

Nexus Implementation at DESY

Virtual Laboratories – PANDATA WP5 / HDRI

Jan Kotański, Thorsten Kracht, Eugen Wintersberger

Deutsches Elektronen-Synchrotron

pandata_{europe}



May 23, 2013

Why Nexus ?

- The **internal structure** of the Nexus files can be adopted to specific experimental techniques
- **Metadata** can be stored for a complete description of the measurement
- Nexus can store **many image frames** that are created by a measuring sequence in a single file.

This way **data** can be managed efficiently

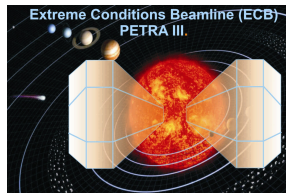
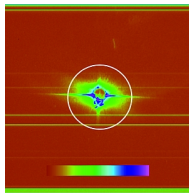
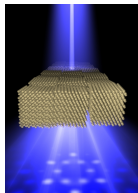


Nexus at PETRA III beamlines

Why Nexus ?

- The **internal structure** of the Nexus files can be adopted to **specific experimental techniques**
- **Metadata** can be stored for a complete **description** of the measurement
- Nexus can store **many image frames** that are created by a **measuring sequence** in a single file.

This way **data** can be managed efficiently



Why Nexus ?

- The **internal structure** of the Nexus files can be adopted to **specific experimental techniques**
- **Metadata** can be stored for a complete **description of the measurement**
- Nexus can store **many image frames** that are created by a **measuring sequence in a single file.**

This way **data** can be **managed efficiently**



Simply creation of Nexus files in C++ (HDRI project)

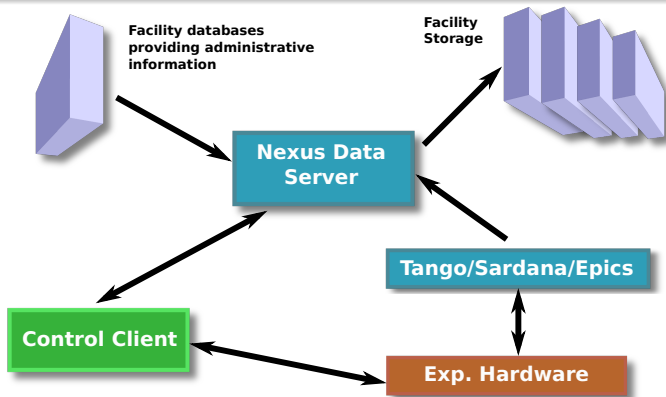
- `libpnicore` provides
 - types of well defined size
 - templates for buffers and arrays
 - reader code to import data from proprietary formats
- `libpniio` classes
 - write Nexus files using HDF5 as its storage back-end
 - make the development independent of the Nexus API
- `python-pniio` Python bindings via the Python package

All libraries are actually in use to develop the software required to establish Nexus as a data format at DESY.

<http://code.google.com/p/pni-libraries/>

<http://www.pni-hdri.de>

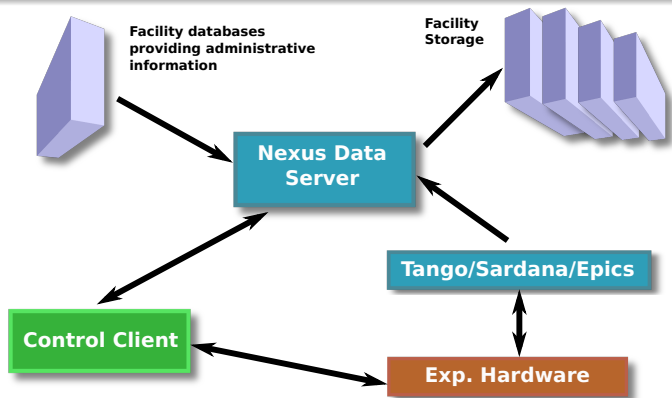
Nexus Data Tango Server - NexDaTaS



We **move** the **responsibility** for data IO out of the control client (CC) into **a separate Tango server**:
the Nexus Data Server (NexDaTaS)

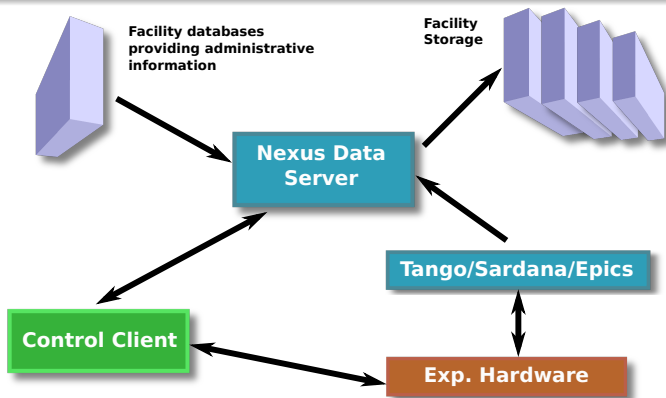
<http://code.google.com/p/nexdatas/>

Nexus Data Tango Server - NexDaTaS



To **Communicate** with this **server** the **control system** must only be aware of **TANGO** for which **bindings** exist to many languages

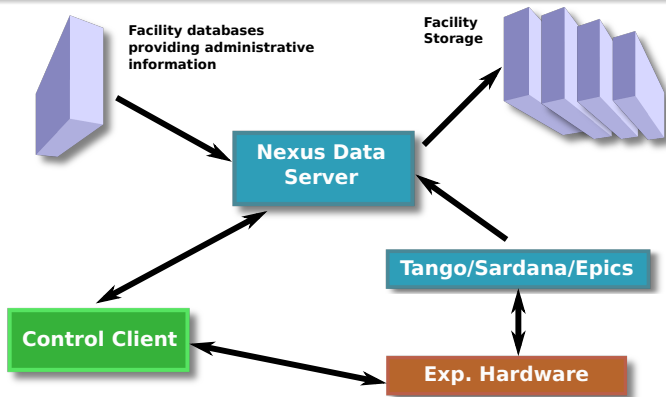
Nexus Data Tango Server - NexDaTaS



The **server** creates Nexus files and fills its field and attributes from various **data sources (DS)**

- directly from the **CC** (also **SARDANA**) using **JSON strings**
- other **TANGO** servers
- **databases**
- **external DS**

Nexus Data Tango Server - NexDaTaS



As **configuration data** the server **needs to know**

- **structure of the Nexus tree** for a **particular entry**
- **datasources** for **each field** in the tree
- **strategy** **when and how** the data is **fetches/written**

Nexus server configuration as NXDL code includes extra tags for storage strategy and data sources

The `<strategy>` attributes:

- `mode` – when the data is fetched:
 - `INIT` – during opening a new entry
 - `STEP` – when the `record()` command is performed
 - `FINAL` – at the time of closing the entry
 - `POSTRUN` – during post-processing stage
- `trigger` name of the related trigger in STEP mode

The `<datasource>` attributes:

- `type` – type of data source:
 - `CLIENT` – communication with client via JSON strings
 - `TANGO` – data from Tango servers
 - `DB` – data from databases

Nexus server configuration as NXDL code includes extra tags for storage strategy and data sources

The `<strategy>` attributes:

- `mode` – when the data is fetched:
 - `INIT` – during opening a new entry
 - `STEP` – when the `record()` command is performed
 - `FINAL` – at the time of closing the entry
 - `POSTRUN` – during post-processing stage
- `trigger` name of the related trigger in STEP mode

The `<datasource>` attributes:

- `type` – type of data source:
 - `CLIENT` – communication with client via JSON strings
 - `TANGO` – data from Tango servers
 - `DB` – data from databases

TANGO datasource in STEP mode

```
<field name="tth" type="NX_FLOAT" unit="degree">
  <strategy mode="STEP" trigger="trigger1"/>
  <datasource type="TANGO">
    <device hostname="haso.desy.de"
      member="attribute"
      name="p09/motor/exp.01"
      port="10000"/>
    <record name="Position"/>
  </datasource>
</field>
```

TANGO datasource in STEP mode

```
<field name="tth" type="NX_FLOAT" unit="degree">
  <strategy mode="STEP" trigger="trigger1"/>
  <datasource type="TANGO">
    <device hostname="haso.desy.de"
      member="attribute"
      name="p09/motor/exp.01"
      port="10000"/>
    <record name="Position"/>
  </datasource>
</field>
```

Extra Tag Examples

CLIENT datasource

```
<datasource type="CLIENT">  
  <record name="counter_1"/>  
</datasource>
```

DataBase datasource

```
<datasource type="DB">  
  <database dbname="tango" dbtype="MYSQL"  
    hostname="haso.desy.de"/>  
  <query format="SPECTRUM">  
    SELECT pid FROM device limit 10  
  </query>  
</datasource>
```

CLIENT datasource

```
<datasource type="CLIENT">  
  <record name="counter_1"/>  
</datasource>
```

DataBase datasource

```
<datasource type="DB">  
  <database dbname="tango" dbtype="MYSQL"  
    hostname="haso.desy.de"/>  
  <query format="SPECTRUM">  
    SELECT pid FROM device limit 10  
  </query>  
</datasource>
```

Example client code

A **short example** should show how **easy** it is to use the server from a client application:

```
import PyTango
device = "p09/tdw/r228"
dpx = PyTango.DeviceProxy(device)
dpx.Init()

# open a new file to store data in
dpx.FileName = "test.h5"
dpx.OpenFile()

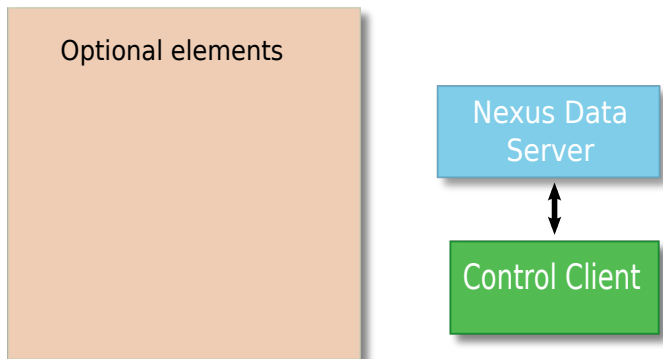
# send configuration for a new entry
# and write initial data (strategy type INIT)
xml = open("configuration.xml", 'r').read()
dpx.TheXMLSettings = xml
dpx.TheJSONRecord = '{"data": {"parameterA":0.2}}'
dpx.OpenEntry()
```


Example client code

```
# experiment main loop
# (write data with strategy STEP)
for i in range(100):
    dpx.Record('{"data": {"counter.exp01":0.1*i,
                          "counter.exp02":1.1*i}}')

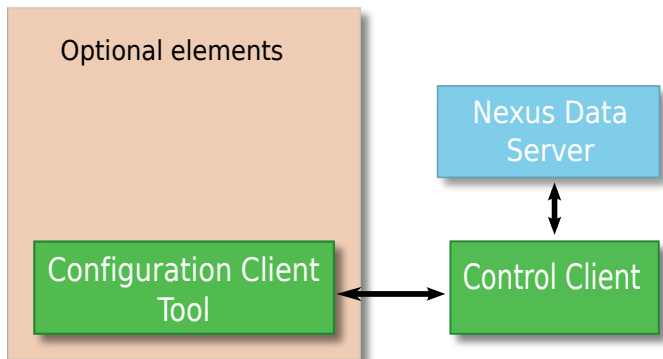
# write final data (strategy FINAL)
# - close the entry - close the file
dpx.TheJSONRecord = '{"data": {"parameterB":0.3}}'
dpx.CloseEntry()
dpx.CloseFile()
```

Configuring the server



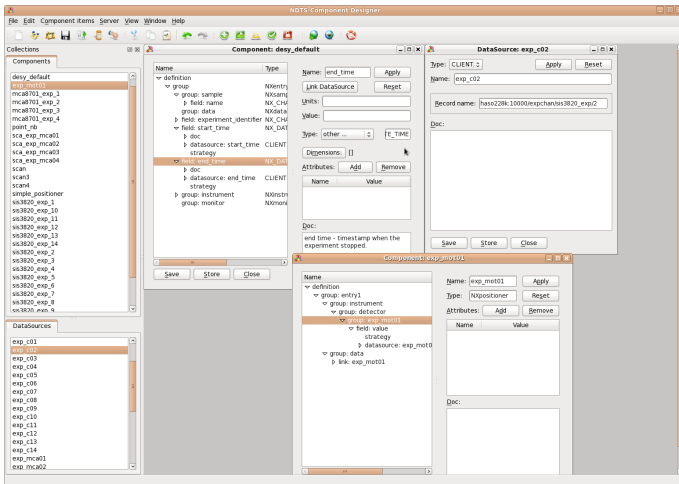
For quite **static** and **not too complex experiments** creating the **configuration NXDL stream** for the Nexus Data Server should be easy and **could be done by the control client**

Configuring the server



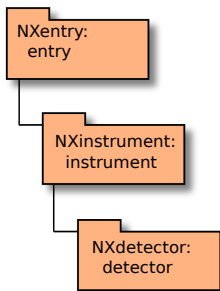
For **complex experiments** we **shift**
the **complexity of the client code** from writing the Nexus file
into creating an **advanced NXDL stream**

Component Designer



The Configuration Client Tool which allows to create XML configuration files, separate components as well as datasources

Components and their merging

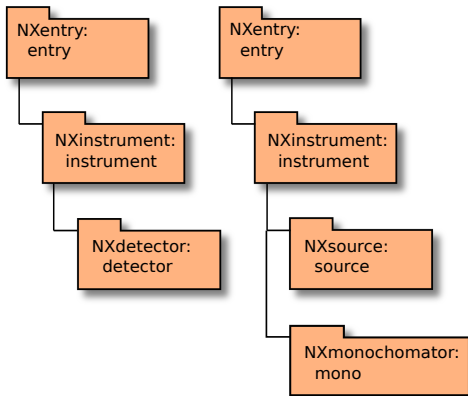


First
Component

Second
Component

Merged
Components

Components and their merging

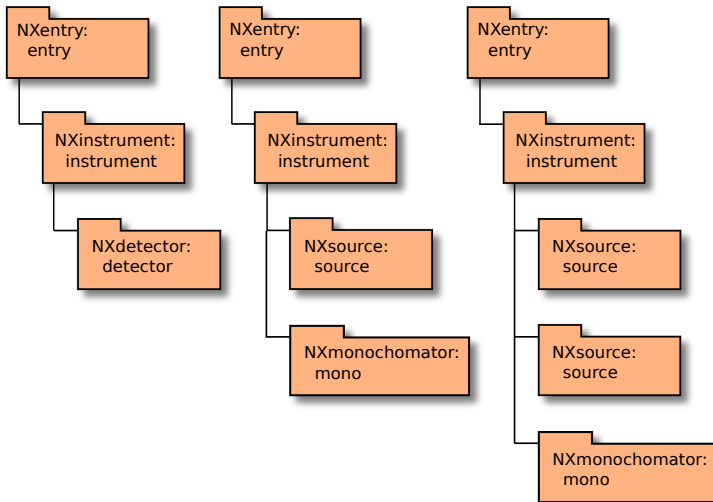


First
Component

Second
Component

Merged
Components

Components and their merging

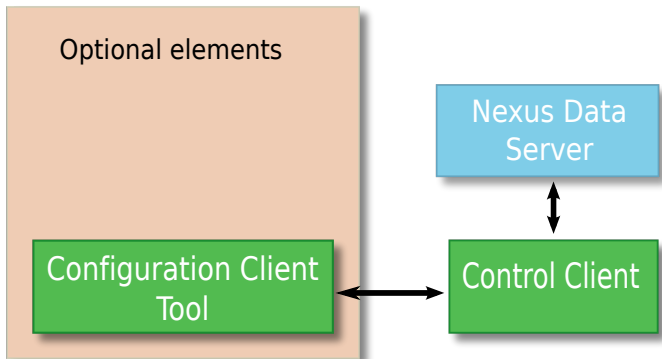


First
Component

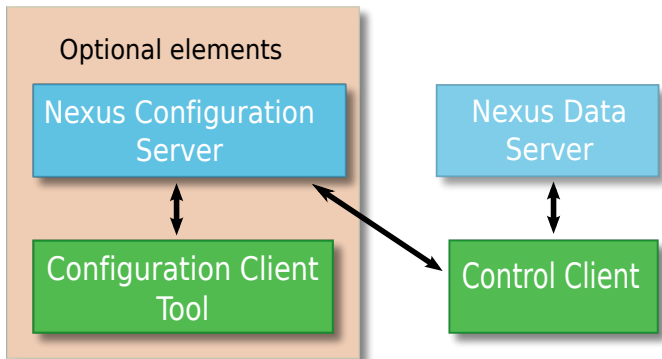
Second
Component

Merged
Components

Nexus Configuration Server



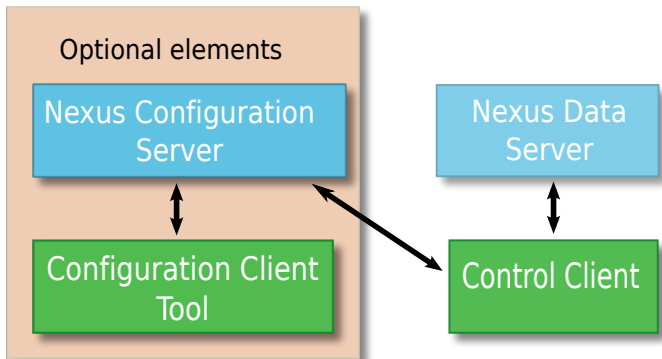
Nexus Configuration Server



Configuration Server

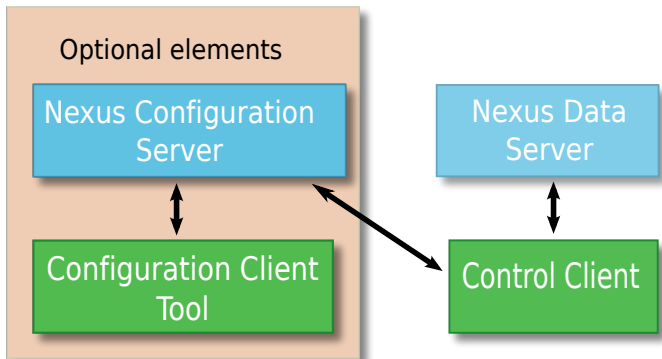
- manages different beamline configurations
- provides the required configuration stream to the Control Client (CC)

Nexus Configuration Server



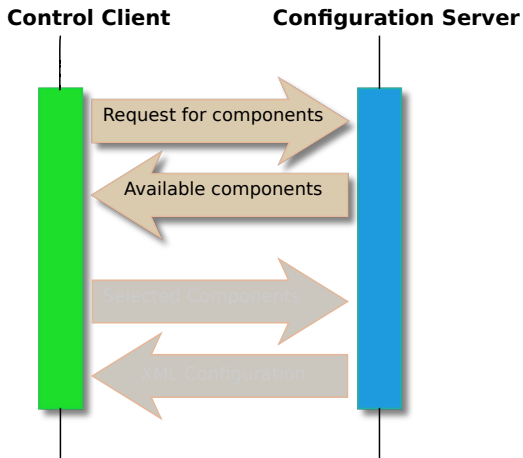
- acts as a central storage facility for experiment configurations
- several Control Clients on a beamline can rely on the same configuration data

Nexus Configuration Server

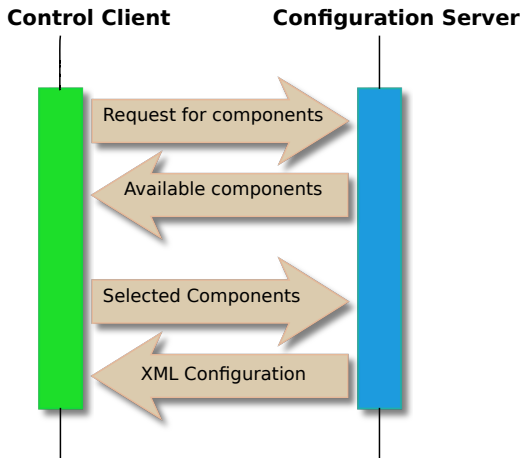


- back-end MySQL DB contains components and datasources
- when final configuration is built components are filled with required datasources and merged

Nexus Configuration Server



Nexus Configuration Server



Control client does **not** have to **take care**
about **creating the configuration**

Currently we are **deploying** the software at P02/P03 PETRA III beamlines

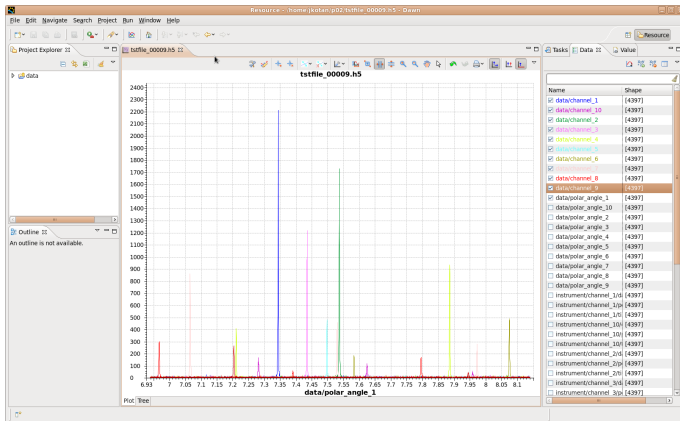
The screenshot displays a software interface with a tree view of data. The tree structure is as follows:

- NeuSConfigurationLegs (Group)
 - NeuS_entry_1_30% (Dataset [1], String, length = variat)
 - entry (Group)
 - data (Group)
 - channel_1 (Dataset [4397], 64-bit unsigned intge, 34.35 KB)
 - channel_10 (Dataset [4397], 64-bit unsigned intge, 34.35 KB)
 - channel_2 (Dataset [4397], 64-bit unsigned intge, 34.35 KB)
 - channel_3 (Dataset [4397], 64-bit unsigned intge, 34.35 KB)
 - channel_4 (Dataset [4397], 64-bit unsigned intge, 34.35 KB)
 - channel_5 (Dataset [4397], 64-bit unsigned intge, 34.35 KB)
 - channel_6 (Dataset [4397], 64-bit unsigned intge, 34.35 KB)
 - channel_7 (Dataset [4397], 64-bit unsigned intge, 34.35 KB)
 - channel_8 (Dataset [4397], 64-bit unsigned intge, 34.35 KB)
 - channel_9 (Dataset [4397], 64-bit unsigned intge, 34.35 KB)
 - polar_angle_1 (Dataset [4397], 64-bit floating-point, 34.35 KB)
 - polar_angle_2 (Dataset [4397], 64-bit floating-point, 34.35 KB)
 - polar_angle_3 (Dataset [4397], 64-bit floating-point, 34.35 KB)
 - polar_angle_4 (Dataset [4397], 64-bit floating-point, 34.35 KB)
 - polar_angle_5 (Dataset [4397], 64-bit floating-point, 34.35 KB)
 - polar_angle_6 (Dataset [4397], 64-bit floating-point, 34.35 KB)
 - polar_angle_7 (Dataset [4397], 64-bit floating-point, 34.35 KB)
 - polar_angle_8 (Dataset [4397], 64-bit floating-point, 34.35 KB)
 - polar_angle_9 (Dataset [4397], 64-bit floating-point, 34.35 KB)
 - end_time (Dataset [1], String, length = variat)
 - experiment_identifier (Dataset [1], String, length = variat)
 - instrument (Group)
 - monitor (Group)
 - sample (Group)
 - start_time (Dataset [1], String, length = variat)

The right-hand pane shows a list of data channels and polar angles with their shapes:

Name	Shape
data/channel_1	[4397]
data/channel_10	[4397]
data/channel_2	[4397]
data/channel_3	[4397]
data/channel_4	[4397]
data/channel_5	[4397]
data/channel_6	[4397]
data/channel_8	[4397]
data/channel_9	[4397]
data/polar_angle_1	[4397]
data/polar_angle_10	[4397]
data/polar_angle_2	[4397]
data/polar_angle_3	[4397]
data/polar_angle_4	[4397]
data/polar_angle_5	[4397]
data/polar_angle_6	[4397]
data/polar_angle_7	[4397]
data/polar_angle_8	[4397]
data/polar_angle_9	[4397]
instrument/channel_1rd	[4397]
instrument/channel_1ip	[4397]
instrument/channel_1bl	[4397]
instrument/channel_1br	[4397]
instrument/channel_10l	[4397]
instrument/channel_10v	[4397]
instrument/channel_2rd	[4397]
instrument/channel_2ip	[4397]
instrument/channel_2bl	[4397]
instrument/channel_2br	[4397]
instrument/channel_3rd	[4397]
instrument/channel_3ip	[4397]

Currently we are **deploying** the software
at P02/P03 PETRA III beamlines



Storage Software

- PNI C++ libraries v 0.9.1
- Tango Data Server v 1.1.4
- Component Designer (Configuration Client Tool) v 1.2.1 under tests
- Configuration Server v 1.1.2
- Sardana Control Client v 1.0.2
- Command Line Tools v 1.0.0

Storage Software

- PNI C++ libraries v 0.9.1
- Tango Data Server v 1.1.4
- Component Designer (Configuration Client Tool) v 1.2.1 under tests
- Configuration Server v 1.1.2
- Sardana Control Client v 1.0.2
- Command Line Tools v 1.0.0

Thank You