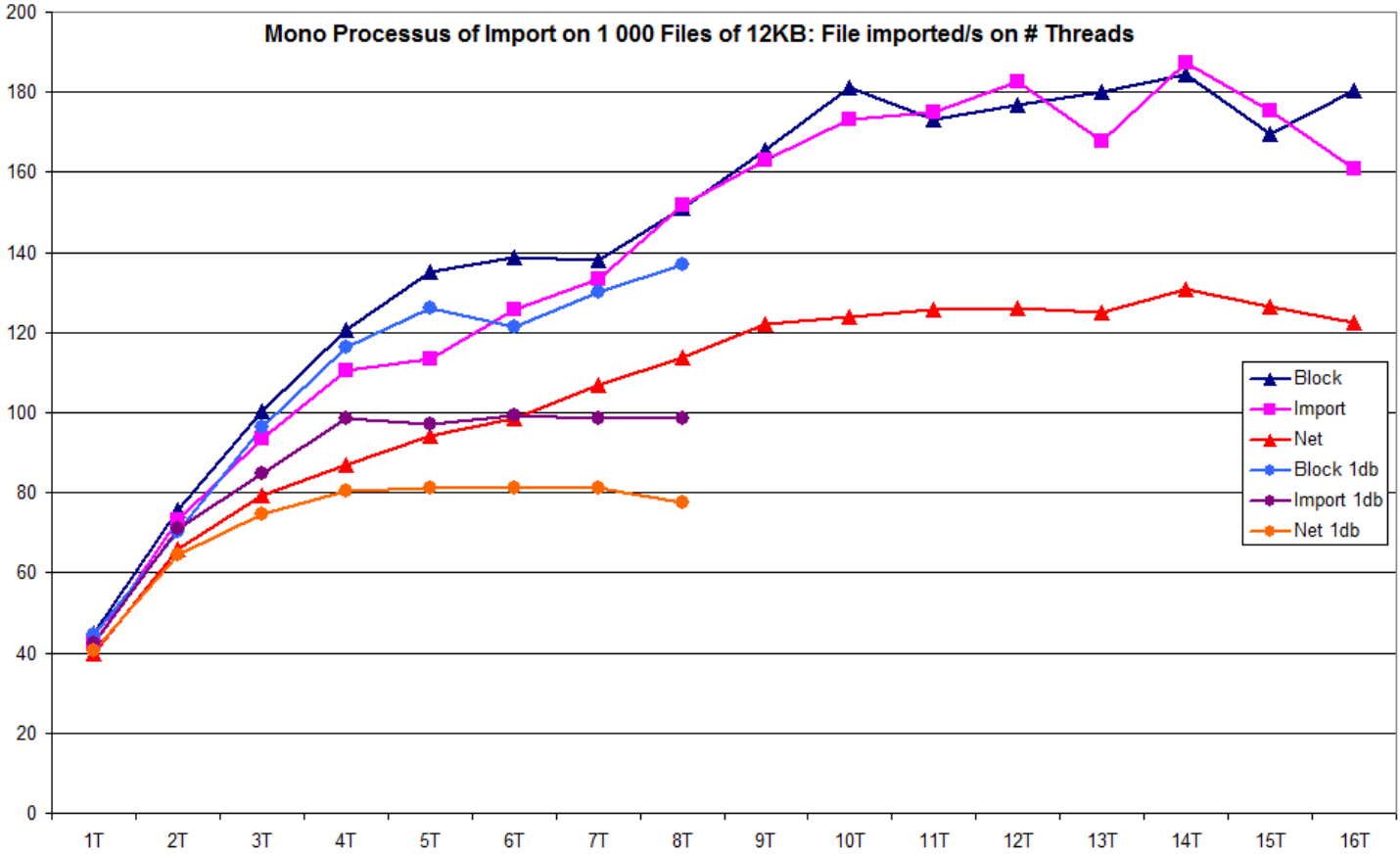
Massive Import of Documents

For the massive import of documents, several cases were studied.

1 000 documents import test case

The first one below shows the case of one process of import (one execution at a time) for relative small number of documents (1 000 documents of 12 KB each). Each process can use 1 to n threads that concurrently import those documents, in order to sustain performances for one bunch of documents (1 000 documents).

- 2 kind of importers were used: one using a net file transfer (Net versions), and one using a local file transfer (file is already on the OpenLSD Server, with Block and Import versions).
- 3 kinds of logic of imports were used: by Block (several documents at a time in each thread), mono document (1 document at a time in each thread: Import and Net).
- 2 options were tested: one using one database connection for each threads (16 threads meaning 16 database connections), and one using one shared database connection among all threads (1 db versions).



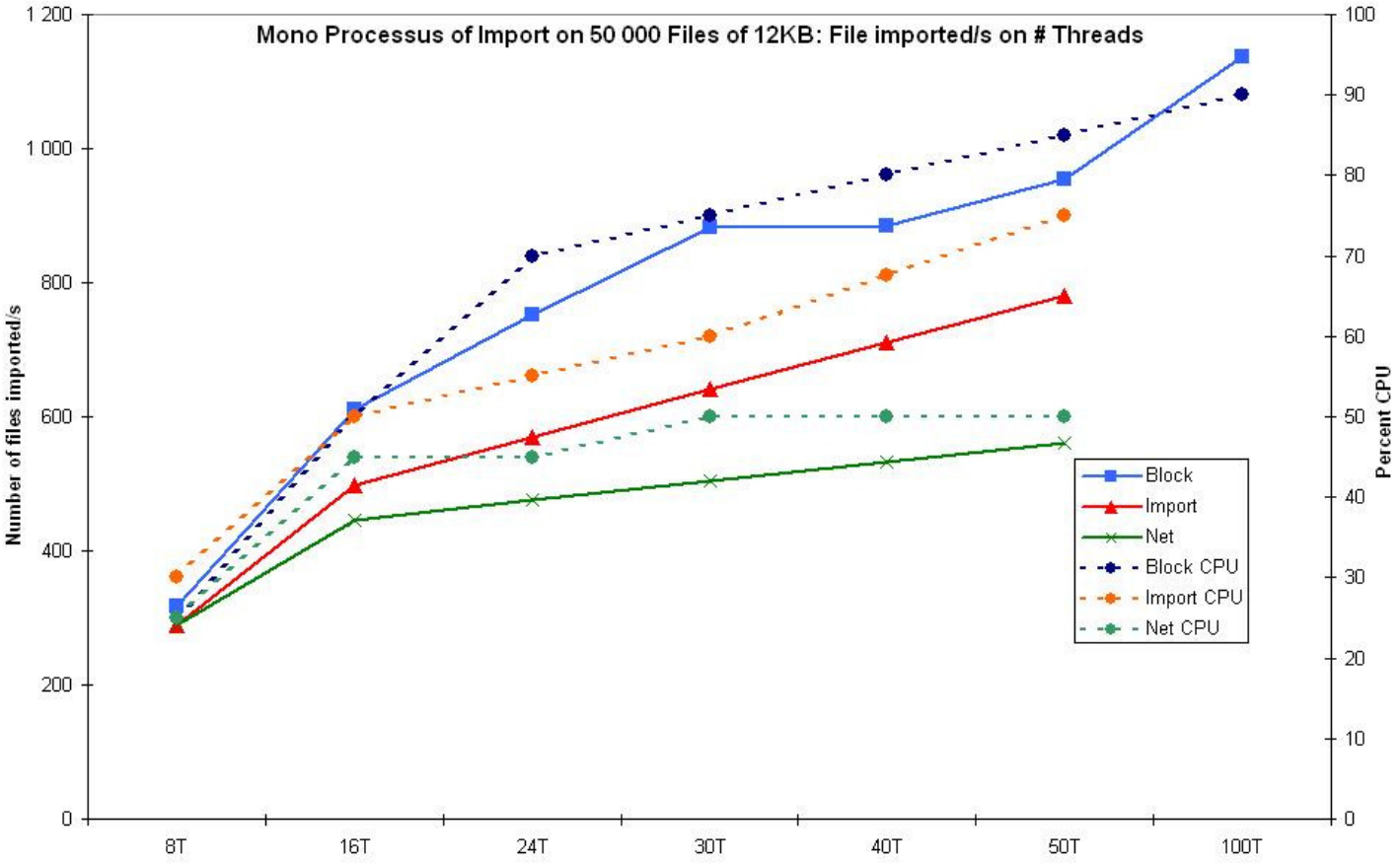
- The first lesson is that obviously network import is slower than local import (around 30% slower).
- The second lesson is that block versions are in general faster than unitary versions, at least up to a certain number of threads, were the performances start to be limited.
- The third lesson is that the more threads we have, the more efficient is the import, with a stagnation observed here around 10 threads for 1 000 files (100 files by thread). The global efficiency is about 50% with 10 threads (5 times faster with 10 threads than with only 1 thread).
- The fourth lesson is that while multiple database connections use more resources, they enable the best efficiency. With one shared database connection, the performance starts to be limited around 4 threads.
  In multiple database connection, the lock procedure is attached to the database connection itself (the database makes everything parallel and correct according to concurrency correctness).
  In shared connection, the lock must be done also in the JVM since multiple threads shared this connection, so as the lock, and to prevent wrong situation, each thread must use a global lock in the JVM to ensure the correctness of their job.
  So the reason of the bad performance of the shared connection version is due to this JVM global lock which limits the effect of the multi-threads approach.

50 000 documents import test case

The second test below shows the case of one process of import (one execution at a time) for a huge number of documents (50 000 documents of 12 KB each). Each process can use 1 to n threads that concurrently import those documents, in order to sustain performances for one bunch of documents (50 000 documents).

- 2 kind of importers were used: one using a net file transfer (Net versions), and one using a local file transfer (file is already on the OpenLSD Server, with Block and Import versions).
- 3 kinds of logic of imports were used: by Block (several documents at a time in each thread), mono document (1 document at a time in each thread: Import and Net).

We present also the CPU graphs for the application server on the same picture with dotted curves.



- The first lesson is the same than the previous graph were Net version is slower (around 50%) than local versions. However the performance is quite good (around 50 MB on network link) with a CPU average keep under 50%. The local versions get superior efficiency but are limited by CPU usage (almost 90% with block local mode on the higher efficiency result and around 70% on local simple import).
- Again, the second lesson is that block version confirms its capacity to ensure the highest performance. The CPU usage is then the limit on this kind of version.
- The second lesson is when we compare the beginning of this graph with the previous one (1 000 documents up to 16 threads). We can see that performance still continue to improve since we get performances up to 3 times better on the same number of threads (8 threads from 140 to 300 documents by second and 16 threads from 190 to 600 documents by second). Increasing the number of threads gives more efficiency, with a global efficiency up to 66% with 32 threads (850 doc by second / 32 threads against 40 docbs / 1 thread) (around 1500 documents by thread), and still 30% of efficiency with 100 threads (1150 docbs / 100 threads).
- Finally, we can see that CPU usage follows the efficiency curves.

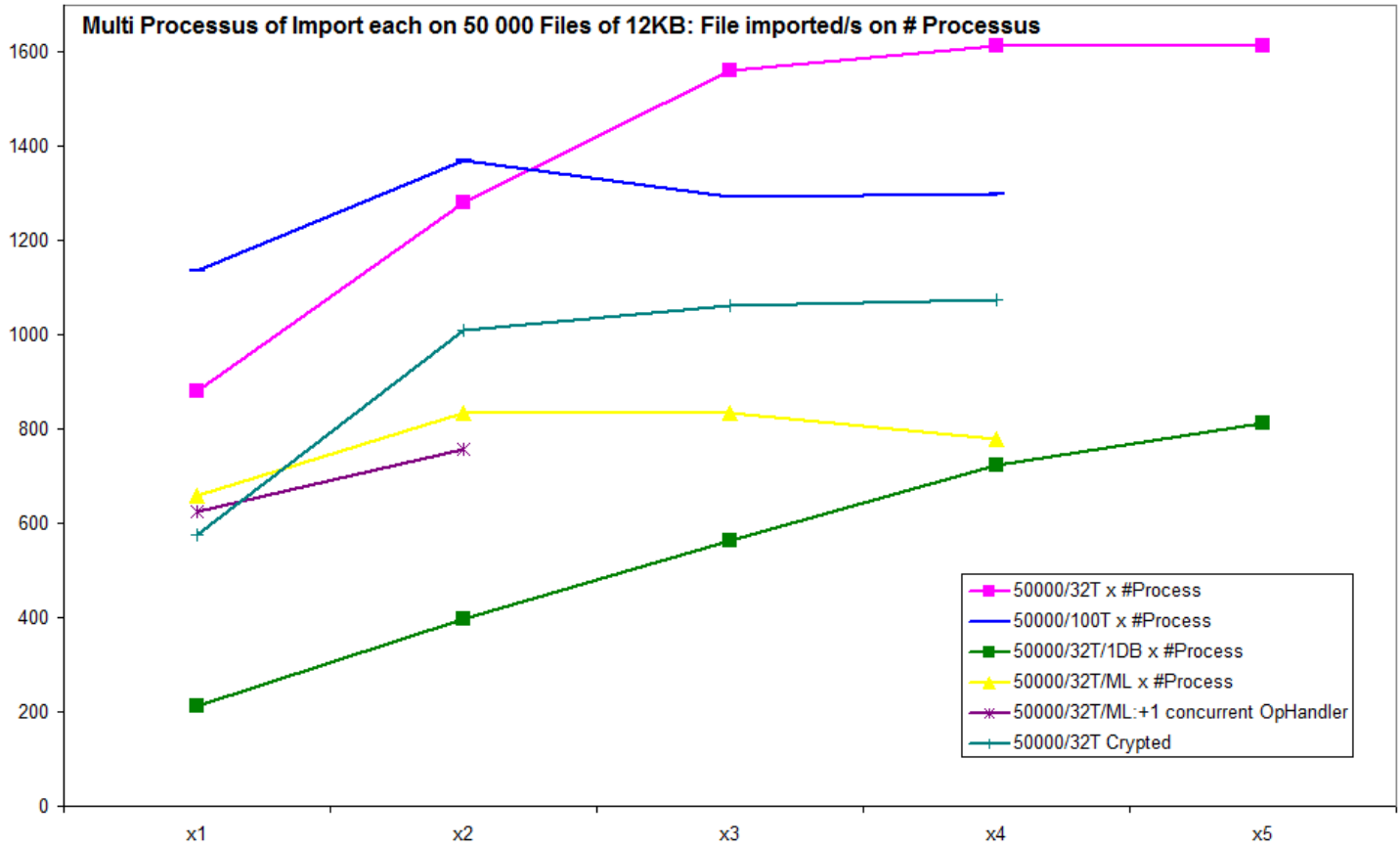
50 000 documents import test case with multiple imports

The third test below shows the case of multiple processes of import (multiple executions at the same time) for a huge number of documents (50 000 documents of 12 KB each for each process of import). Each process can use 1 to n threads that concurrently import those documents, in order to sustain performances for one bunch of documents (50 000 documents).

We used mainly 32 threads versions since it seems from the previous tests that for 50 000 documents it was the best compromise in efficiency. However we include also one version using 100 threads.

- Only Block versions (several documents at a time in each thread) were used.
- 2 database options were tested: one using one database connection for each threads (32 threads meaning 32 database connections), and one using one shared database connection among all threads (1 db versions).
- ML (Multiple Legacies) versions are also tested, one that only set the operations to do in the database for the Op Handler but without the Op Handler running to estimate the cost of those operations, and one that runs also the Op Handler in the same time, so running the replication at the same time.
- 2 versions were used on Legacy encryption support aspect: one using no encryption at all, and another one using encryption of files in the Legacy.

Name	Number of Threads	Multi/Mono DB	ML Support	Encryption
50000/32T	32	Multi	No	No
50000/100T	100	Multi	No	No
50000/32T/1DB	32	Mono	No	No
50000/32T/ML	32	Multi	Yes, alone	No
50000/32T/ML+1 Op Handler	32	Multi	Yes with Op Handler	No
50000/32T Crypted	32	Multi	No	Yes



- CPU usages are not included in order to keep some readability.
  - From 3 concurrent processes, all versions, except ML one, were at 95% of CPU usage (from 60% at 1 process as in previous tests and up to 95% CPU at 5 processes).
  - ML version was around 60% at all stages.
  - 1DB version starts from 25% and up to 87% at 3 processes and stays around this level after.

  From the database server point of view, all versions except 1DB are getting almost 90% percent CPU usage from 3 concurrent processes, while with 1 DB it increases slowly.
- Starting from 3 concurrent processes, no version performs better with more processes, except the 1DB version since the CPU usage was quite low before. This shows the interest however of the 1DB version when a lot of processes run at the same time, since it keeps the CPU usage low and still get good scalability (77% with 5 processes compared to 1 process).

  Other versions get around 60% of scalability until 3 processes and decrease around 36% at 5 processes. However 1DB versions are at least 100% less efficient than others.

- Comparing the 32T and 100T versions, we can see that 32 threads version gets better performances. This is unsurprising since we see before that 32 threads were the best compromise on 50 000 documents.
- The crypted version is about 66% slower than unencrypted one, due to higher usage of CPU and not so efficient filesystem access (write is done through the encryption package).

Globally, we can conclude that the best performance can be reach using a compromise between the number of threads and the database options. On low number of files (less than 1000), one must ensure at least 100 documents by thread. On huge number, one should not get more than 32 concurrent threads.

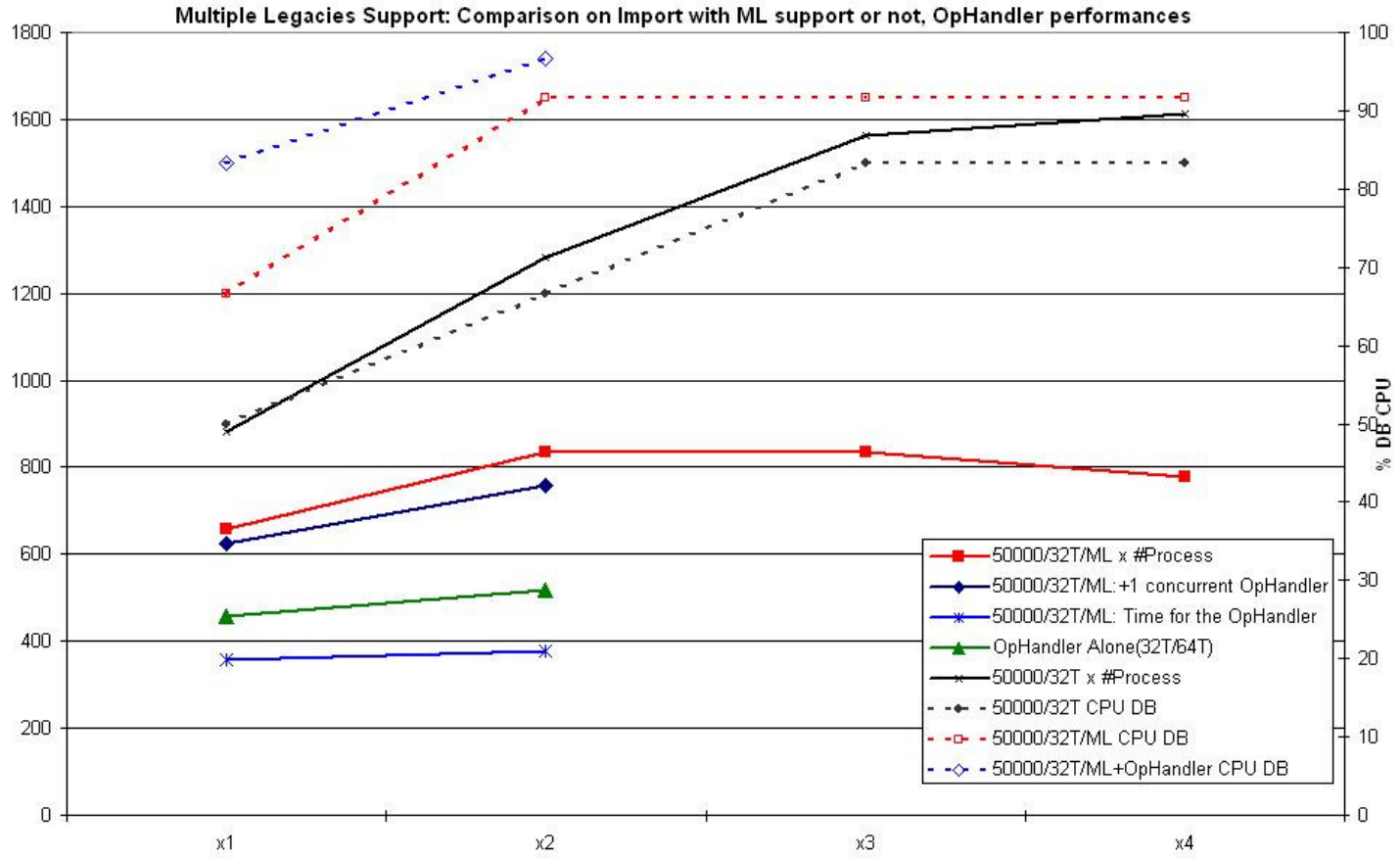
- Also, if multiple processes can be huge (more than 5 processes at the same time), one could consider to use the 1DB option since it allows better scalability but at the price of lower efficiency on each process.
- Another lesson is that even in huge CPU usage, the system is stable (no error while at almost 95% CPU on both application server and database server).
- Finally, we can see that the ML support decreases the global performance (about 50%) but quite efficient however (more than 800 documents by second). Having the Op Handler running at the same time gives only 10% of lower performance, due to database concurrent accesses and CPU usage, which is quite efficient.

50 000 documents import test case with ML focus

Now we focus on Multiple Legacies support efficiency.
The fourth test below shows a focus on Multiple Legacies support efficiency with multiple processes of import (multiple executions at the same time) for a huge number of documents (50 000 documents of 12 KB each for each process). Each process can use 1 to n threads that concurrently import those documents, in order to sustain performances for one bunch of documents (50 000 documents).
We used 32 threads versions as the best compromise in efficiency.
We present also the CPU graphs for the database server on the same picture with dotted curves

- Only Block versions (several documents at a time in each thread) were used.
- ML (Multiple Legacies) versions are tested, but we keep one version without ML support for comparison purpose.
    - We use one ML that only set the operations to do in the database for the Op Handler but without the Op Handler running to estimate the cost of those operations.
    - We use also one that runs also the Op Handler in the same time, so running the replication at the same time, with one special focus on Op Handler.
    - And finally we use one version with Op Handler running alone on 50 000 files with 32 or 64 internal threads.

Name	Number of Threads	ML Support	Op Handler running
50000/32T	32	No	No
50000/32T/ML	32	Yes, alone	No
50000/32T/ML+1 Op Handler	32	Yes with Op Handler	Yes
Op Handler with 50000/32T/ML	32	Yes with Op Handler	Yes
Op Handler Alone	32 (x1) / 64 (x2)	No	Yes Alone



- Comparing the graph of import with and without ML support, we can see that ML support brings at this time a limit on scalability, starting from 25% less efficient when only one process of import is running, then 35% with two processes running, and after the efficiency is no more increasing, being always around 800 documents by second, while without ML support it still improves its efficiency up to 1600 documents by second (4 processes).
  The main reason is that while ML support is inactive, the CPU usage on the database server is growing slowly from 50% to 85%, and with ML support this CPU usage is starting at 65% with 1 process and growing up to 95% from 2 running processes.
  Therefore the database server is the bottleneck of the ML support. Increasing the database server CPU number should improve the global performances of the ML support. That is the reason why we stop our tests up to 2 processes on the other tests.
- When the Op Handler is running at the same time than the ML Import, we observe only 10% of lower efficiency. This result is perfectly OK.
- If we focus on the Op Handler itself, we can see that its performance is about the half of the ML import when both are running at the same time (replication is concurrent with import) with a global performance of almost 400 documents replicated by second.
- When the Op Handler is running alone, after in this example an import of 50 000 documents in ML mode but without the Op Handler running, the efficiency is quite better than in concurrent mode (about 25% better up to 520 documents replicated by second).
  We see again the impact of CPU usage in the database server. When only the Op Handler is running, the database is less used, so the performance improves.
  We can see also that performance of the Op Handler is relative to the number of threads, with a similar limit of 32 threads since increasing to 64 threads only increase efficiency of 10%, as for the import in previous benchmark.

Globally, the ML support seems enough efficient for huge organization since it can sustain up to 800 imports by second as primary import while having up to 500 replications by second. The efficiency is directly related to the efficiency of the database server.

Consistency check of documents

In this test case, we measure the efficiency of different kind of consistency checks.

- CheckFile MD5: This program checks the consistency between the physical documents and their relative information in the database (existence and MD5 testing). The result is in number of checked files by second.
- CheckFile: This program is the same as the previous one but only checking existence and not the MD5. The result is in number of tested files by second.
- UsedSpace: This program computes the global size of each Storage (so in filesystems). The result is in number of file-sizes by second.
- MD5 KB/s: This curve shows the maximum efficiency in KB/s of the first program CheckFile MD5.

All these tests are running using filesystems on SATA disks, so these are slow storages.

- Obviously, we can see that checking MD5 takes more time than not checking MD5. With enough documents in the system (more than 450 000 documents), the efficiency is stable with around 36 000 documents by second in existence checking and around 16 600 documents by seconds (12 KB each) in existence and MD5 checking.
  For the test purpose, we only rely on one filesystem of 9 TB. Such performance can be exceeded when considering multiple physical filesystems, since parallelism can be achieved then.
  With the SATA disks, we were able to sustain 200 MB/s, so around 1600 Mbit/s, which is almost perfect considering the 2Gbit FC links we have on the SAN.

  So, in production status, we can say that if a check is running each night on the new documents:
  - In one hour the system can easily run the existence check on 130 000 000 documents,
  - and run the consistency check (same with MD5) on 60 000 000 documents on 12KB or on 700 GB used space, so almost the double time than without MD5 checking.

  Since the best import can be (from previous benchmark) at 1600 documents by second, saying at most during 22 hours in this high efficiency level, we get around 130 000 000 documents a day (on 22 hours) as a limit.

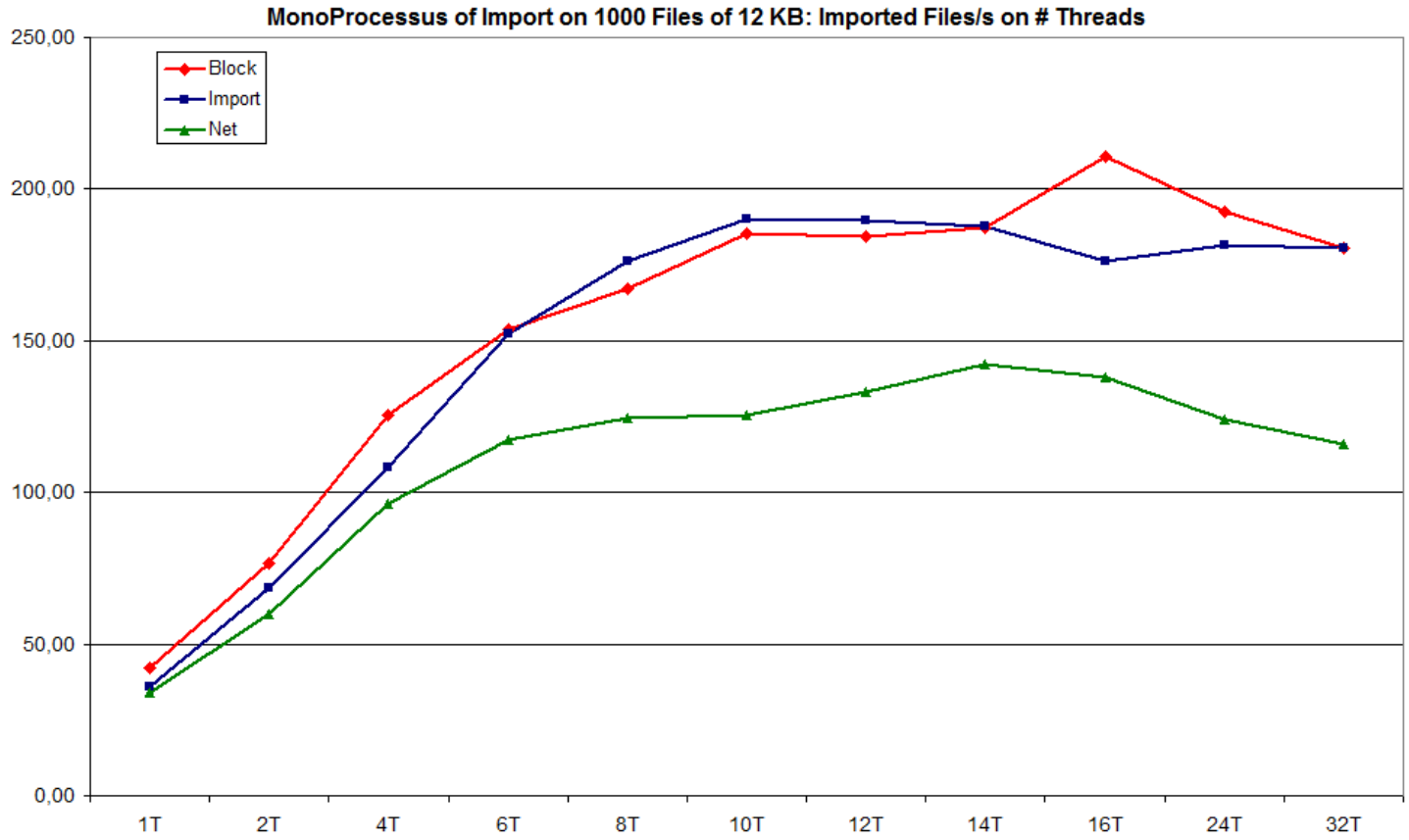
Therefore the system is reliable since if we can afford 2 hours a day of checking, we can have a full efficiency of import during 22 hours a day and still have time to run the consistency check.

- When we only check the real size used of files in each Storage, we sustain a performance close to 160 000 file by second, since the program mainly have read access on the filesystem directories structure and not to the real file.

Updated Benchmarks on version 1.0

Several optimisations were done on final Version 1.0 of OpenLSD.

1 000 documents import test case

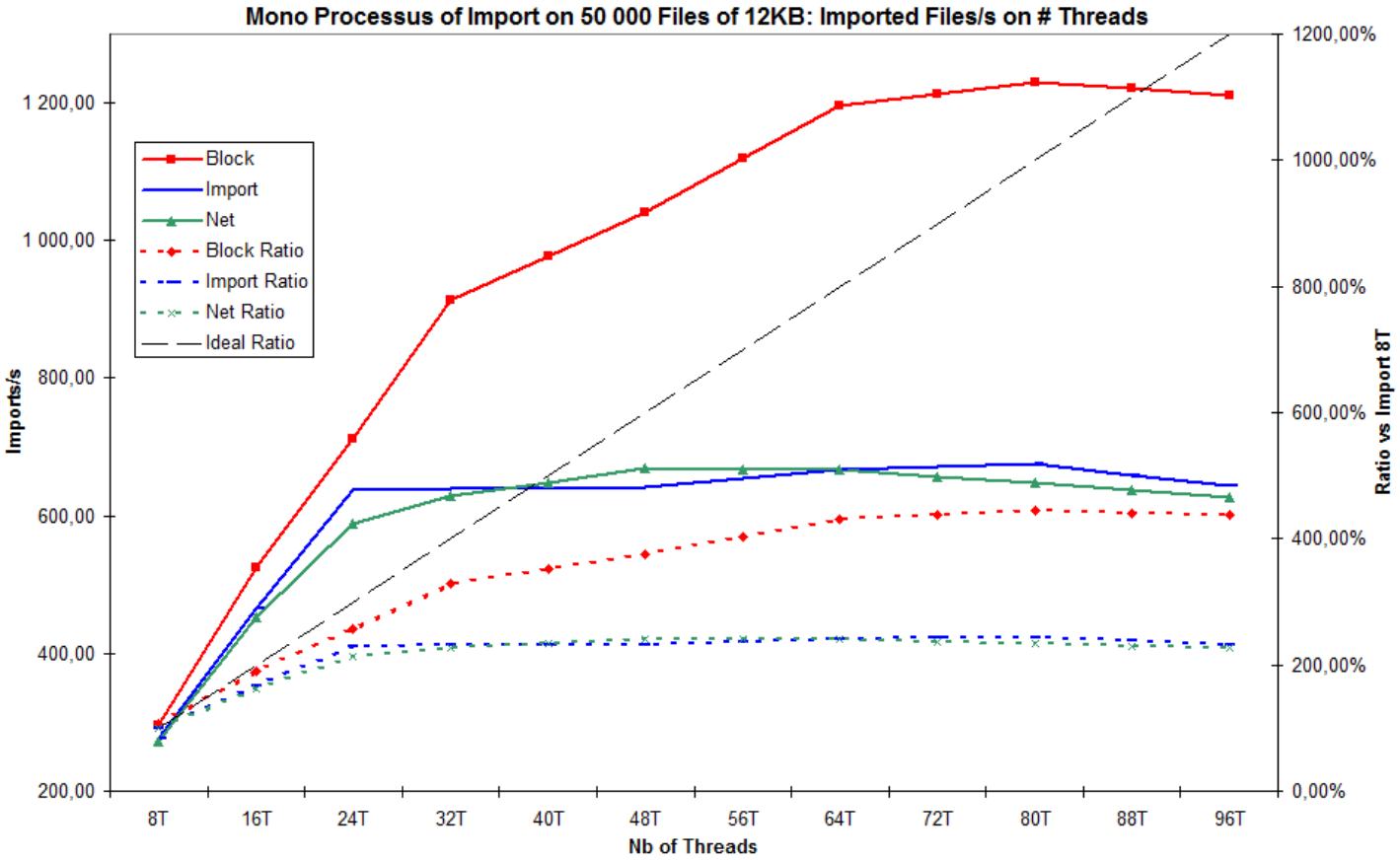


This benchmark improves the result up to 210 imported files by second with 16 threads against 180 in previous benchmarks. This improvement is about 15%.

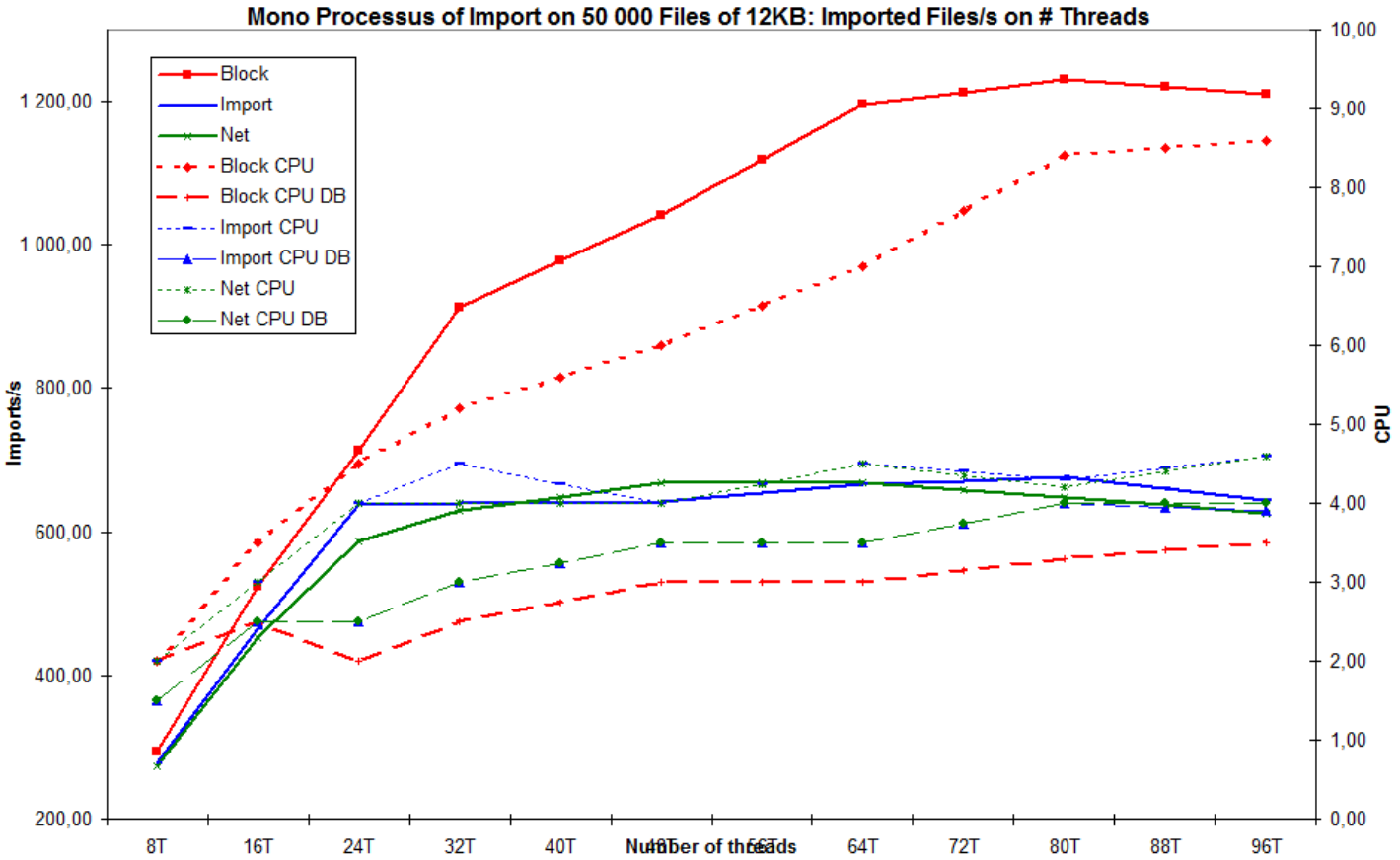
50 000 documents import test case

We see same kind of improvement in 50 000 test case.
In previous benchmarks, we saw up to 1150 import/s with 100 threads. In the new benchmarks on final version, we get up to 1230 import/s with 80 threads.
This improvement is about 7%.
We show on the first picture the ratio (compares to Import version on 8 Threads) with the theoretical ration (black doted line).

- In theory, the more the number of threads, the more the efficiency.
- In practice, we can see that after 32 threads, the efficiency is almost not going up anymore.
- For the Block version, there is still some improvement but from 64 threads, again, almost no more improvement is obtained.



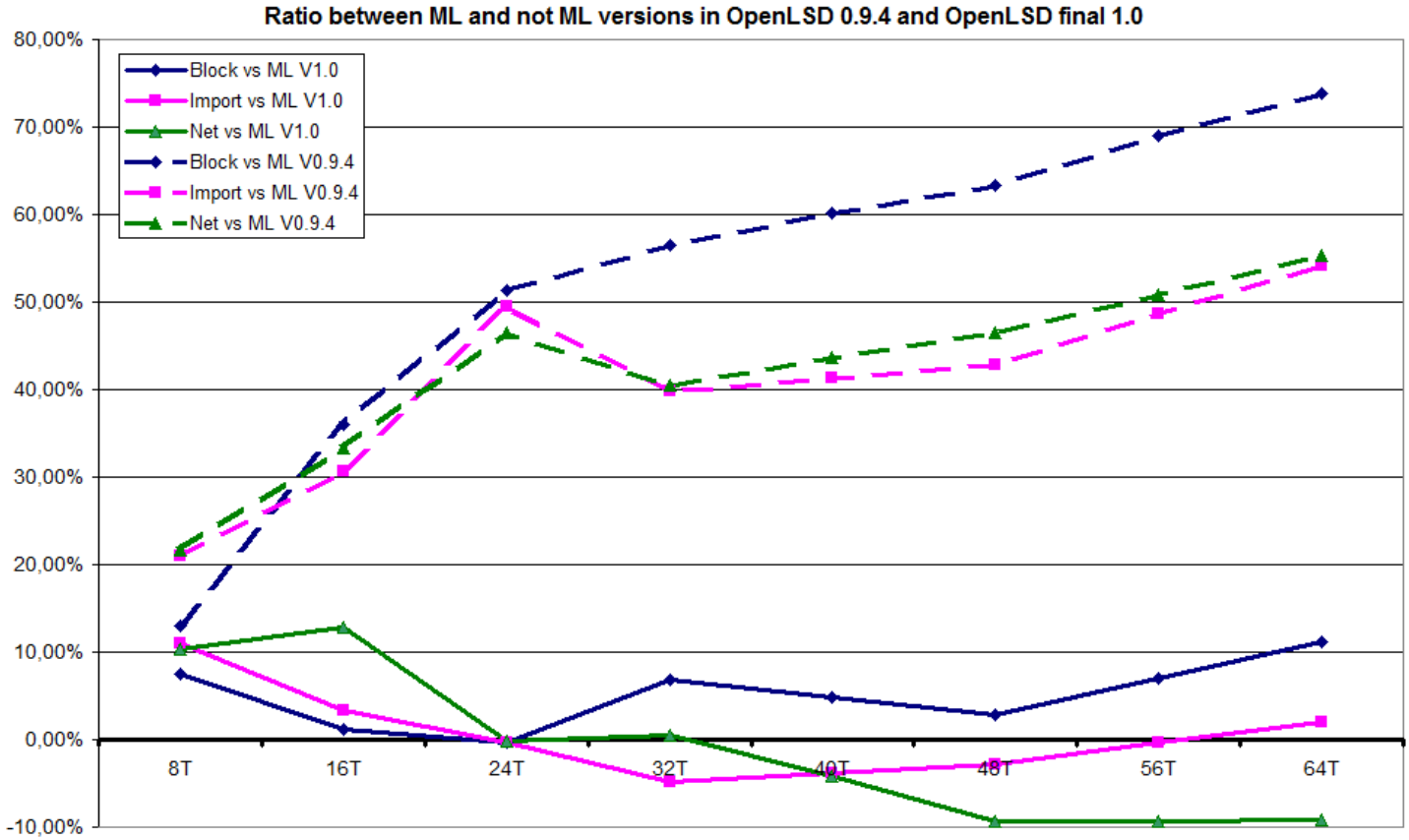
The next graph shows the CPU consumption based on the number of CPU. For the "CPU", the server has 10 Power5 CPU. For the "CPU DB", the server has 8 Power5 CPU.



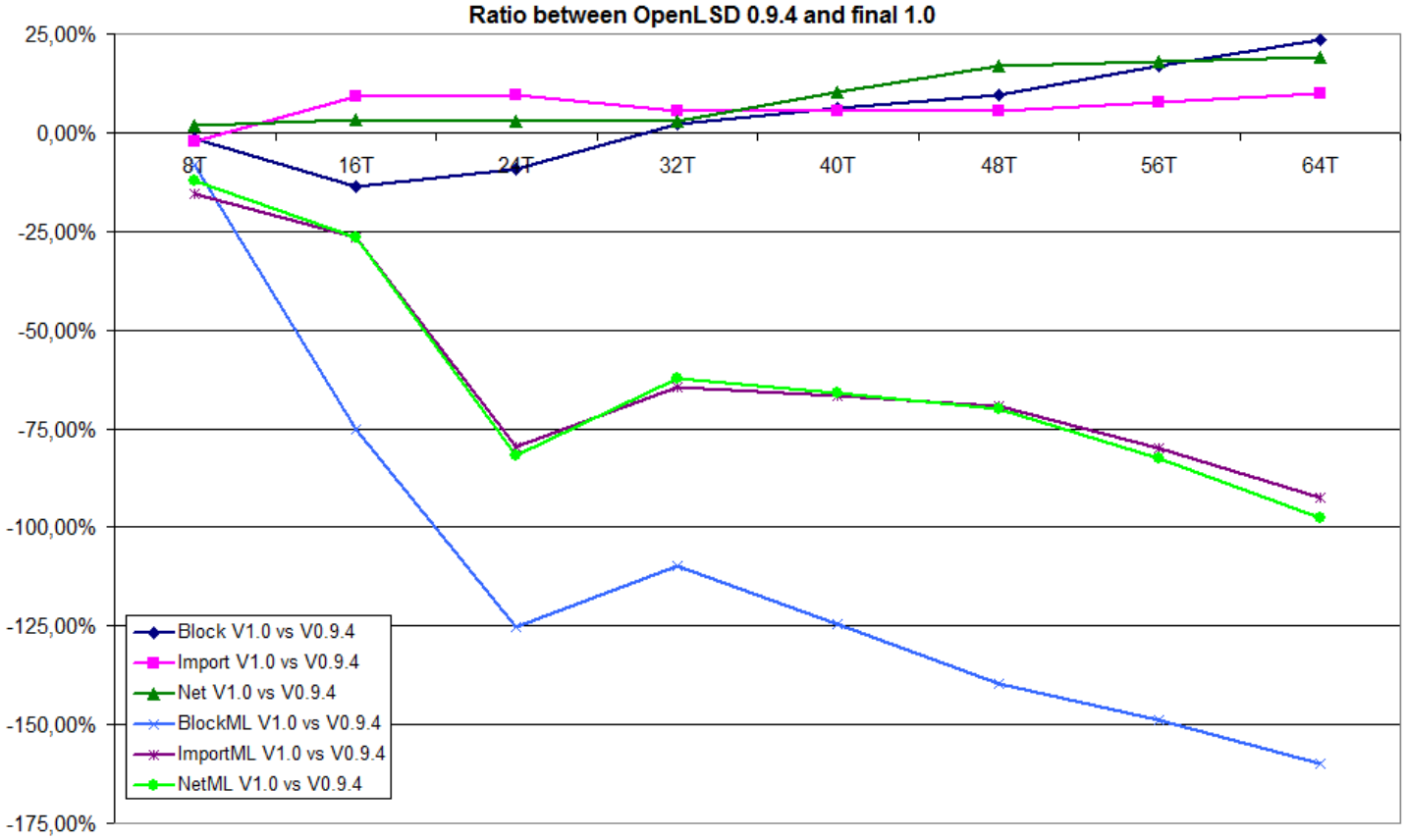
- We can see that the database server is almost stable with a slow increase when more threads are running.
- For the application server (OpenLSD Server), we can see the CPU is directly related to the efficiency of import.

50 000 documents import test case with multiple imports

One of the biggest optimisation concerns the ML support of import functions. The version 1.0 includes an improved algorithm and correctness. Below, two graphs show the improvements.



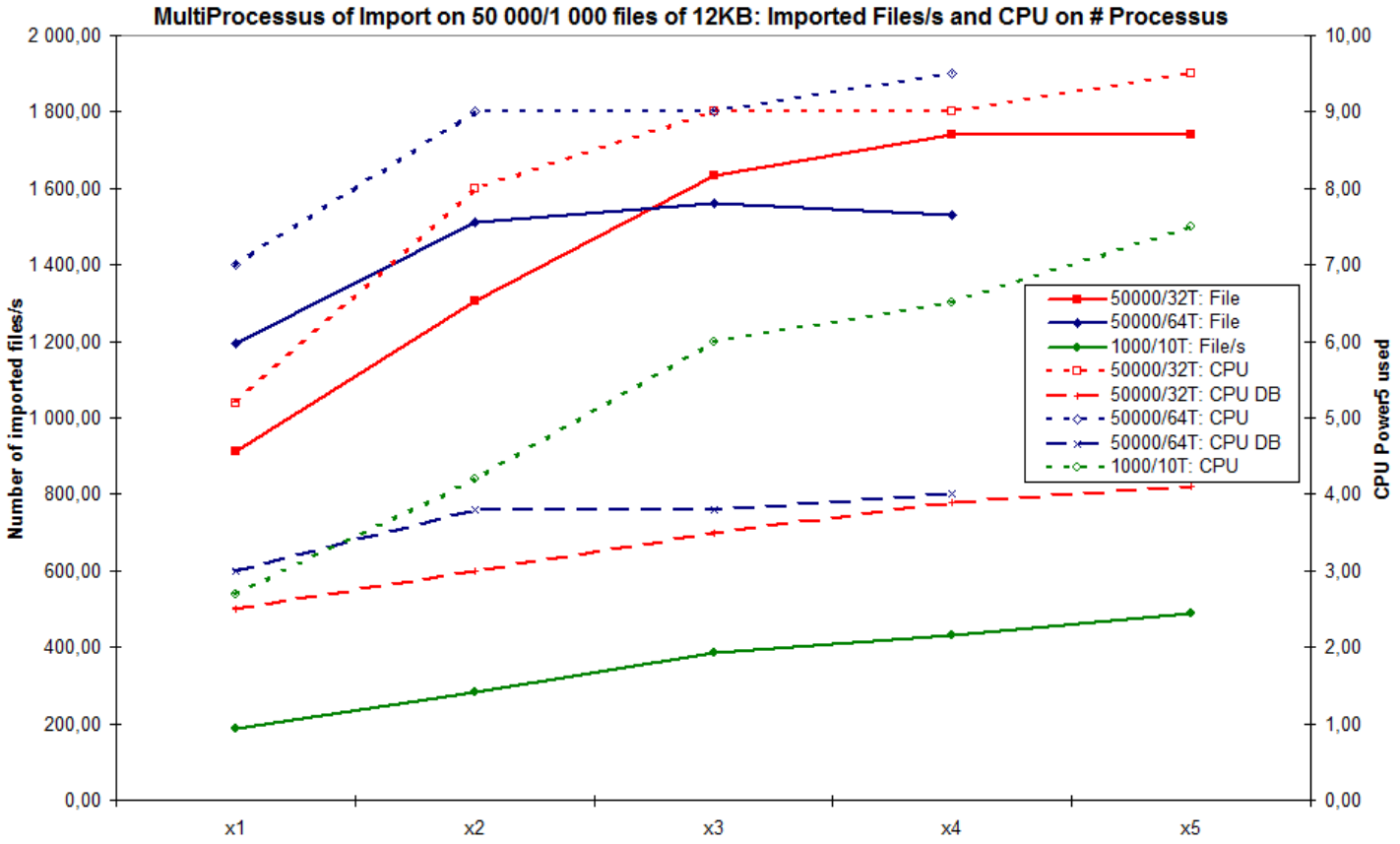
This graph shows that ML versions are now very close to the performances of not ML version (between -10 to +10%, instead of +50% and increasing).



- This graph shows that differences between V0.9.4 and V1.0 are very profitable to ML version (up to 160% of improvement). The previous contention was mainly in database locks which were optimized.
- We found some low performance (about 10 to 20%) with not ML versions in V1.0 compare to previous benchmarks results. The problem is that we don't find any clue on those degradations since no codes

were changed on those versions. We also get back to version 0.9.4 to check again and found the same degradations than in 1.0, so the OpenLSD code should be exonerated for this degradation. We suspect those numbers came from the servers itself (disk, network or database behaviours) since we got some problems on the underlying platform (not related with our benchmarks) but we don't find any clue. However, we prefer to show those numbers, even if we suspect something wrong outside OpenLSD itself.

The biggest news is that ML versions are now at the same efficiency than not ML versions.



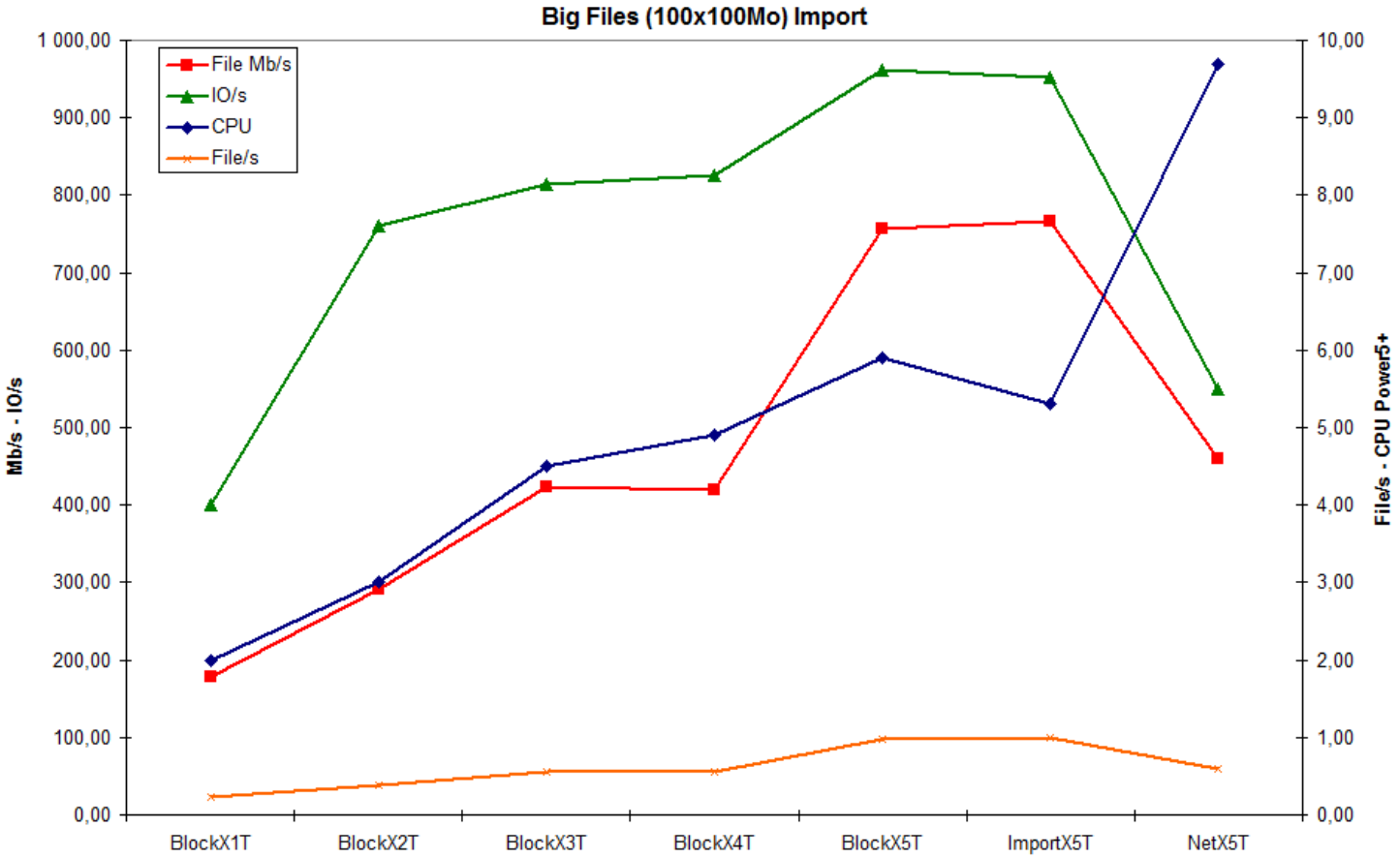
In the previous version, we get up to 1612 imported documents/s. In the final V1.0, we get up to 1740 imported documents/s. This is an improvement of 8%.

No improvement can be obtained from 4 processes for 50 000 test cases (32 and 64 threads) since the test reaches around 90% and more of CPU usage.

For 1000/10 threads test case, the CPU usage is not so far from the limit (around 75% with 5 processes) but with an acceleration (scalability) of 84% (5 concurrent processes goes almost at 84% of efficiency compares to 1 process alone).

100 documents of big files (100 M Bytes) import test case

We made some benchmarks on heavy big files (100 Mbytes) to see the impact on import function (and later on Web retrieve).



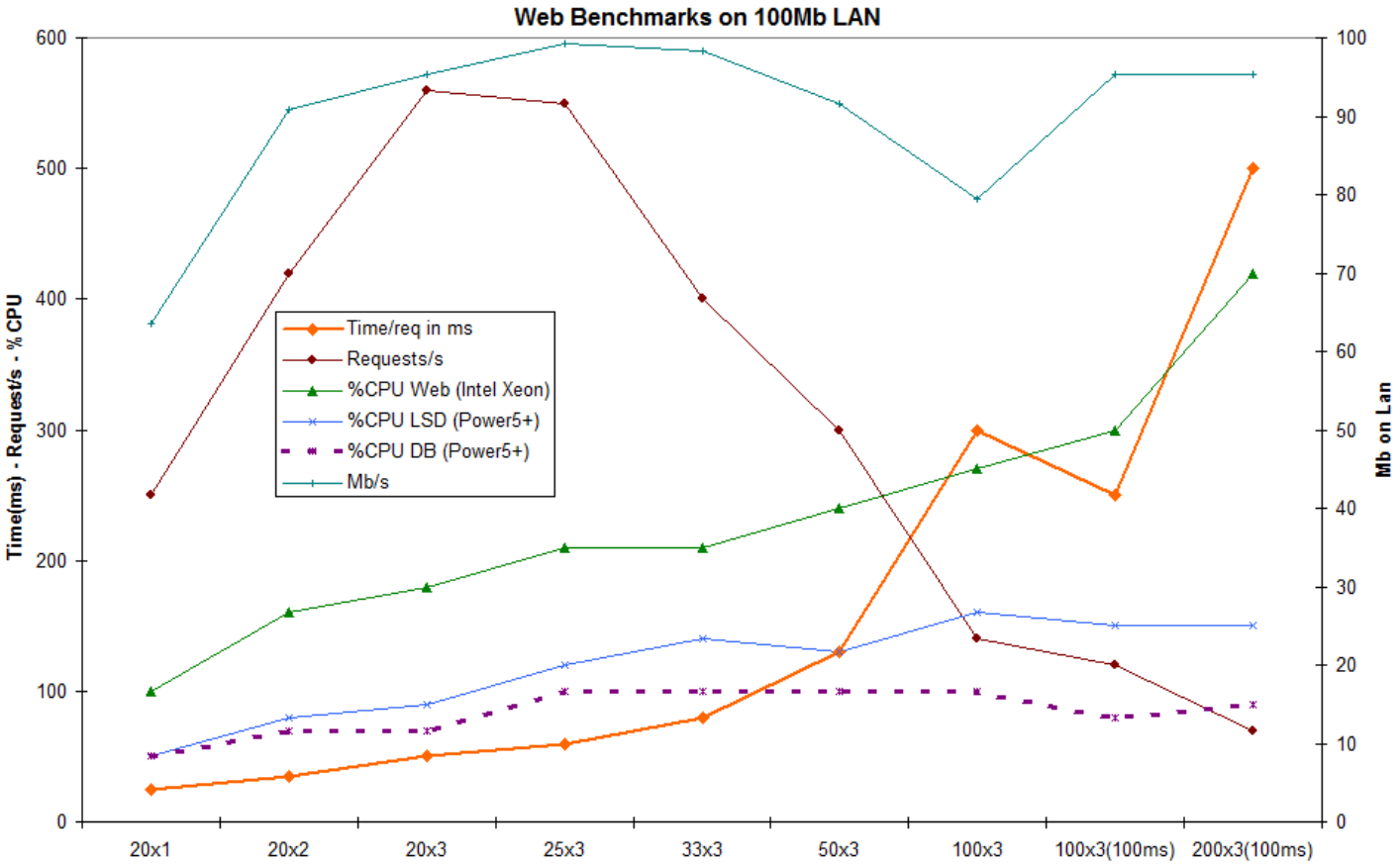
- On block versions, each block is composed by 20 documents (of 100 Mo each). So obviously, the performance increases according to the ratio of threads and the number of block (5 blocks maximum).
- We can see that we reach about almost 760 Mb/s on disk write (on 2Gbs fibre channel SAN) and about 950 IO/s. This for us a very good result since we can reach almost the half optimal write bandwidth.
- Using x threads in such scenario will use almost the full processor for the underlying IOs, so around x CPU Power5+.
- Performances are almost the same between Block and Import (no block) considering 5 threads.
- Performances on Network version (with 5 threads) are half of the local import (block or not). The LAN is a Gigabit network and we reach about 460 Megabit of bandwidth. However, the CPU gets higher consume with about almost 10 CPU, so almost the double of the CPU in block version (5 for the importer, 5 for the OpenLSD Server).

Based on these results, we found that OpenLSD is really optimized both on small files (12 KB) and big files (100 MB), and is only limited to the underlying platform (CPU, network, SAN).

Web retrieve of documents

We made some Web retrieve benchmarks using a 100 Megabit LAN from the client point of view, even if the network in OpenLSD platform is a Gigabit LAN.
We made several tests to stress the web retrieval. All tests use a 10ms of latency between each request, except where it says 100ms (the two last benchmarks). So we are in high stress benchmark mode.
We use from 1 to 3 Tomcat independent servers, each of them is a 2 Core Xeon 64 bits with a Suse SLES 10 SP1 64 bits system in an IBM Blade Center.
We use for commodity OpenSTA to benchmark the web retrievals on 2 Intel servers with Windows 2003 servers on a 100Mb LAN.

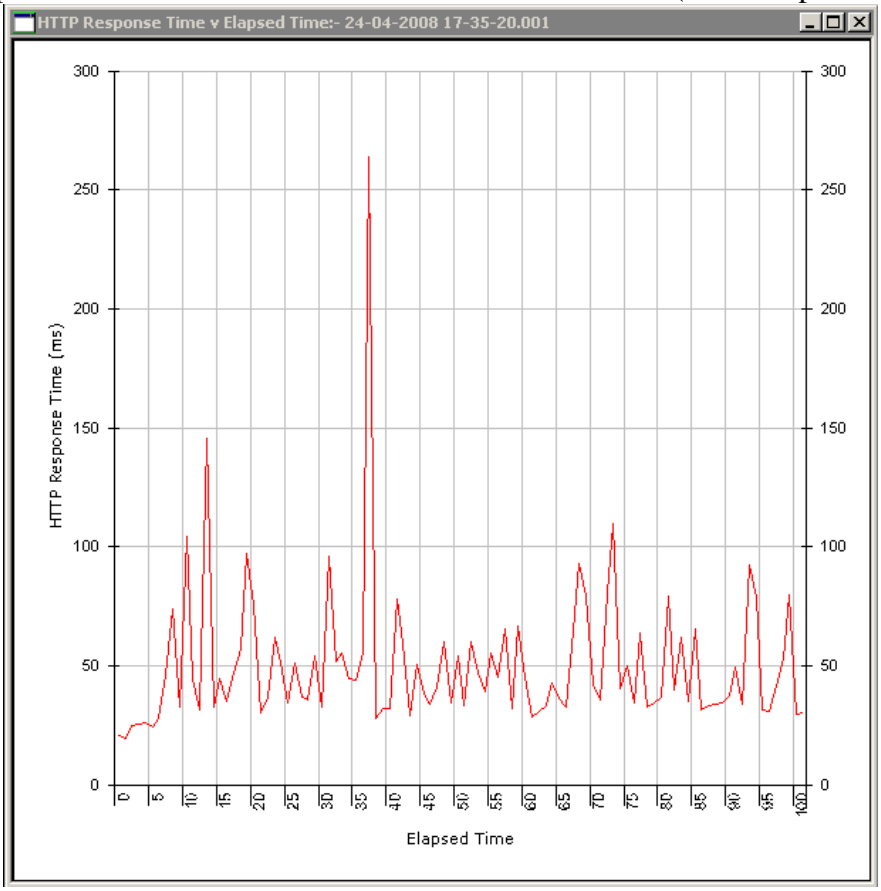
12KB documents test case



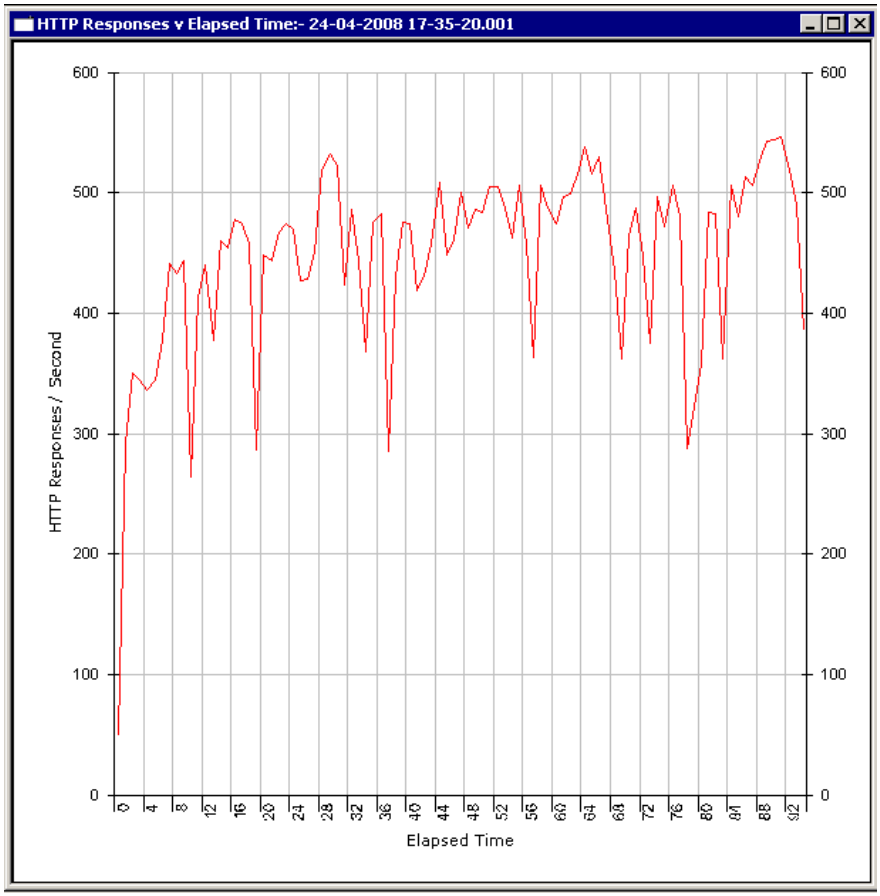
- The first 3 tests were used to qualify the maximum throughput. Then next results almost always reached the maximum throughput of the 100 Mb LAN.
    - From one Tomcat server, we reached up to 250 requests by second with 25 ms of average response time for 20 concurrent requests.
    - From two Tomcat servers, we reached up to 420 requests by second with 30 ms of average response time for 40 concurrent requests.
    - From three Tomcat servers, we reached up to 560 requests by second with 50 ms of average response time for 60 concurrent requests.
- When increasing the number of concurrent users, using always 3 Tomcat servers, we reached the maximum throughput for the LAN bandwidth, so the number of requests by second is decreasing and the average response time is increasing:
    - 75 concurrent users: up to 550 requests/s and 60 ms of average response time
    - 99 concurrent users: up to 400 requests/s and 80 ms of average response time
    - 150 concurrent users: up to 300 requests/s and 130 ms of average response time
    - 300 concurrent users: up to 140 requests/s and 300 ms of average response time
- When increasing the latency (10ms to 100ms), so decreasing the concurrency of simulated users:
    - 300 almost concurrent users: up to 120 requests/s and 250 ms of average response time
    - 600 almost concurrent users: up to 70 requests/s and 500 ms of average response time

- The more the concurrent users, the more CPU are consumed on Tomcat servers (almost linear). For others servers (OpenLSD Server and Database Server), the CPU are increasing but very slowly (from 50 to 160%).

We show two graphs of the benchmark done with 75 concurrent users (scenario previously named 25x3).

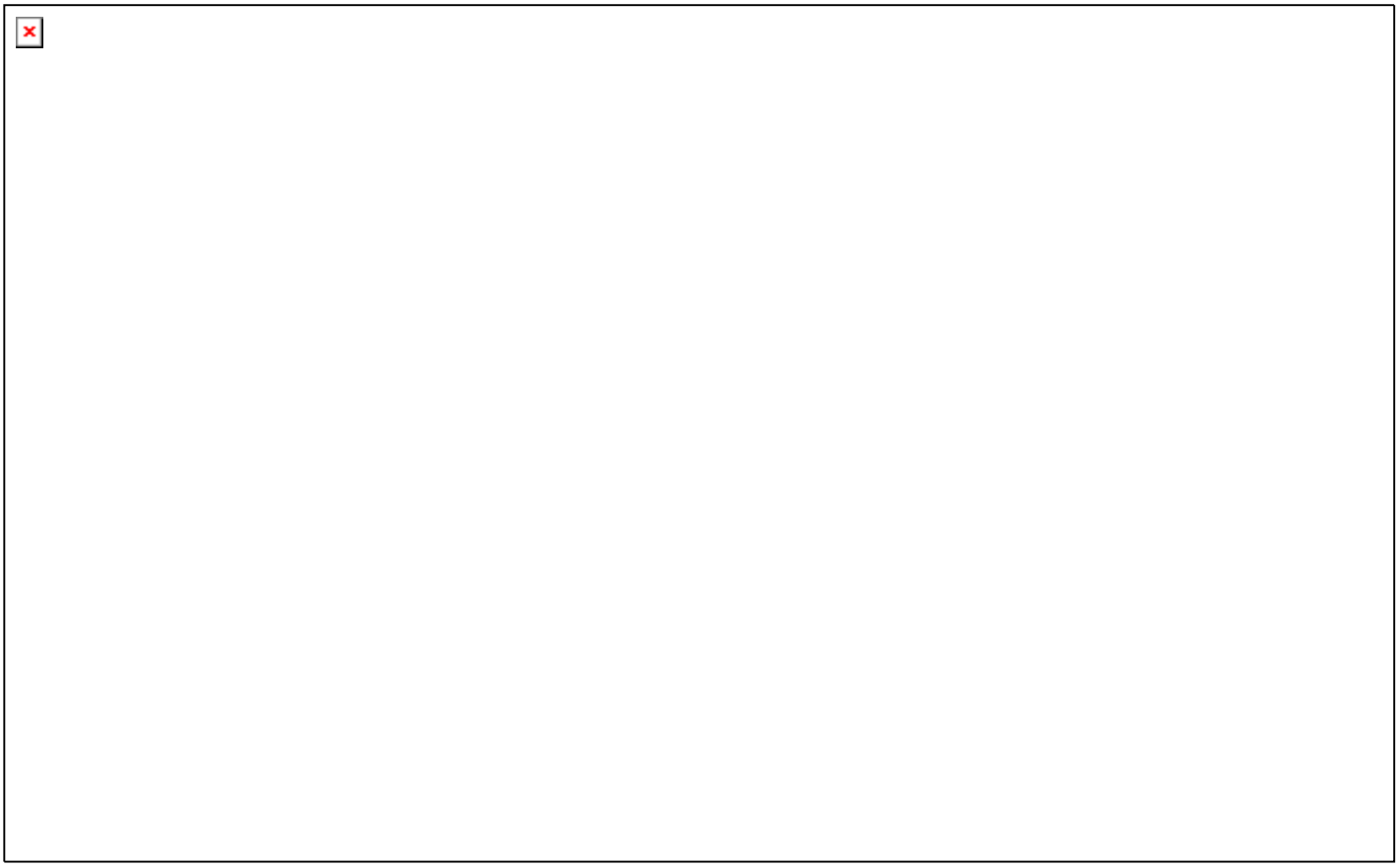


This graph shows a relative stability of requests by second once the users are all actives, mainly between 400 and 550 requests by second.



This graph shows a relative stability of the HTTP response times once the users are all actives, mainly between 30 to 100 ms (with an average of 60 ms).

100MB documents test case



We made the same benchmark but on big file (100 MegaBytes) in the same condition, but limited by the number of users to 1 or 2. The objective of this benchmark is to see if the OpenLSD behaviour is OK when one big file can completely fill the bandwidth of the LAN.

- First of all, the bandwidth is almost fully used (> 96%).
- The time is proportional with the bandwidth and the size of the underlying file (around 8 seconds for one file, around 16 seconds for two files).
- The CPU goes double (two Tomcat servers were used) when two files are served simultaneously.

This benchmark gives the result that the underlying implementation is efficient both on small files (12 KB) and on big files (100 MB).

Memory consumption

However, during our benchmarks on big files, when we try to implement more concurrent users on the same Tomcat server, it brings Out Of Memory Exception due to the option we used on the network protocol. Indeed, we generally used the “No Ack” protocol version which enables more throughputs of the files in web retrieve. The drawback of this method is bigger memory consumption if the final user has a lower bandwidth than between the Tomcat server (OpenLSD Client) and the application server (OpenLSD Server). In our benchmark protocol, this is the case (1 Gb LAN in OpenLSD area and 100 Mb in client area), so the OpenLSD Client (Tomcat server) will store packets that are not yet sent to the final client, given a big memory consumption with big files. For instance, 50 concurrent users will probably tend to have almost 50*100 MB in one Tomcat server memory, so 5 GB of memory consumption, where we limit our Tomcat to 2GB for the benchmark.
We decide to change the behaviour of the JSP page were if the document size is lower than 1MB, the usual “No Ack” protocol will be used, and if bigger the “Acked” protocol will be used, that is to say that every packet must be validated by the Tomcat server before it gets the next block. Obviously this protocol should be not as fast as the usual one, but at least it tries to prevent the OOM.
The memory consumption in “Acked” mode is proportional to the concurrent allowed users. If 300 concurrent users are allowed on one Tomcat server, then the memory consumption is about 300 x 16KB x 3 (one packet received, one packet currently sent to the ServletReponse, and another one for the next packet received), so around 14,5 MB for the data. Of course the underlying objects need also memory (OpenLSD objects, the underlying MINA network layout …), so considering a Tomcat with 2GB of memory should be enough.