All-test setup, unless onterwise specified:

- 3 replicas on `hpc{2..4}`
- 3×500 clients on `hpc{5..7}`
- default properties (tcp, window size=2)
- no-op logging
- 1kB request size

Code base: commit 5617bc047f0660d05f14b87b1e42f163d5b248ae

1 CPU or network saturation

Target: see what is the bottleneck in JPaxos - network or CPU?

Measured: cpu and network usage on leader, follower and client machine.

1.1 1024B requests

1.1.1 CPU

Iddle CPU on nodes [percentage]:

/	Client	Follower	Leader
0	89,60	69,57	58,67
1	89,16	80,12	57,95
2	86,34	89,32	67,76
3	64,52	60,87	37,45

Leader and follower average CPU detailed usage:

	user	sys	iowait	irq	soft	idle
leader:	23,92	10,13	0,07	1,77	8,66	55,46
follower:	14,51	04,71	0,12	1,06	4,62	74,97

Leader per-core CPU detailed usage:

CPU	user	sys	iowait	irq	soft	idle
0	31,25	8,27	0,00	0,00	1,81	58,67
1	30,08	9,55	0,00	0,00	2,42	57,95
2	18,26	11,22	0,00	0,00	2,76	67,76
3	16,09	11,47	0,28	7,06	27,66	37,45

Legend:

user	executing at the user level (application)
sys	executing at the system level (kernel) without any interrupts
irq	hardware interrupts
soft	software interrupts

Conclusions CPU is not saturated. Thread CPU usage is unbalanced. Thread performing network tasks uses processor at most. If the throughput would increase by 60%, the CPU would be a possible bottleneck.

New test needed: how small must the request size be to saturate CPU.

1.1.2 Network

Used bandwidth on nodes [10^6bits/s aka Mbit/s]:

/	Client	Follower	Leader
in	130	484	394
out	128	215	908

Measured TCP throughput on the network is 941.07 [10^6bits/s] (3× `netperf -H hpc3 -cC -fm -j -tTCP_STREAM -- -m 1024`). Bandwidth used during the test TCP throughput test is 987 [10^6bits/s].

Conclusions Network usage reaches 92% of its capabilities. This means that the network is close to saturation, yet not fully saturated. Network is a bottleneck in this test.

1.2 Multiple request sizes

Following test shows how JPaxos behaves with 16B, 128B, 1024B and 8196B requests.

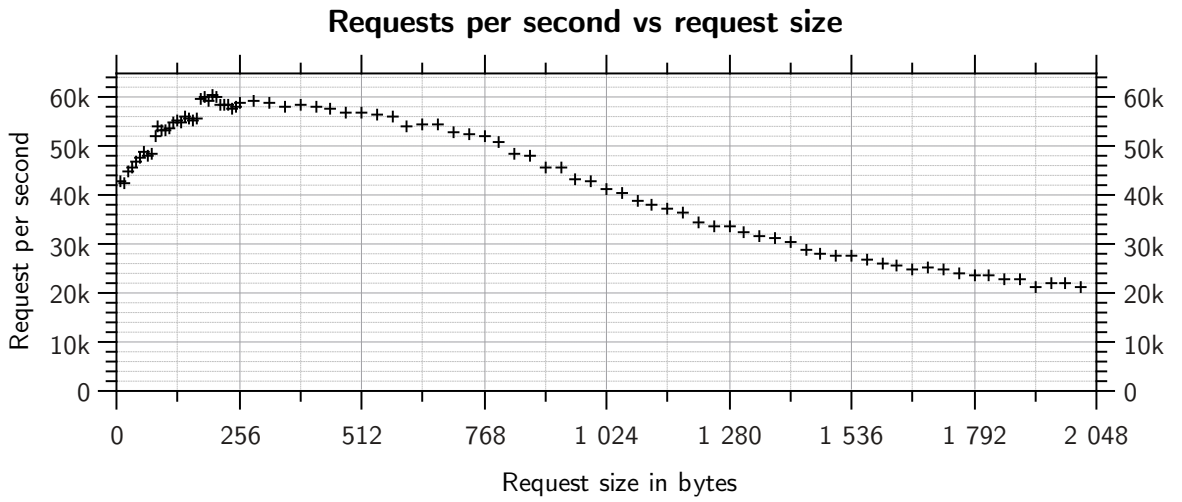
The values on the plot are normalized, i.e., 8KB request count per second has been multiplied by 8, and the 128B request count per second has been divided by 8. This makes the results comparable.

With 8KB requests the network gets saturated very soon (with 150 clients). With 1KB requests the network gets saturated starting from 1200 clients. Smaller requests cannot saturate the network, and after reaching peek throughput at about 2500 clients, the performance drops.

To see when the border request byte size where JPaxos stops saturating CPU and starts saturating network, the following test has been done:

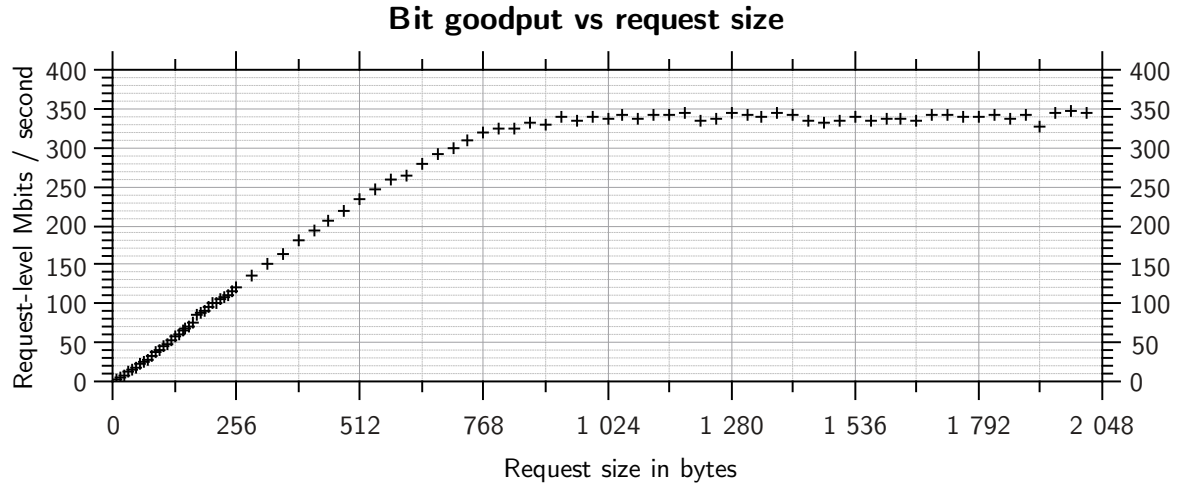
- 3×700 clients
- request size varying from 16B to 2048B

The results, unnormalized, are as follows:



So to get highest request per second value (in order to show 'we have the highest r/s'), one must take 200B requests. This however says nothing about real performance.

Following plot shows state-machine level goodput (i.e. how much data the state machine got in a time unit):



The plot shows that when the request size exceeds about 900B, the network gets saturated. Up to 768 bytes the bit throughput grows nearly linearly with increasing request size – larger request takes the same CPU time, but transports more data. Starting from about 900B the number of requests per second decreases, but the amount of data remains on a stable level, indicating ghat the network bandwidth limits throughput.

The test included 2100 clients total, which is enough to produce an amount of request needed to saturate the network from clients to the replicas starting with request size of 180B (in theory), so increasing the number of client won't change the 900B limit.

1.3 200B requests

After analysing various request sizes, 200B requests were taken under account:

1.3.1 Network

Used bandwidth on nodes [10^6bits/s aka Mbit/s]:

/	Client	Follower	Leader
in	57	172	136
out	52	99	297

There is a lot of bandwidth free. Less than 1/3rd of bandwidth is used on any machine in any single direction. Network is not the bottleneck.

1.3.2 CPU

Iddle CPU on nodes [percentage]:

/	Client	Follower	Leader
0	81,66	50,03	43,72
1	82,75	54,90	36,86
2	79,96	81,70	62,68
3	55,98	68,00	46,98

All machines have higher CPU usage, but there is still a lot of free time on any node. (?)

Leader and follower average CPU detailed usage:

	user	sys	iowait	irq	soft	idle
leader:	33,09	12,12	0,09	0,83	6,16	47,4
follower:	24,86	7,63	0,17	0,51	3,17	63,66

Leader per-core CPU detailed usage:

CPU	user	sys	iowait	irq	soft	idle
0	39,13	15,63	0,00	0,00	1,50	43,74
1	51,78	10,23	0,00	0,00	1,07	36,91
2	24,30	11,43	0,00	0,04	1,43	62,80
3	17,14	11,18	0,35	3,36	20,95	47,03

Leader has a lot to do, but there is free processor time left.

If not CPU and network, it seems that synchronisation issues slow down the JPaxos now. (I hope to have time to discuss it with Tadek and Maciej).

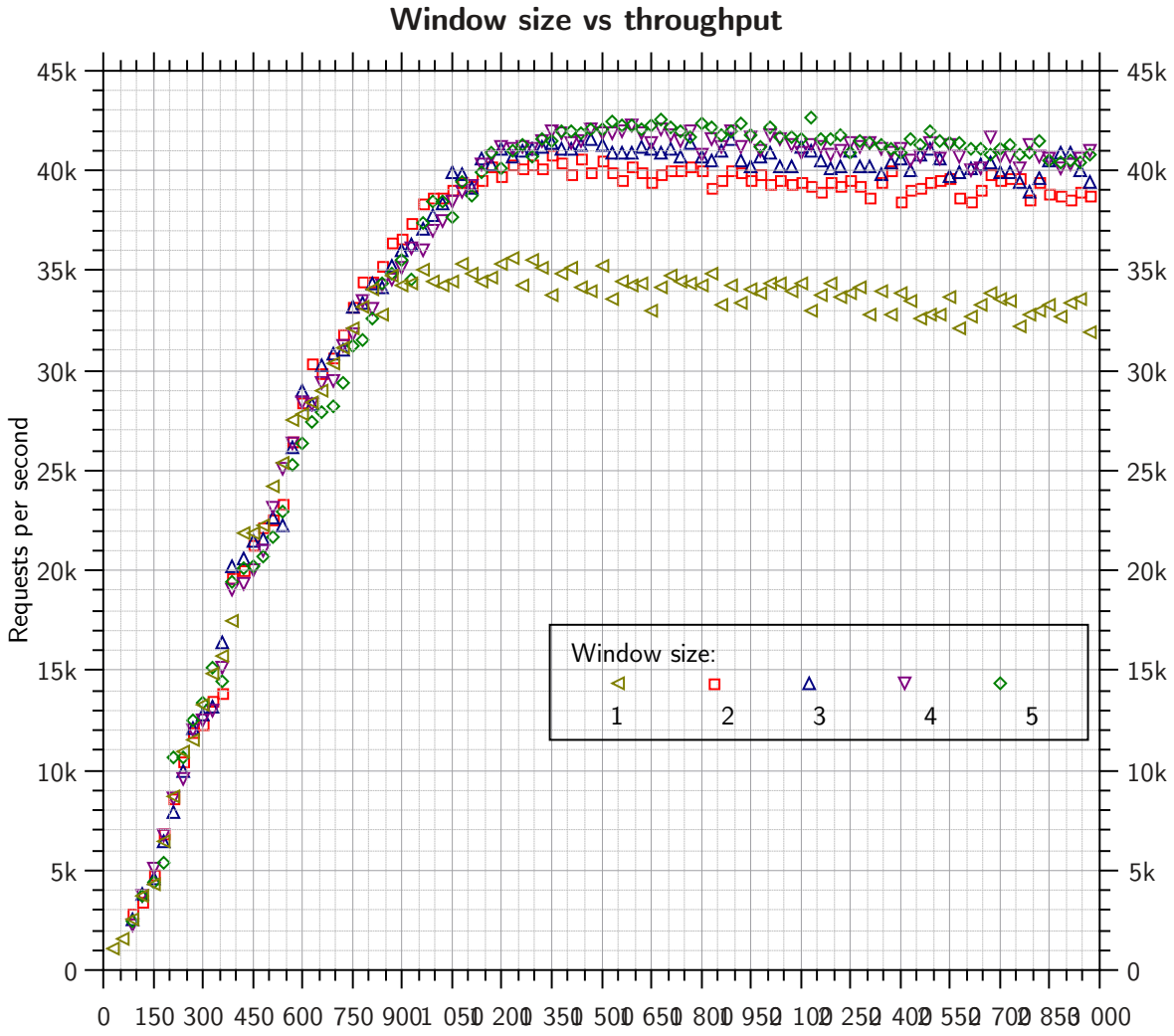
2 Choosing window size

Nuno in [Tuning Paxos for high-throughput with batching and pipelining] stated that:

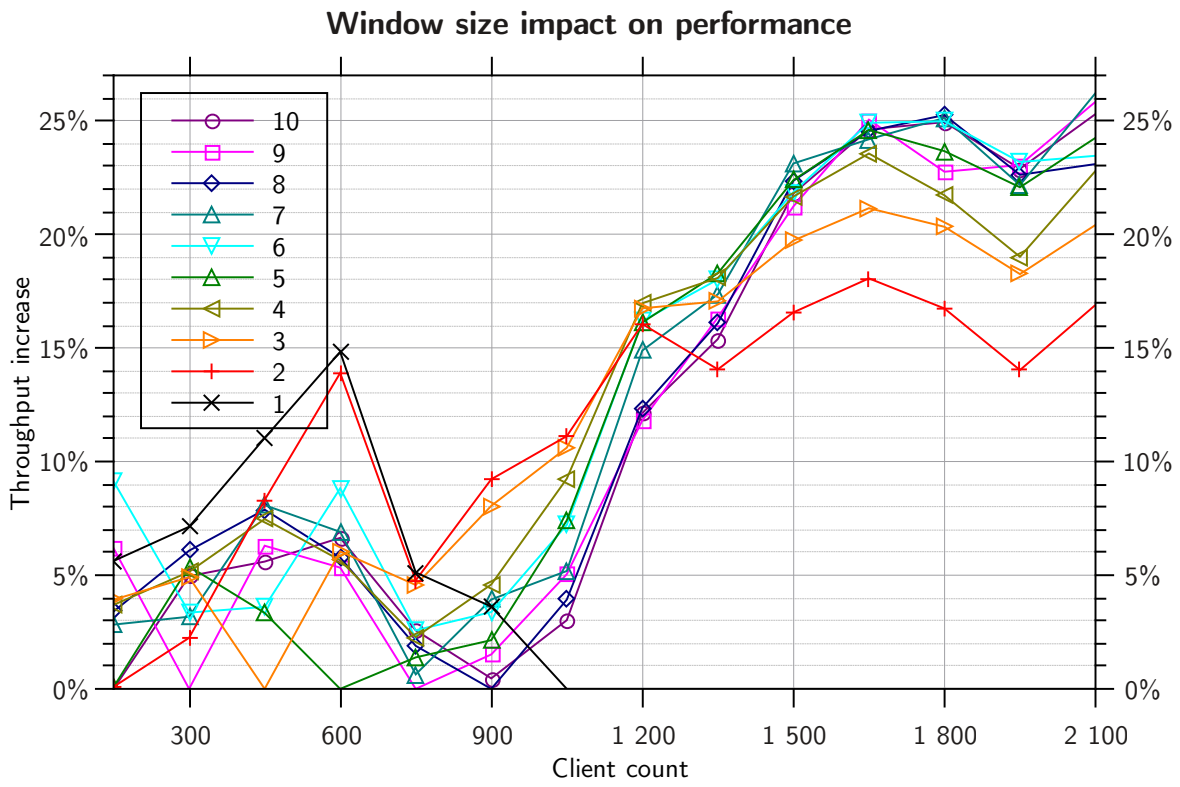
> A note on the cluster results: In the experiments performed in a cluster environment, batching by itself is enough to achieve the maximum throughput, with pipelining having minimal impact on the results. The reason for this difference is that the latency in a cluster is very low so the leader does not have time to start new instances while waiting for the results of previous instances.

This is however false. (At least on HPC cluster.)

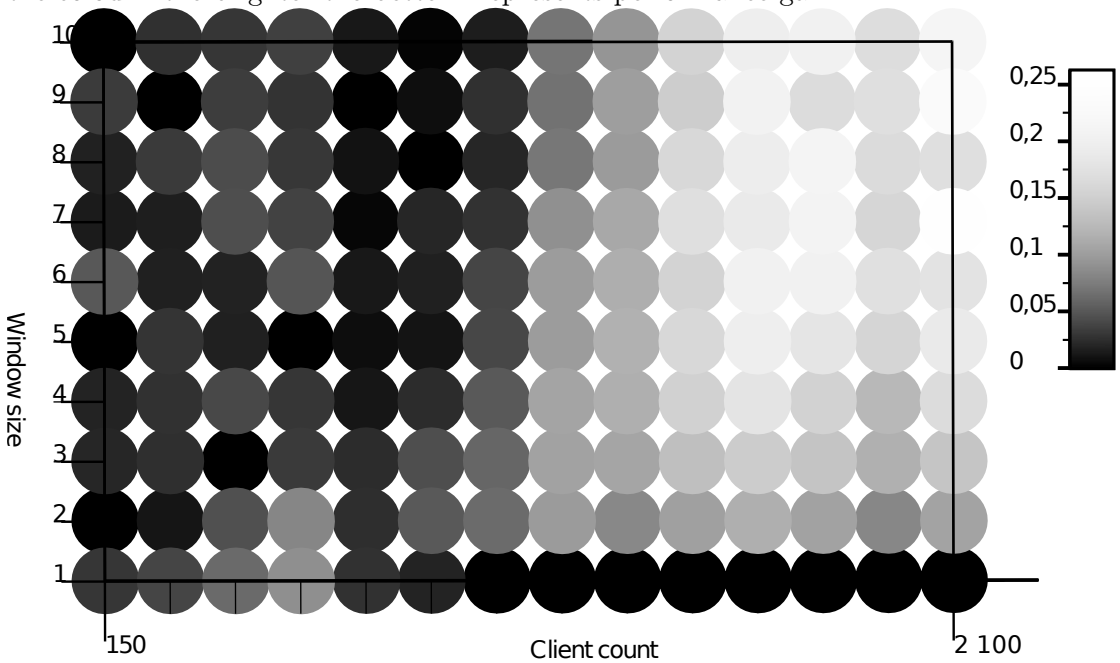
During the tests first window size $1 \div 5$ has been tested, showing that the higher window size, the better the results are:



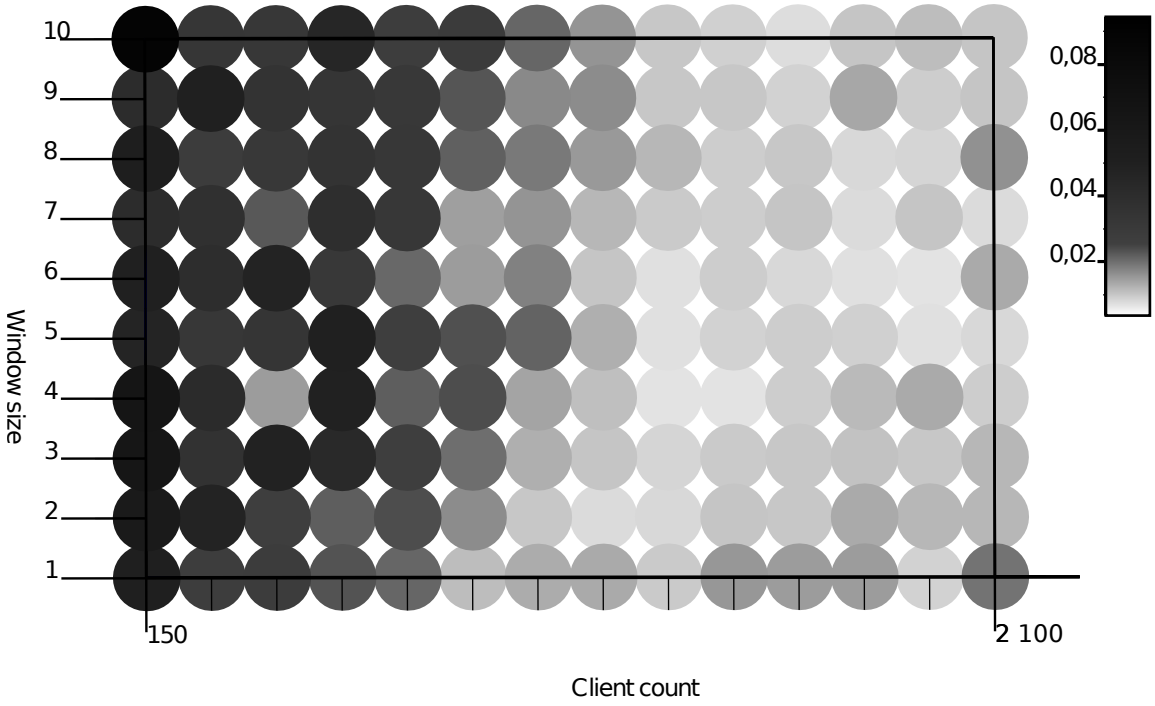
The tests have been continued up to window size 10, as the performance increase stabilized at window size 6 and higher. To present the results in more readable way, the points represent the gain (in %) from the worst result at each client count:



To make the results even more readable (at least for me) the performance gain has been also presented on a 3d plot – x and y axes represent client count and window size accordingly, while the colour - the brighter the better - represents performance gain:



What is not presented on the plots is the variance of the results. As for each client count and window size JPaxos has been run 10 times, except from bare performance also the variance of the performance can be calculated. The results are presented below – *notice:* the color scale is not linear this time



As long as the network is not saturated, the throughput is not stable; after the network becomes the bottleneck, the results are rather stable.

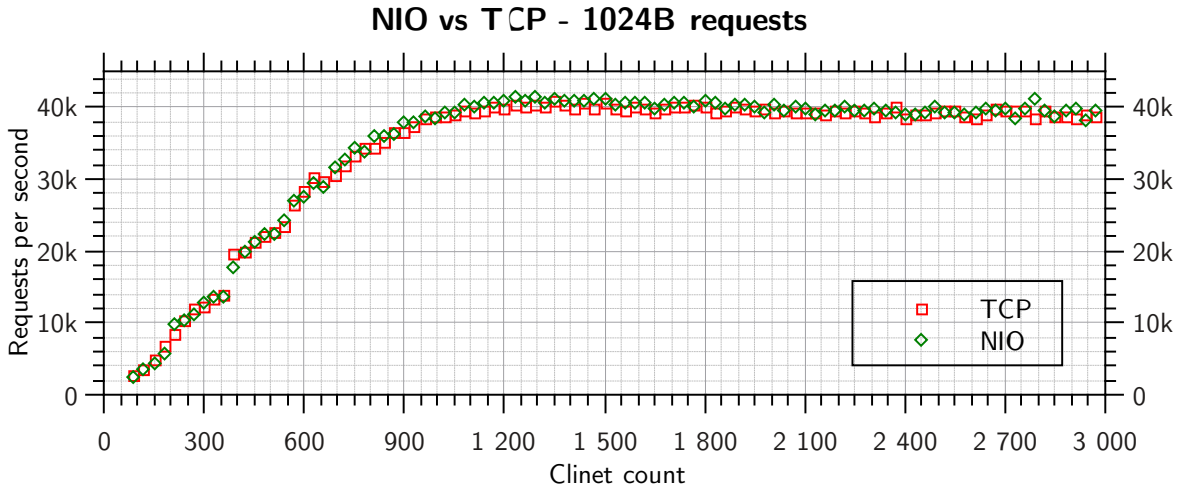
Conclusions The window size (and whole pipelining optimisation) is also important on cluster environments, increasing the speed in critical cases by 25% compared to no pipelining. As long as the network is not saturated, the result stdev is about 5%, and the gains are in order of 5% as well. On one hand, this makes choosing right window size hard, on the other – the performance loss on wrong choice is not big. After saturating the network pipelining increases the performance a lot. Also the stdev drops to 1%, making the results easier repeatable.

The window size of 6 and higher seems to be the best choice. It is advised to keep the window size as low as possible (due to its negative impact on the speed of recovering from any failures, i.e., view change and recovery). So proper window size for HPC cluster at tests with 1kB requests is six.

3 NIO network implementation impact on performance

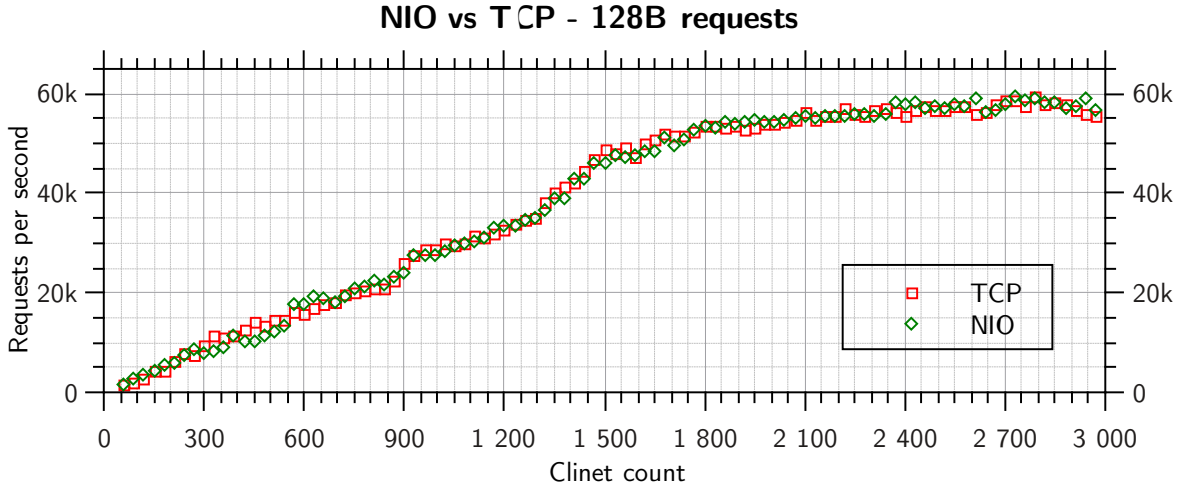
Maciej and Tadek extended JPaxos by new implementation of Network module – NIO. They expected NIO to be faster than 'old' java IO (which we call TCP, despite both Network modules using TCP). These tests show if it was worth writing.

3.1 1024B requess – network saturation



NIO throughput is 'a little better' – on average higher by 0,7%. The trend is identical.

3.2 128B requess – CPU saturation



Here, the CPU is more saturated. NIO was meant to help in such case. The experiments however do not agree with this claim – NIO and 'old IO' perform the same (NIO is on average 0,3% slower).

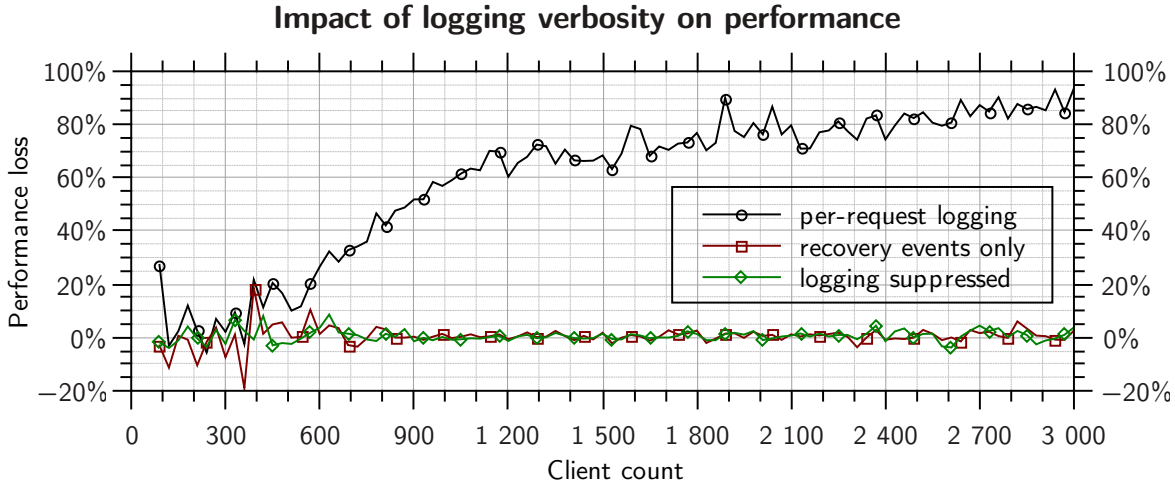
Conclusion it makes no real difference for JPaxos whether it uses NIO or 'old IO'.

4 Logging impact on performance

Using standard logging facility is flexible and cost-efficient method for analysing the behaviour of systems like JPaxos. However, logging takes some CPU time, and therefore may impact results. Previous JPaxos version used `java.util.logging` (JUL) for logging, which – as most built-in Java mechanisms – is rather performance-unfriendly. Now JPaxos uses `slf4j` with `logback` backend, and two classes of messages are tagged with marks: logging all performance-related events and logging recovery events only. Slf4j (contrary to java.util.logging) can be turned off

at all (i.e. there is no effective logging code). Logback can as well as JUL suppress all messages, makes it however a lot faster.

Tests how much recording events affects performance are presented here:



Test for each parameter values have been performed once only, however total of 100 tests for different client count are enough to take general conclusions. First, no-op logging is faster than suppressing all logging by 6‰, faster than selecting recovery-related events by 5‰, and faster than logging all performance-related events by 60%. This means that using logging for recording recovery-related events has little impact on performance and can be used in benchmarks, while full logging on request basis cannot. Also, full logging cost increases with client count (what is quite obvious, more clients = more requests to trace). The results for tests up to 900 clients have big variations (what is normal for JPaxos in such case), but the trends remain unaffected.