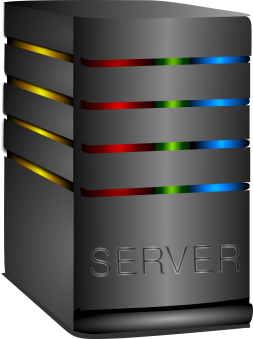
USERGUIDE

**Performance Measurements**

# About

The aim of the performance measurement framework is to allow the user to evaluate the performance limits of a given FEP system. Therefore, it provides means to measure roundtrip times of a client –server configuration. The server may be situated on the same or on a remote machine.

Performance measure calculates the roundtrip time starting at the client (application layer) which sends packets with timestamp t0, the server receives the packets with timestamp t1, and sends them back after processing the packet and adding timestamp t2. Finally, the client receives the packets at the interface with timestamp t3. The roundtrip time is calculated as t3 – t0.

t1

t0

Request

Response

t3

t2

**CLIENT**

Figure 1: Sketch of a measurement setup

For a more detailed measurement, measuring points are spread across the different layers. They reflect how long it takes the middleware to process a data sample on its way towards the transmission layer, and likewise how long it takes the middleware to supply the sample to the user after its reception.

Please note that measuring points can only be used with a single signal, since it is not recorded which signal triggered a measuring point.

# Carry out a measurement

The performance measurement is provided by a command line executable, located in the folder

*test/stimuli/perf\_measure\_test*

relative to your install directory. It may be executed by

*$ ./perf\_measure\_stimuli* (Linux)

or

› *perf\_measure\_stimuli* (Windows)

You can stop the measurement by pressing CTRL+C. The application will print a measurement statistic before exiting.

The executable has several command line arguments, which are optional and described in the next paragraph.

**Arguments:**

**-?, --help, -h** shows you the help menu with different settings for carrying out a measurement, providing short explanations

**-<n>** sets the numeric identifier of the signal so you can define several different signals for one measurement. “-0” defines a signal with id 0, and the following arguments will further define the signal.

**-t <…>** sets the transmission adapter. Possible adapters are “connlib”and “dds” (default) e.g. “–t connlib”

Diese Option wird nicht mehr unterstützt.

**-C, --client** puts the application in client mode so that it sets the timestamps *t0* and *t3*

**-S, --server** puts the application in server so that it sets the timestamps *t1* and *t2*

**-r, --receiver** developer option, do not use

**-s, --sender** developer option, do not use

**--type <…>** sets the type of the FEP Signal to be used. If a DDL is provided (see “--ddlpath”), the signal type will be used to resolve the struct from the description file. Otherwise, a new signal description will be generated. The resulting signal will contain a payload so that the overall size corresponds to the packet size defined by parameter “-b”.

(Client: default signalttype = “--type Ping”,

Server: default signaltype = “--type Pong”)

**-e <…>,**

**--element <…>** sets the name of the FEP element. Only important for the transmission of data during simulation. (Identification for an element outwards of system)

**--domain <…>** sets the FEP Domain Id of the transmission layer. For measurements, you should use an unoccupied domain where you have no interference from other FEP elements.

e.g. “--domain 68”

**--ddlpath <…>** option to set the path of another DDL signal description file

**--serialize <…>** option to deactivate serialization mode. Valid options are “RAW” and “DDL” (default: “DDL”, i.e. serialization is activated)

**--cpu-stress <…>** creates a cpu stress if you set more than one cpu (default: 1)

“--cpu-stress 2” defines usage of two cpus for stress testing. This option is used to simulate real scenarios where some computation is done within the fep element.

**-m <…>,**

**-ddbsize <…>** sets a DDB buffer e.g. “-m 128” creates a buffer with a size of 128 frames. Accordingly, 128 frames will be sent and only the last one will be provided with a sync flag. By default, no buffer is initialized.

**-d <…>,**

**--delay <…>** sets a sending delay in μs for every packet,

“-d 300000” defines that a packet will be sent every 300000 μs. The default value is 100 μs.

This option is complementary to “-f”

**-f <…>** sets a sending delay with frequency f in Hz (default 10 kHz), “-f 300” defines a packet sending frequenzy of 300 Hz.

This option is complementary to “-d”.

Please note that the operating system limits the maximum

possible frequency. As a rule of thumb do not use

frequencies above 1000 Hz. If a higher rate is required

increase the number of packets sent within a single frame

(see option “-N”: Number of packets).

**-b <…>** sets the size of a packet in bytes, e.g. “-b 256” defines a packet with a size of 256 byte. By default is 64 byte initialized and minimum packet size is 32 byte

**-ti <…>** sets the time for the measurement to take, in seconds (default is infinite). After the specified time has passed, the application will terminate automatically. “-ti 10” defines that a measurement will take ten seconds.

**-N <…>** sets the number of packets which will be transmitted concurrently in a burst. “-45” defines that 45 packets will be sent in short time intervals.

**–v <…>** sets the verbosity. The default: logging level is 1. “-v 10” sets the highest logging level. Logging level range is from 1 to 10.

**–q, --quiet** flag which sets the verbosity to zero. (Default is 1)

**-Z** changes the format of statistics output. (Default: false)

The statistics output contains special markers to enabled parsing the contents by automated measurements scripts .

---statistics---- … packets transmitted, … received, … packets lost, Rtt min/avg/max = …/…/ us

If *–Z* is set:

===PACKET\_STAT===;n;o;p ===RTT\_STAT===;min;avg;max

(n = number of packets transmitted, o = numer of packets received, p = number of packets lost)

**-P <…>** sets the process priority, value ranges from 1 to 31. “-P 1“ sets the process to the highest priority. By default is priority 8 initialized.

**--results** <…> sets the name of result output file (default: “results.csv”) e.g.: –results measure\_1 creates the output file measure\_1.csv with timestamps *t0 … t3* in directory of the executable. If measure points are set active, the output file measure\_1\_framework\_measure.csv will created

# Measurement Files

Measurements are stored in csv-files. The results of the roundtrip measurements (see Chapter 1) are stored in a file “results.csv” created in the current working directory. If the distributed data buffer is used an additional file “results\_ddb.csv” is created. If measure points are activated an additional file “results\_framework\_measure.csv” will be created.

## Roundtrip-Measurement File: “results.csv”

The roundtrip measurement file is named “results.csv” and saved to the current working directory after the measurement is completed. This file contains the timestamps t0 to t3, along with some additional information. The columns are labeled SignalName, SeqNr, ClientSend, ServerRecv, ServerSend, ClientRecv.

***SignalName*** name of Signal (Default or changed by command line)

***SeqNr*** consecutive number identifying each packet

***ClientSend*** timestamp *t0, Client Sending Timestamp*

***ServerRecv*** timestamp *t1, Server Receiving Timestamp*

***ServerSend*** timestamp *t2, Server Sending Timestamp*

***ClientRecv*** timestamp *t3, Client Receiving Timestamp*

This measurements can be used to calculate some considerable values:

**RTT** Round Trip Time. Calculated as: RTT = t3  – t0

**STT** Single Trip Time. Estimated as STT = RTT / 2

**TTSender**  Send Trip Time. Calculated as: TTSender = t1 – t0

**TTReceiver**  Receive Trip Time. Calculated as: TTReceiver = t3 – t2

**WT**  Work Time. Calculated as: WT = t2 – t1

Please note that TTSender and TTReceiver can only be calculated correctly if client and server are running on the same host. This is due to the fact that the clocks on different hosts aren’t synchronized with the required accuracy.

If packages are lost the timestamps t1, t2 and t3 are missing and marked with the value “0”.

As this measurement file is in CSV format a spreadsheet application can be used to import the data and to perform the described calculations on them.

## Roundtrip-Measurement File: “results\_ddb.csv”

The roundtrip measurement for file for ddb is named “results\_ddb.csv” and saved to the current working directory after the measurement is completed. This file contains the timestamps t0 and t3  along with some additional information. The columns are labeled SignalName, FrameNr, ClientSend, ClientRecv, IsComplete, NumValidSamples.

***SignalName*** name of Signal (Default or changed by command line)

***FrameNr*** number of signal inside the frame

***ClientSend*** timestamp *t0, Client Sending Timestamp*

***ClientRecv*** timestamp *t3, Client Receiving Timestamp*

***IsComplete*** Boolean flag indicating wether frame was complete

***NumValidSamples*** Number of valid samples within the frame

This measurements can be used to calculate the Roundtriptime:

**RTT** Round Trip Time. Calculated as: RTT = t3  – t0

The IsComplete-Flag and the NumValidSamples are indicating if the frame was received completely with the correct number of valid samples.

As this measurement file is in CSV format a spreadsheet application can be used to import the data and to perform the described calculations on them.

## Measurement Points File: “results\_framework\_measure.csv”

To get measurement points it is required to activate measurement points during the build process of the FEP SDK or to use binaries with measurement points activated. As doing the measurements might possibly have a noticeable impact on the performance, they are not activated by default.

## Activating Measurement Points

To activate measurement points it is required to set the CMake-Variable “FEP\_ENABLE\_PERFORMANCE\_MEASUREMENT” to value “1”, “on” or “true”. On the command line this is done using the following option for CMake

*$ cmake –G … -DFEP\_ENABLE\_PERFORMANCE\_MEASUREMENT=1”*

The results file for the measurement points is named “results\_framework\_measure.csv” and also located in the current working directory.

Alternatively, a binary with activated measurement points may be obtained from AEV.

## Measurement Points File Format

The generated csv-file contains the timestamps of the measuring points, along with some additional information. The location of the measuring points is also illustrated in Figure 2.

***\****) ***IndexOfSendPacket***

consecutive number identifying each transmitted packet

1. ***ElementTransmitCalled***

timestamp just before fep::IUserDataAccess::TransmitData is called

1. ***ApiTransmitCalled***

timestamp after “TransmitData” has been called using FEP API

1. ***DdsTransmitBeforeLocking***

timestamp before locking sample for transmission

1. ***DdsTransmitBeforeSerialize***

timestamp before serializing the sample with ddl

1. ***DdsTransmitLowLevel***

last timestamp before sample is handed over to the transport layer

\*) ***IndexofReceivedPacket***

consecutive number identifying each received packet

1. ***DdsReceivedLowLevel***

first timestamp after sample is handed over from transport layer to FEP

1. ***DdsReceivedAfterLocking***

timestamp after the received sample has been locked.

1. ***DdsReceivedAfterDeserialize***

timestamp after the sample has been deserialized

1. ***ApiReceivedCalled***

timestamp just before the sample is about to be distributed to the user using the FEP API

1. ***ElmentReceivedCalled***

timestamp after the sample has been delivered to the user by the “Update” callback.

Attention:

* The measurement points are only useful for a single signal. More signal measurements interfere with each other and the result is not useful at all.
* Please do not use/publish performance numbers with activated measurement points. The results might be considerably lower than the actually possible performance.

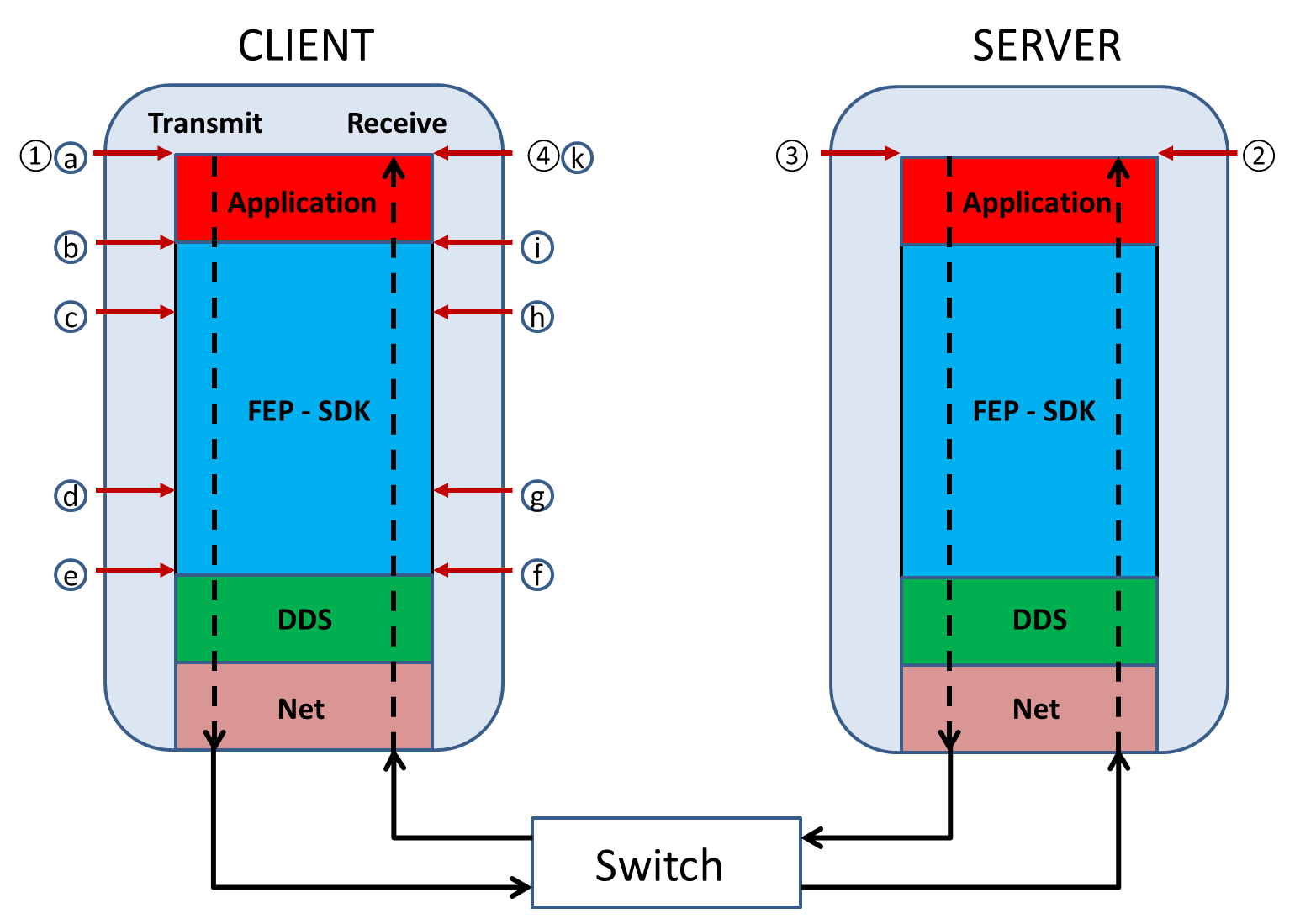


Figure 2: ① ClientSend ② ServerRecv ③ ServerSend ④ ClientRecv

## Options to calculate:

**Processingtime Client before sending:**

**WT**[Client before transmitted]= DdsTransmitLowLevel – ElementTransmitCalled

**Processingtime Client after receiving:**

**WT**[Client after receiving] = ElmentReceivedCalled – DdsReceivedLowLevel

# Configuration Examples

**Example 1:** Single signal

Server mode:

*$ perf\_measure\_stimuli -S -0 -b 512*

The server sends packets with a size of 512 byte after receiving them from a client with signal number 0.

Client mode:

*$ perf\_measure\_stimuli -C --time 10 -0 -f 300 -b 512*

The client sends creates a signal (named “Signal-0”) using a sample size of 512 bytes sent with a frequency of 300 Hz. The measurement will run for 10 seconds.

Please note:

* **Server has to be started first since it is the passive part of the setup**
* **Server configuration does not use nor require the frequency option as it just responds to incoming packets by sending them after processing.**
* **The Configuration of client and server must match**

**Example 2:** Two signals with different configurations

The arguments using numbers like “-0” or “-1” sets a signal number. Some arguments like “-f” (frequency), “-b” (sample size) are only relevant to the current signal, others like “—time” are overall options.

Client mode:

*$ perf\_measure\_stimuli -C --time 10\*

*-0 -f 300 -b 512\*

*-1 -f 480 -b 1052 -m 128*

The client uses two signals, one with id “0” of size 512 bytes which is sent at a frequency of 300Hz. The second signal with id “1” is of size 1052 bytes and will be sent at 480Hz. A DDB buffer of size 128 is used for the second signal.

Server mode:

*$ perf\_measure\_stimuli -S -0 -b 512 -1 -b 1052 -m 128*

The configuration needs to match the client configuration for all given signals.

Please note:

* **If measurement points are activated in the FEP SDK, it is not possible to run a configuration with more than one signal.**
* **The server will run for an infinite time. To terminate the server press “CTRL-C” or kill the process using the “Task Manager” (Windows) or the kill-Command (Linux).**

# Python script for automated measurements

For sufficient performance results the execution of different configurations might be needed. This task is done by an additional script “perf\_measure\_scenarios.py” distributed along with the performance measurement executable “perf \_measure\_stimuli.exe” (Windows) or “perf \_measure\_stimuli” (Linux).

## Predefined Scenarios

The measure script contains various different usage scenarios:

**Scenario 1: Varying Signal Sizes (“Scenario\_Different\_SignalSizes”)**

This scenario is used to measure performance effects of different signal sizes.

**Scenario 2: Varying Transmission Frequencies (“Scenario\_Different\_Frequencies”)**

This scenario is used to measure performance effects of different transmission frequencies.

**Scenario 3: Varying Number of Signals (“Scenario\_Different\_NumberOfSignals”)**

This scenario is used to measure performance effects of different number of signals.

**Scenario 4: Distributed Data Buffer Transmission (“Scenario\_Different\_UsingDDB”)**

This scenario is used to measure performance effects of using DDB (Distributed Data Buffer) for transmissions.

**Scenario 5: Burst Transmission (“Scenario\_Different\_NumberOfBursts”)**

This scenario is used to measure performance effects of sending signals in burst mode.

## Customizing Scenarios

The scenarios are defined within the measurement script and can be adapted.

A typical scenario definition inside the measurement script (see “perf\_measure\_scenarios.py”) looks like:

class Scenario\_Different\_SignalSizes(Generic\_Scenario):

def \_\_init\_\_(self):

Generic\_Scenario.\_\_init\_\_(self)

self.name= "Scenario\_Different\_SignalSizes"

for i in range (1):

self.signals.append({

"bytes": [1024, 4\*1024, 4\*4\*1024, 4\*4\*4\*1024, 4\*4\*4\*4\*1024],

"frequency": [1000],

"ddbsize": [0],

"numpercycle": [1]

})

A scenario consists of settings concerning:

* the number of signals used
* the signal sizes
* the transmission frequency
* distributed data buffer
* burst mode settings

The user may take these definitions as a blueprint for the creation of own additional scenarios:

Class **name\_of\_your\_scenario** (Generic\_Scenario):

def \_\_init\_\_(self):

Generic\_Scenario.\_\_init\_\_(self)

self.name= “**name\_of\_your\_scenario**”

self.common\_args[‘**Argument\_1**’] = ‘**option**’

self.server\_args[‘**Argument\_2**’] = ‘**option**’

self.client\_args[‘**Argument\_3**’] = ‘**option**’

…

for i in range (**number\_of\_signals**):

self.signals.append({

“**Argument\_4**”: [**value**],

})

The arguments are the same as for the main application. Options listed as “*common\_args*” are processed for both the client and the server, “*server\_args*” are server specific, and “*client\_args*” are client specific.

For the “*signals*” option the user might also provide a list of options, e.g. *“frequency”: [100, 500, 1000]*. This will be processed such that every possible combination of options is generated. This will result in not only one, but several test runs at once.

**Note:**

* **As the measurements script permutes over all combinations of the signal parameters, a huge set of measurements may be generated.**

## Selecting Scenarios

After defining a measurement scenario, the user can define a list of scenarios to select.

Test\_Scenarios= [ …, **name\_of\_your\_scenario, …** ]

## Running Measurements

The script may be invoked by the command

*$ ./* perf\_measure\_scenarios.py(Linux)

or

› python perf\_measure\_scenarios.py(Windows)

**Note:**

* **The script is written to run with python version 2 or version 3.**

After execution of the script result files for the different scenarios are created. For each scenario a directory is created inside the current directory. Within this directory one or more csv-files are generated.

## Evaluating Results

The result files can be evaluated using standard spreadsheet programs (e.g. Excel). Import one of the csv-Files into the spreadsheet (delimiter is “;”) and insert formulas to calculate the timings.