Workshop – practical project.

Testin tool.

Aliya Ibragimova, Teodor Macicas

Table of Contents

1. Introduction

The context of this project is the workshop Masters course held by University of Neuchatel. The task we had to accomplish was to create a flexible testing tool for benchmarking different I/O Java frameworks such as NIO.2, XNIO3 and Netty. In order to test them, simple Web client server applications were used. Being based on a previous project, the servers for NIO.2 and XNIO3 as well as the generic HTTP client have been reused. However, support for Netty asynchronous server was added.

The testing tool is meant to ease and automatize the run of the experiments as well as gathering the results. There are three main performance metrics that are intended to be compared: response time, CPU usage and memory usage on the server side. By offering flexibility in terms of environmental parameters and testing scenarios, our tool has been proved to be a solution for avoiding the burden of using manual scripts.

Next chapters explains a bit the differences of the three benchmarked I/O frameworks, then an architecture view over the developed testing tool, ending with hints how to use our software and a glimpse over the final results.

2. Input/Output frameworks

I/O frameworks are frameworks that facilitate low-level I/O processing programming by providing high-level abstractions for low-level operations. Modern I/O frameworks aim to improve performance of I/O processing by providing features for intensive I/O operations.
This section briefly describes three frameworks: NIO.2, XNIO.3 and Netty. NIO.2, framework released with Java Platform Standard Edition 7 (Java SE 7). XNIO.3 and NIO.2 are frameworks developed by JBoss. All frameworks were developed to replace old NIO framework and support new powerful asynchronous mechanism of I/Os processing for better performance.
NIO framework is a first Java I/O framework providing non-blocking capabilities for I/O processing that allows developers to write scalable solutions. However it has some limitations, the most important among them for network I/O operations are:

- The complex API. The programming constructs of Socket and Channel are implemented in different packages, so it is difficult to put them together.
- Lack of API for asynchronous (as opposed to non-blocking) I/O operations
- Framework doesn't allow to use platform- or OS-specific features.

2.1. NIO.2 framework

NIO.2 framework is one of the main new functional areas of Java SE 7. NIO.2 has following advancements:

- The API for asynchronous I/O operations. To handle asynchronous I/O operations NIO.2 supports two approaches: Future and Callback paradigms.
- The unification of socket-channel functionalities, with the support of binding, option configuration, multicast diagrams.
- Platform-independent file and asynchronous operations.

2.2. XNIO3 framework

XNIO.3 framework is a Java open source framework developed by JBoss. In comparison to NIO XNIO.3 has following advantages:

- API for combining blocking and non-blocking operations, that is easy to use.
- Callback based interface to handle asynchronous I/O operations.
- Support for the multicast sockets and non-socket I/O
- XNIO.3 framework can be used anywhere as a replacement of NIO framework in contrast to NIO.2 framework which is compatible only with Java SE 7.

2.3. Netty framework

Netty framework is “an asynchronous event-driven network application framework for rapid development of maintainable high performance protocol servers and clients”. Netty has following features:

- Unified API for various transport types – blocking and non-blocking socket.
- Highly customizable thread model -single thread, one or more thread pools such as SEDA (staged event-driven architecture).
- Asynchronous API for I/O processing following Future and Callback paradigms.
- Connectionless datagram socket support.
- Flexible and easy to use API. Netty framework comes with well-documented Javadocs and a lot of provided examples.
- Compatible with Java SE 6, however some features are only usable with Java SE 7.

3. Testing tool architecture

The testing tool is a Java SE7 application using Maven for dependency resolution and project building. It uses a properties file where environmental parameters are passed to the program (e.g. IP addresses of server and clients). Another bunch of properties files are used for defining testing scenarios as explained in the section 4.2.

It uses an SSH library (https://github.com/shikhar/sshj) for being able to connect and send commands to remote machines. After copying the Java programs to the clients and server, it starts the test and waits for the results to download. A more detailed execution flow is available in the following subsections.

3.1. Project dependencies

Here are highlighted the important dependencies that are presented in the pom.xml file for automatic installation:

- library for easily running ssh commands: sshj
- library for parsing properties file: Apache commons configuration

3.2. Helper threads

There is a set of parameters used by helper threads and also by the main one. All of them are declared into the *Utils* class. Please check that in case you need to change (most probably the timeouts or file names).

Besides the main execution thread whose flow is explained in the section 3.4, there are some additional concurrent threads as follows:
a) *MachineConnectivityThread* :: it periodically checks the server's and clients' SSH connection by simply issuing a dummy command remotely; there is an associated flag that is changed accordingly to the operation status

b) *WriteStatusThread* :: it periodically writes the status of the server and clients into a file; all available information is flushed to the status file

c) *CheckMessagesThread* :: it periodically checks the presence of some messages on each client; the semantic of the messages are later explained in the section 3.5 as all of them are related to the fault tolerant mechanism; an associated status parameter is accordingly updated

d) *CheckRunningPIDsThread* :: after the programs are executed on the server and clients, this thread periodically checks if any of the processes are stopped or not; again, there is a flag that is updated after each run that indicates the status of remote processes

e) *FaultTolerantThread* :: periodically checks whether the failing condition is met; for further information, please refer to section 3.5.

3.3. Synchronization problem

As we aim to test I/O frameworks and to measure the response time (respectively, the requests rate), we need a good control of the clients starting time. If we do not synchronize at all, it may happen to get big differences between two consecutive calls of the same test scenario. Moreover, trying to assure that a given request rate is reached within a test is even more difficult. For those reasons, we employed two different levels of synchronization as follows:

a) process level :: using a conditional object to synchronize all the threads within one process. After a thread is spawned and ready to send requests, it increments a counter and then sleeps. The main thread would wake up the already sleeping threads when the counter has reached the expected value. This assures the threads start sending requests roughly at the same time.

b) clients level :: having synchronized the threads within a client program, now we have to synchronize two different remote clients. For this purpose, we use the following technique that is a bit similar to the 2PC protocol: whenever a client has synchronized its threads, it writes a message locally (i.e. an empty file); the coordinator waits until all the clients have created the aforementioned file. If this is the case, it means that all clients are now ready to send requests with all their threads. They now keep on checking if a message from the coordinator arrived and, if so, they start issuing requests to the server. The coordinator sends messages to the synchronized clients by running SSH commands that create empty files.

This synchronization procedure is run before each test takes place.

3.4. Execution flow

This subsection explains the execution flow of our application from the parsing input files to the collection of remote data. The *Coordinator* represents the main thread of our application and it executes all the steps described below. The picture summarizes these steps and it omits few of them (the number in parentheses represents the corresponding step of the figure).

a) parsing input data (1) :: the *server_clients.properties* file is parsed as this file must contain information regarding the JAR programs, the SSH connection details and others for server and clients as well; for further information regarding the content of this file, please have a look at section 4.1.

Also here, the test scenarios properties files are parsed (e.g. test1.properties). Again, further information available under section 4.2.

b) create SSH clients (2) :: using SSHJ library, a re-usable SSH key-based connection is created for each remote machine (server and clients). This step takes place here as an optimization of creating an SSH connection each time a remote command is fired. It saves resources and time as otherwise our tests showed that around 0.6s were spent each time for creating the ssh connection for every remote sent command.

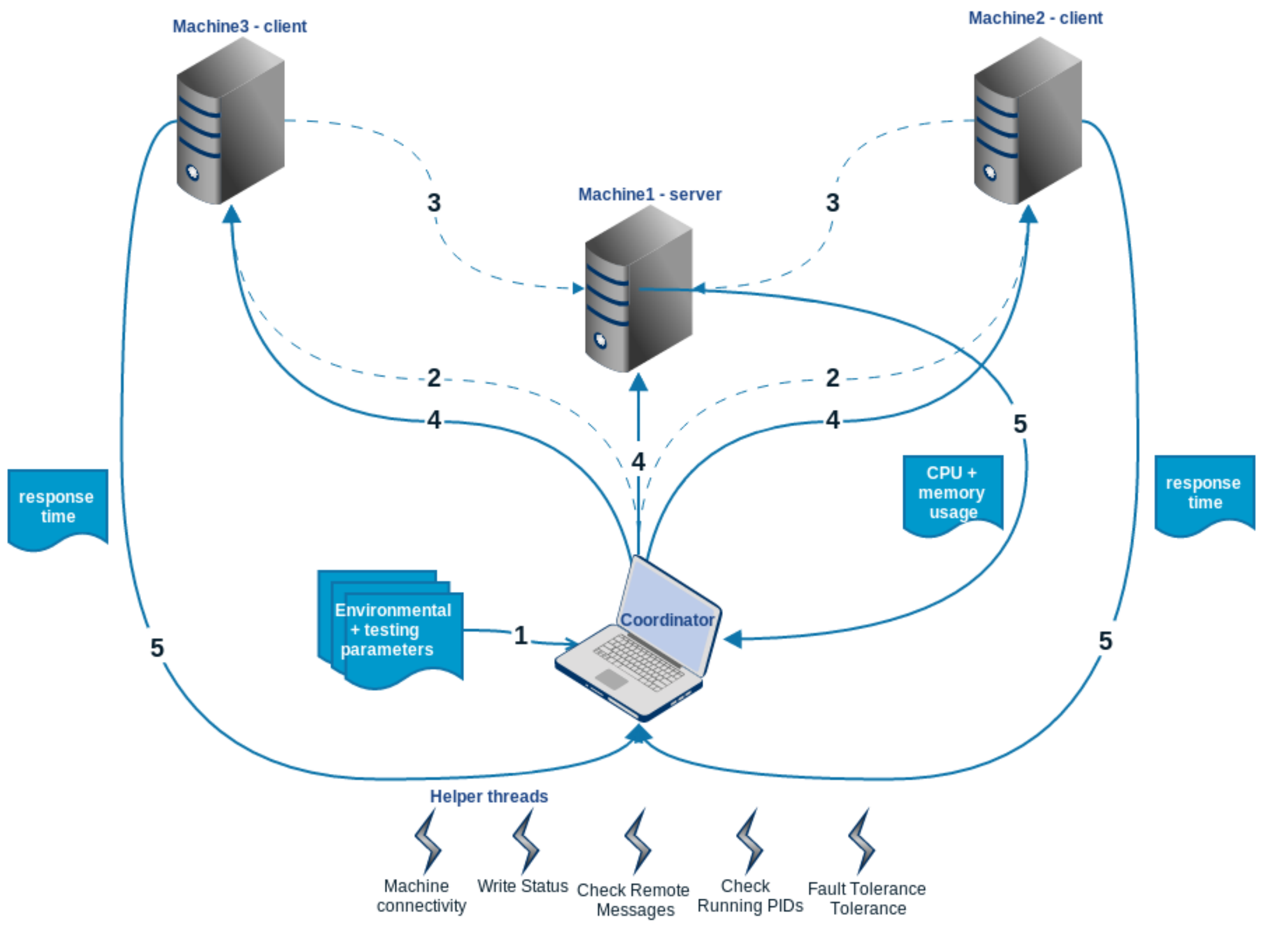
c) check network connectivity (3) :: first of all, we test if any of server and clients IP addresses are set to a local address - either all or none must be local addresses because otherwise a remote machine cannot connect a local one.
Secondly, a test runs to check whether all the clients can access the server. If not, they would not be able to send HTTP requests to a non-reachable remote machine.

d) copy remote files (4) :: now it's time to upload the programs to our remote machines. The uploaded program will reside in the working directory passed as parameter to the properties file.

e) parsing tests scenarios (1) :: finally, the tests scenarios files associated with the intended tests to be run are parsed.

f) running test (4) :: finally, for each test context a *RunClient* thread is created and run. It runs a tests, waits to be done and collects locally the results. This object also is in charge with clients synchronization (see 3.3) and it is fault tolerant (see 3.5).

g) gathering results (5) :: the log files created by the clients and server are downloaded locally. Please have a look at chapter 5 for further information regarding the output files.



3.5. Fault tolerance

The problem is that our envisioned tests will take quite long time, in the order of hours. Because of this and of using a maybe uncontrollable network of machines, faults are possible. Both cases are taken into consideration - either a client is down or the server does not run anymore. To alleviate this problem, a lightweight fault tolerant mechanism has been developed.

The idea is that a different thread of the coordinator runs and periodically checks for faults.

We defined a fault as following:

- a server failed if the process ID (PID) is not running anymore; the helper thread described here 3.2.d) is used for this purpose
- a client failed if the PID is not running anymore and the test status is not done; another condition is that the PID is still running but lately there is not activity in the remote log file (the client must periodically write data in the log file regarding the requests that it runs).

Besides this, there is predefined proportion of the clients that must fail in order the test to be restarted. The parameters are shortly explained. In case the failing condition is met, the running test thread is interrupted and the test is rerun.

There are few parameters that can tweak the behavior of the fault tolerance mechanism as follows:
a) *server.faultTolerant* = yes/no :: this can turn on/off the fault tolerance mechanism

b) *server.restartAttempts* = Integer (e.g. 5) :: how many retrials are run in case a test fails

c) *clients.restartConditionPropThreadsDead* = Double [0,1] (e.g 0.5) :: as described before, a given proportion of the clients must be down in order to a test to be re-run. For example, passing 0.25 as value means that a quarter of the clients must not perform anymore to trigger a restart.

d) *clients.timeoutSeconds* = Integer (e.g. 20) :: if a client program is still running, its log file must be updated at least once this amount of seconds. If not, the client will be considered failed.

4. User guide

The entire project is hosted on github and can be accessible using the following link: https://github.com/teodormacicas/jboss_benchmarkIO. Please clone the latest tag (i.e. v1.1 at the time the document was written).

The repository contains three projects:

- **nio2--xnio-client** :: implements the client that basically issues a set of HTTP requests to the server; it also contains the logic for synchronization
- **nio2-xnio3-netty-servers** :: contains the implementation of servers using all three I/O frameworks. Nio2 and Xnio3 come with two different flavours: synchronous and asynchronous. On the other hand, Netty comes only with asynchronous running mode.
- **testing_tool** :: is the automatic and flexible testing tool developed in the context of the workshop project

Requirements to run:

- Maven >v2.2 must be installed
- because the coordinator uses SSH to remotely run commands, the public-private key pair must be generated

- all the public keys of the clients and server must be known by the coordinator (add them in the ~/.ssh/authorized_keys file)

Commands to build:
- go to nio2-xnio-client and run *mvn -install*
- go to nio2-xnio3-netty-servers and run *mvn -install*
- go to testing_tool and run *mvn -install*

Command to run:
- go to testing_tool project folder
- be sure that you correctly changed the parameters of *server_clients.properties* and *testX.properties* files
- after building the project, run *java -jar testing_tool.jar* ; the properties files are looked in the homepath, local directory and classpath as well.

4.1. Environmental parameters

The below explained parameters reside in *server_clients.properties* file.

a) Server related properties

server.programJarFile	*Path to the server module jar file.* Example: nio2-xnio3-netty-servers/target/nio2-xnio3-test-jar-with-dependencies.jar
server.dataFolder	*Contains files that will be requested via HTTP.* Example: nio2-xnio3-netty-servers/data/
server.sshUserHostPort	*User, IP address and host for ssh connection (used by coordinator).* Example: user@host[:22], port is optional, default is 22
server.listenHostPort	*IP address and port on which the server HTTP server will listen on; clients will send requests to this address.* Example: 127.0.0.1:8085
server.workingDirectory	*The directory used as a working place.* Example: /tmp
server.faultTolerant	*If 'yes' tests will be rerun in case of failure.* Example: no/yes
server.restartAttempts	*Number of restart will be made in case of failure.* Example: 5

b) Client related properties

clients.programJarFile	*Path to the client module jar file.*

	Example: nio2-xnio-client/target/nio2-xnio-client-jar-with-dependencies.jar
clients.sshUserHostPort	*User, IP address and host tuples of clients for ssh connection (used by coordinator). Can have multiple values separated with comma for different clients.* Example: user1@host1, user2@host2
clients.workingDirectory	*The directory used as a working place.* Example: /tmp
clients.restartConditionPropThreadsDead	*Double value representing percentage of clients should be dead to make restart.* Example: 0.5. Condition: values must be in [0,1] interval.
clients.timeoutSeconds	*Clients are considered to have a failure after certain amount of seconds if there is no activity in the log file.* Example: 20
clients.tests	*Tests to be run separated by whitespace. Names of test files.* Example: test1, test2. For each test, a file named "{testName}.properties" must exists. It defines the test scenario.

4.2. Testing scenarios

The testing scenarios are manageable through properties files. For every test that is added to *clients.tests* property, a corresponding file named like {testName}.properties must exists. It must contain the below described parameters in order to create a test scenario.

server.type	*Server type can be one of the following xnio3, nio2 or netty. Single value.* Example: xnio3
server.mode	*Server mode asynchronous or synchronous.* Example: async/sync
test.num	*Number of time each test will be performed.* Example: 2
test.threads.num	*Number of threads that will be created on each client. Single value.* Example: 10
test.request.num	*Number of requests that will be made by each client. Multiple values separated by whitespace.* Example: 1000, 250.
test.delays	*Number of delays between requests. Multiple values.* Example: 150, 170.

5. Output

This section describes briefly what is the output after a test has been run. The output is represented by a bunch of files, some of them remotely downloaded after the test ended. These are as follows:

a) client logs

1. File format: log-client{ID}-{NumberTest}.data;
   - it is the raw client data file, before parsing
   - lines like ‘WRITE 937676697 READ 979499222’ are of interests
   - in case of problems, please check this file; exceptions may be shown here
2. File format: log-RESP_TIME-client{ID}-{NumberTest}.data;
   - it is the parsed client data file
   - contains only the response times of the sent requests

b) server logs

1. File format: log-server{NumberTest}.data
   - it is the raw server data file, before parsing
   - in case of problems, please check this file; exceptions may be shown here
2. File format: log-TOP-server{NumberTest}.data
   - it contains the top unix command output that ran at server side
   - CPU and memory usage information are to be found here
3. File format: log-CPU-server{NumberTest}.data
   - it contains only the CPU usage information after the previous file has been parsed
4. File format: log-MEM-server{NumberTest}.data
   - it contains only the memory usage information gathered by the top command

c) status logs

- this is the log file of the testing tool
- it is periodically appended with information about the running remote programs, server and clients, as well as tests

6. Results

6.1 Test conditions

To perform tests we have the same conditions for NIO.2, XNIO.3 and Netty frameworks. To run the server and the client applications we used two 64bit 8-core machines (the processor belongs to the x86_64 family and its clock rate is 3 GHz) with 4 GB of available memory).

The machines are connected to network switch through the Gigabit Ethernet network (link capacity around 100 MB/sec). Each machine has Java Virtual Machine version 7 with “RHEL-4” operating system to run applications. The number of threads of the client is constant and equal to 1000 threads. Every thread makes approximately 100000 requests to the server.

Figure 1 illustrates a thread activity on a timeline. The thread makes a request and waits for a response from a server, we call waiting time as a response time. When the thread receives a response it sleeps for a certain amount of time (delay time) to make another request.

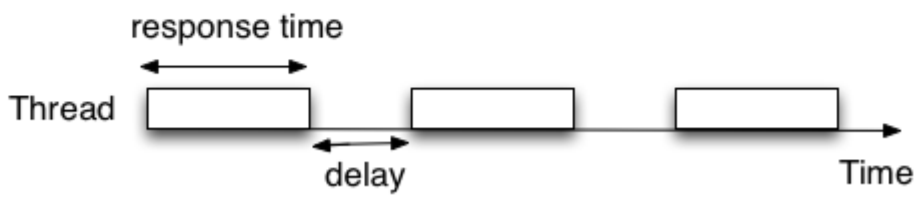


Figure 1. Thread activity with time.

Figure 2 illustrates all threads activities on a timeline. We calculate request load as a number of all requests to time required to issue all requests. By varying delay time we can change the request load and estimate the response time and CPU consumption as its function. To approximate the response time distribution, we fit Gamma distribution because for low rates it was difficult to fit other functions.

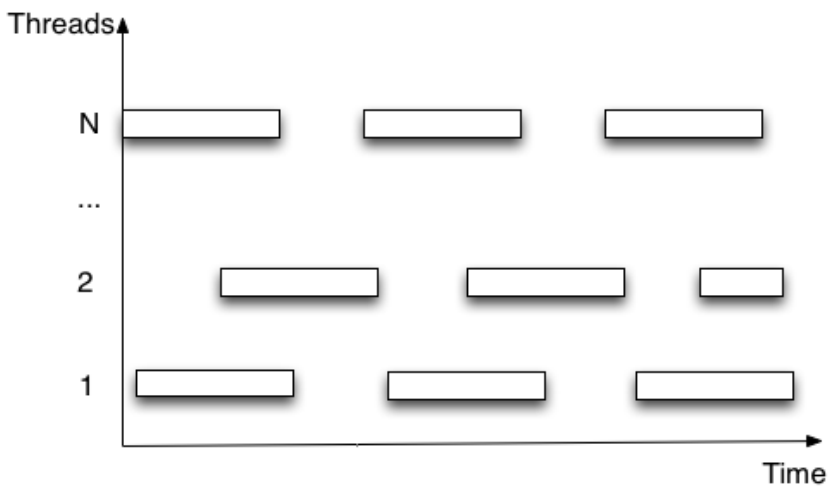


Figure 2. All threads activities with time

6.2 Testing

We are interested in the comparison of frameworks during high request loads so we performed a series of tests with low delay times (delay times are given in ms):

test.delays = 250 225 200 175 150 125 100 75.

For each delay we computed the request load and fit time response distribution with Gamma function. We then considered the response time as a function of the request load. We used scripts implemented by previous research for all calculations. Figure 3 illustrates results for NIO.2, XNIO.3 and Netty server applications running in an asynchronous mode.

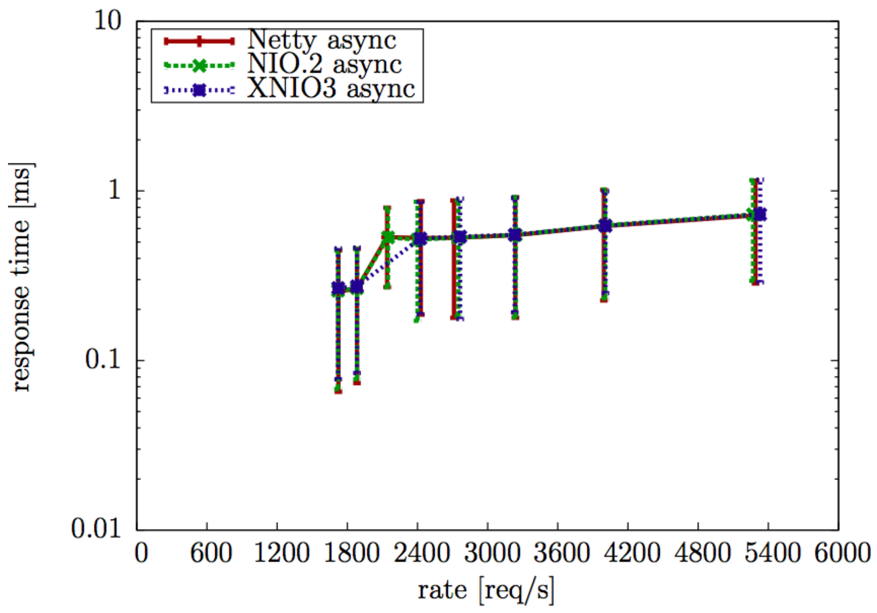


Figure 3. Response time to load for NIO.2, XNIO.3 and Netty.

Figure 4 illustrates results for NIO.2 and XNIO.3 server applications running in a synchronous mode.

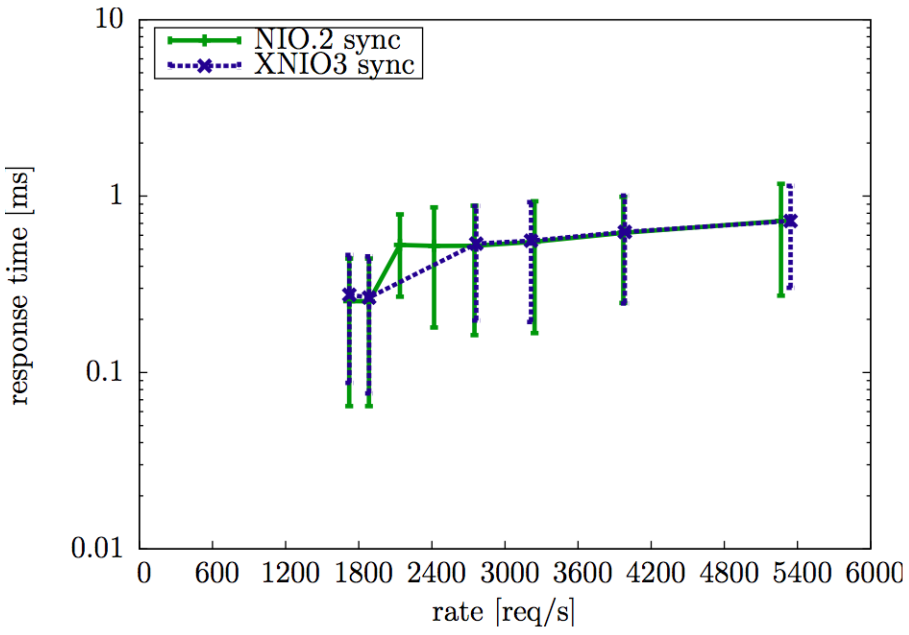


Figure 4. Response time to load for NIO.2, XNIO.3 and Netty.

It is clear that Figure 3 and Figure 4 cannot show reasonable difference in request handle time for different frameworks due to a big variance for each request rate. However we can conclude that the response time increases with the request load and we can determine the range in a response time for a particular request load.

We calculated CPU load percentage as a difference between a maximum load and an idle load. Figure 5 illustrates CPU usage for different frameworks. Again it is difficult to make comparison on CPU usage for different frameworks due to a big variance for each request rate.

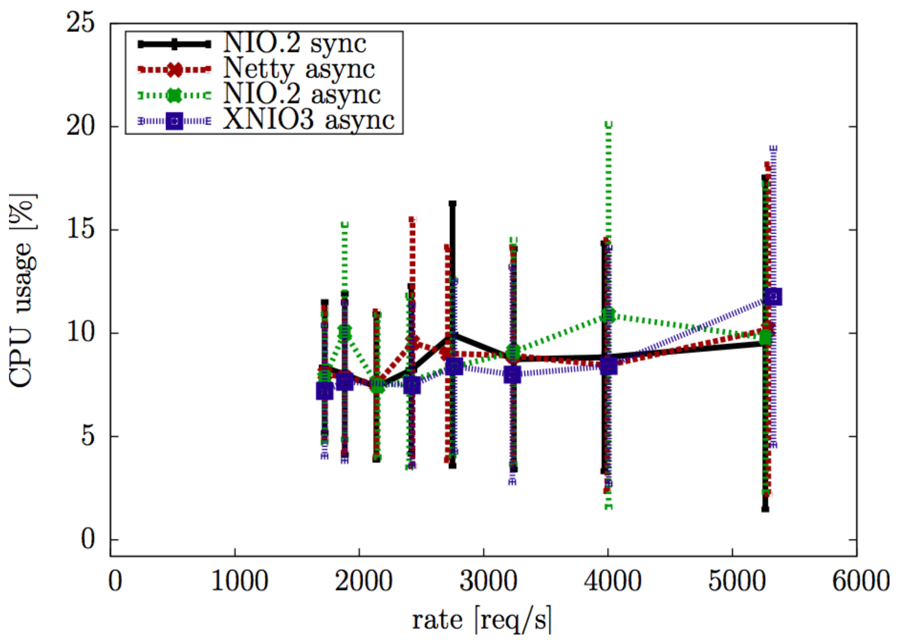


Figure 5. CPU usage to request load.

7. Conclusion

In this work, we have implemented an automated tool to compare modern I/O frameworks, that are NIO.2, XNIO.3 and Netty, in terms of different performance characteristics, such as response time, CPU usage and memory consumption. The tool is extendable to add new I/O frameworks and flexible to perform different test scenarios.

We have run some tests and analyzed results, however we were not be able to identify any difference in performance measures due to big variance in results.