The DDS User Manual v3.6

The DDS User Manual v3.6

Table of Contents

List of Tables

List of Examples

1. Introduction

1.1. The Dynamic Deployment System

The Dynamic Deployment System (DDS) - is a tool-set that automates and significantly simplifies a deployment of user defined processes (tasks) and their dependencies on any resource management system using a given topology.

In order to execute user tasks, DDS deploys agents. Each agent supports multiple tasks slots and therefore is able to run and watchdog multiple tasks simultaneously. Agent can be deployed using the dds-submimt command.

1.2. Basic concepts

DDS:

- implements a single-responsibility-principle command line tool-set and APIs,
- treats users' tasks as black boxes,
- doesn't depend on RMS (provides deployment via SSH, when no RMS is present),
- supports workers behind FireWalls (outgoing connection from WNs required),
- doesn't require pre-installation on WNs,
- deploys private facilities on demand with isolated sandboxes,
- provides a key-value properties propagation service for tasks,
- provides a simple custom command protocol, to help tasks to communicate between each other and with process outside of the topology,
- provides a rules based execution of tasks.

1.3. Features

2. Requirements

2.1. Server/UI

DDS UI/Server/WN run on Linux and Mac OS X.

General requirements:

- Incoming connection on dds-commander port (configurable)
- a C++11 compiler
- cmake 3.11.0 or higher
- BOOST 1.67 or higher (built by a C++11 compiler, with C++11 enabled)
- shell: BASH (or a compatible one)

Additional requirements for SSH plug-in:

- A public key access (or password less, via ssh-agend, for example) to destination worker nodes.

2.2. Workers

General requirements:

- Outgoing connection on dds-commander's port (configurable). This is required by dds-agent to be able to connect to DDS commander server
- shell: BASH (or a compatible one)

3. Download

3.1. Download location

Please, use DDS's Download page to get the latest version and all other versions of DDS.

3.2. DDS Version Number Scheme

DDS version has a form of MAJOR.MINOR(.PATCH), where:

- MAJOR - the major number is increased when there are significant jumps in functionality.
- MINOR - the minor number is incremented when only minor features or significant fixes have been added.
- PATCH - represents a number of commits (patches) to a current major.minor pair.

Note

The DDS's version scheme reflects the fact that DDS is both a production system and a research project. DDS uses odd minor version numbers to denote development releases and even minor version numbers to denote stable releases.

4. Installation

4.1. Step #1: Get the source

4.1.1. from DDS git repository

```
git clone https://github.com/FairRootGroup/DDS.git DDS-master
cd ./DDS-master
```

4.1.2. from DDS source tarball

Unpack DDS tarball:

```
tar -xzvf DDS-X.Y.Z-Source.tar.gz
```

Tar will created a new directory `./DDS-X.Y.Z-Source`, where *X.Y.Z* represents a version of DDS.

```
cd ./DDS-X.Y.Z-Source
```

4.2. Step #2: Configure the source

You can adjust some configuration settings in the `BuildSetup.cmake` bootstrap file. The following is a list of variables:

Table 4.1. DDS configuration variables

Variable	Description
CMAKE_INSTALL_PREFIX	Install path prefix, prepended onto install directories.(default $HOME/DDS/[DDS_Version])
CMAKE_BUILD_TYPE	Set cmake build type. Possible options are: None, Debug, Release, RelWithDebInfo, MinSizeRel (default Release)
BUILD_DOCUMENTATION	Build source code documentation. Possible options are: ON/OFF (default OFF)
BUILD_TESTS	Build DDS tests. Possible options are: ON/OFF (default OFF)

Now, prepare a build directory for an out-of-source build and configure the source:

```
mkdir build
cd build
cmake -C ../BuildSetup.cmake ..
```

Tip

If for some reason, for example a missing dependency, configuration failed. After you get the issue fixed, right before starting the **cmake** command it is recommended to delete everything in the build directory recursively. This will guaranty a clean build every time the source configuration is restarted.

4.3. Step #3: Build and install

Issue the following commands to build and install DDS:

```
make -j
make install
```

Installation Prefix

Please note, that by default DDS will be installed in `$HOME`/DDS/X.Y.Z, where X.Y.Z is a version of DDS. However users can change this behavior by setting the install prefix path in the bootstrap script `BuildSetup.cmake`. Just uncomment the setting of CMAKE_INSTALL_PREFIX variable and change dummy MY_PATH_HERE to a desired path.

WN package

Users have a possibility to additionally build DDS worker package for the local platform. In case if you have same OS types on all of the target machines and don't want to use WN packages from the DDS binary repository, just issue:

```
make -j wn_bin
make install
```

the commands will build and install a DDS worker package for the given platform.

We also recommend to build boost without icu library support. This will reduce the size of the WN package dramatically. The following is boost build options you can use to switch of icu:

```
./bootstrap.sh --without-icu ...
./b2 --disable-icu ...
```

4.4. Step #4: DDS Environment

In order to enable DDS's environment you need to source the `DDS_env.sh` script. Change to your newly installed DDS directory and issue:

```
cd [DDS INSTALL DIRECTORY]
source DDS_env.sh
```

You need to source this script every time before using DDS in a new system shell. Simplify it by sourcing the script in your bash profile.

5. Configuration

The default location of DDS's configuration file is `~/.DDS/DDS.cfg`. If missing, the configuration file will be automatically created once the DDS environment script is sourced.

DDS's configuration engine looks for the configuration file in the following order:

1. `$HOME/.DDS/DDS.cfg`
2. `$DDS_LOCATION/etc/DDS.cfg`
3. `$DDS_LOCATION/DDS.cfg`

Table 5.1. DDS configuration variables

Key	Description
server.work_dir	DDS commander will use this directory to create session files.
server.sandbox_dir	Some RMS, like LSF and slurm for example, require a shared files system to submit jobs. A shared folder (shared between the submit host and worker nodes). DDS will place RMS job script in this folder and will also use this folder as a sandbox for DDS workers.
server.log_dir	DDS commander will use this directory for logs.
server.log_severity_level	A global log severity level. Used by all DDS modules. **Log severity can be one of the following values:** • p_l - protocol low level events and higher, • p_m - protocol middle level events and higher, • p_h - protocol high level events and higher, • dbg - general debug events and higher, • inf - info events and higher, • wrn - warning events and higher, • err - error events and higher, • fat - fatal errors.
server.log_rotation_size	Log rotation size in MB. Once a log file reaches this number DDS will automatically create another log file.
server.log_has_console_output	0 or 1. If 1, then DDS console outputs will be also saved into the log.
server.commander_port_range_min and server.commander_port_range_max	A port range used by the commander.
server.idle_time	An idle time in seconds. DDS Commander and Agents respect this number and will automatically shutdown if inactive for this amount of seconds.

Key	Description
agent.work_dir	Use this key if you want to relocate working directories of DDS agents. By default they will use the directory specified by "server.sandbox_dir".

6. Quick Start

- Download DDS source tarball.
- Install DDS from source.

```
cd [DDS INSTALLATION]
source DDS_env.sh
dds-session start
dds-submit --rms localhost --slots 50
dds-info -n
dds-info -l
dds-topology --activate $DDS_LOCATION/tutorials/tutorial1/tutorial1_topo.xml
```

Enable DDS environment.
Start DDS commander server.
Deploy 1 DDS agent with 50 task slots on the localhost.
Use dds-info to find out a number of agents, which are online.
Use dds-info to check detailed information about agents.
Set and activate the topology.

7. Topology

The definition of the topology by the user has to be simple and powerful at the same time. Therefore a simple and powerful so called topology language has been developed.

The basic building block of the system is a *task*. Namely, a task is a user defined executable or a shell script, which will be deployed and executed by DDS on a given Resource Management System.

In order to describe dependencies between tasks in a topology we use *properties*. In run-time properties will be turned into simple key-value pairs. DDS uses its key-value propagation engine to make sure, that once property is set by one task, it will be propagated to other depended tasks. DDS treats values of properties as simple strings and doesn't do any special treatment/preprocessing on them. So, basically tasks can write anything into the values of properties (256 char max). Any of depended tasks can set properties. Anytime property is set it will be propagated to other depended tasks. (see for details TODO:"key-value propagation").

Tip

For example, if one task needs to connect with another task they can have the same property. A "server" task can store its TCP/IP port and host in the property. Once the property set, DDS will notice that and propagate it to other tasks.

Tasks can be grouped into *collections* and *groups*. Both collections and groups can be used to group several tasks. The main difference between collections and groups is that a collection requests from DDS to execute its tasks on the same physical machine, if resource allow that. This is useful if tasks suppose to communicate a lot or they want to access the same shared memory. A set of tasks and task collections can be also grouped into task groups. Another difference between groups and collection is that only groups can define multiplication factor for all its child elements.

Main group defines the entry point for task execution. Only main group can contain other groups.

7.1. Topology file

At the moment we use an XML based file to store topologies. XML is chosen because it can be validated against XSD schema. DDS's XSD schema file can be found in `$DDS_LOCATION/share/topology.xsd`.

```
<topology name="myTopology">
   [... Definition of tasks, properties, and collections ...]
   <main name="main">
      [... Definition of the topology itself, where also groups can be defined ...]
   </main>
</topology>
```

The file is basically divided on two parts: declaration and main part.

All properties, tasks and collections should be defined in the declaration part of the file. Users can define any number of topology entities in that block, even some, which are not going to be used in the main block.

In the main block the topology itself is defined. Groups and multiplication factors are also defined in main block.

7.2. Topology file example

Example 7.1. A topology file example

```
<topology name="myTopology">

   <var name="appNameVar" value="app1 -l -n --taskIndex %taskIndex% --collectionIndex %
```

```
  <var name="nofGroups" value="10" />

  <property name="property1" />
  <property name="property2" />

  <declrequirement name="requirement1" type="hostname" value="+.gsi.de"/>

 <decltrigger name="trigger1" condition="TaskCrashed" action="RestartTask" arg="5"/>

  <decltask name="task1">
     <requirements>
        <name>requirement1</name>
     </requirements>
     <exe reachable="true">${appNameVar}</exe>
     <env reachable="false">env1</env>
     <properties>
        <name access="read">property1</name>
        <name access="readwrite">property2</name>
     </properties>
     <triggers>
        <name>trigger1</name>
     </triggers>
  </decltask>
  <decltask name="task2">
     <exe>app2</exe>
     <properties>
        <name access="write">property1</name>
     </properties>
  </decltask>

  <declcollection name="collection1">
     <requirements>
        <name>requirement1</name>
     </requirements>
     <tasks>
        <name>task1</name>
        <name>task2</name>
        <name>task2</name>
     </tasks>
  </declcollection>

  <declcollection name="collection2">
     <tasks>
        <name>task1</name>
        <name>task1</name>
     </tasks>
  </declcollection>

  <main name="main">
     <task>task1</task>
     <collection>collection1</collection>
     <group name="group1" n="${nofGroups}">
        <task>task1</task>
        <collection>collection1</collection>
        <collection>collection2</collection>
     </group>
     <group name="group2" n="15">
        <collection>collection1</collection>
```

```
      </group>
   </main>

</topology>
```

DDS allows to define variables which later can be used inside the topology file. During the preprocessing all variable are replaced with their values. Variables are defined using the `var` tag which has two attributes `name` and `value`. Inside the file variable can be used as follows *${variable_name}*. In the above example we define two variables *${appNameVar}* and *${nofGroups}*.

When a particular task or collection is multiplied, sometimes it is necessary for the user to get the index of the task or collection instance. This can be done in two different ways. In the definition of the executable path one can use special tags *%taskIndex%* and *%collectionIndex%* to get the task and collection index respectivley. Before the task execution these tags are replaced with real values. The second possibility is to get task and collection index from environment. Two environment variables are defined for each task *$DDS_TASK_INDEX* and *$DDS_COLLECTION_INDEX*.

For each user task a set of environment variables is populated.

Populated environment variables

- *$DDS_TASK_PATH* - full path to the user task, for example, *main/group1/collection_12/task_3*
- *$DDS_GROUP_NAME* - ID of the parent group.
- *$DDS_COLLECTION_NAME* - ID of the parent collection if any.
- *$DDS_TASK_NAME* - ID of the task.
- *$DDS_TASK_INDEX* - index of the task.
- *$DDS_COLLECTION_INDEX* - index of the collection.
- *$DDS_SESSION_ID* - DDS session this task belongs to.

In the example above we define 2 properties - *property1* and *property2*. As you can see the `property` tag is used to define properties. *name* attribute is required and has to be unique for all properties.

Requirements is a way to tell DDS that a task or a collection has to be deployed on a particular computing node. As of now only host name or worker node name which is defined in the SSH configuration file are supported. Requirements are defined using `declrequirement` tag which has a number of attributes. All attributes are required. *name* attribute is an identifier and has to be unique for all requirements. *type* attribute is a type of the requirement. *value* attribute is a string value of the requirement. In order to define the pattern of the host name use either *hostname* or *wnname* values for the *type* attribute. `value` attribute for these requirement types can be either a full host name or a regular expression which matches the required host name. Use *hostname* if the requirement is defined based on the host name or *wnname* if the requirement is defined based on the SSH worker node name.

Task trigger defines a certain action which has to be performed whenever a specified condition is triggered. For example, if task crashed DDS will try to restart the task multiple times. For the moment only predefined conditions and actions are supported. Triggers are defined using `decltrigger` tag which has a number of attributes. All attributes are required. *name* attribute is an identifier and has to be unique for all triggers. *condition* attribute is a predefined condition. Has to be one of the following: *TaskCrashed*. *action* attribute is a predefined action. Has to be one of the following: *RestartTask*. *arg* is an argument for the action, for example, it can specify the number of attempts to restart the task.

In the next block we define tasks. For this the `decltask` tag is used. A task must also have the `name` attribute which is required and has to be unique for all declared tasks. The `requirements` element is optional and specifies the list of the already declared requirements for the task. The `triggers` element is optional and defines the list of task triggers. The `exe` element defines path to executable. The path can include program options, even options with quotes. DDS will automatically parse the path and extract program options in runtime. The `exe` tag has an optional attribute `reachable`, which defines whether executable is available on worker nodes. If it is not available, then DDS will take care of delivering it to an assigned worker in run-time.

In case when there is a script, that, for example sets environment, has to be executed prior to main executable one can specify it using the `env` element. The `env` tag also have `reachable` attribute.

If a task depends on some properties this can de specified using the `properties` tag together with a list of `name` elements which specify ID of already declared properties. Each property has an optional `access` attribute which defines whether user task will read (`read`), write (`write`) or both read and write (`readwrite`) a property. Default is `readwrite`.

Collections are declared using the `declcollection` tag. It contains a list of `task` tags with IDs which specified already declared tasks. Task has to be declared before it can be used in the collection. As for the task collection has an optional `requirements` element which is used to specify a list of the requirements for the collection. If the requirement defined for both task and collection than collection requirement has higher priority and is used for deployment.

The `main` tag declares the topology itself. In the example our main block consists of one task (*task1*), one collection (*collection1*) and two groups (*group1* and *group2*).

A group is declared using the `group` tag. It has a required attribute `name`, which is used to uniquely identify the group and optional attribute `n`, which defines multiplication factor for the group. In the example *group1* consists of one task (*task1*) and two collections (*collection1* and *collection2*). *group2* consists of one collection (*collection1*).

7.3. Topology XML tag reference

Table 7.1. Topology XML tags

Tag	Description

`topology`	*Parents:* No
	
	Children: `property`, `task`, `collection`, `main`
	
	Attributes: `name`
	
	Description:
	
	Declares a topology.
	
	`<topology name="myTopology">`
	`  [... Definition of tasks,`
	`  properties, collections and`
	`  groups ...]`
	`</topology>`

`var`	*Parents:* `topology`
	
	Children: No
	
	Attributes: `name`, `value`
	
	Description:
	
	Declares a variable which can be used inside the topology file as *${variable_name}*.
	
	`<var name="var1" value="value1"/>`
	`<var name="var2" value="value2"/>`

`property`	*Parents:* `topology` *Children:* No

Tag	Description

	Attributes: `name`
	
	Description:
	
	Declares a property.
	
	`<property name="property1"/>`
	`<property name="property2"/>`

`declrequire-ment`	*Parents:* `topology` *Children:* No *Attributes:* `name`, `type`, `value` *Description:* Declares a requirement for tasks and collections. `<declrequirement name="requirement1" type="hostname" value="+.gsi.de"/>`
`decltrigger`	*Parents:* `topology` *Children:* No *Attributes:* `name`, `condition`, `action`, `arg` *Description:* Declares a task trigger. `<decltrigger name="trigger1" condition="TaskCrashed" action="RestartTask`

`decltask`	*Parents:* `topology`
	
	Children: `exe`, `env`, `requirements`, `triggers`, `properties`
	
	Attributes: `name`
	
	Description:
	
	Declares a task.
	
	`<decltask name="task1">`
	`  <exe reachable="true">app1 -l -n</exe>`
	`  <env reachable="false">env1</env>`
	`  <requirements>`
	`  <name>requirement1</name>`
	`  </requirement>`
	`  <triggers>`
	`  <name>trigger1</name>`
	`  </triggers>`
	`  <properties>`
	`  <name access="read">property1</name>`

Tag	Description
	<name access="readwrite">property2</name> </properties> </decltask>
declcollection	*Parents:* topology *Children:* task *Attributes:* name *Description:* Declares a collection. <declcollection name="collection1"> <task>task1</task> <task>task1</task> </declcollection>
task	*Parents:* collection, group *Children:* No *Attributes:* No *Description:* Specifies the unique ID of the already defined task. <task>task1</task>
collection	*Parents:* group *Children:* No *Attributes:* No *Description:* Specifies the unique ID of the already defined collection. <collection>collection1</collection>
group	*Parents:* main *Children:* task, collection *Attributes:* name, n *Description:* Declares a group. <group name="group1" n="10"> <task>task1</task>

Tag	Description

	`<collection>collection1</collection>`
	`<collection>collection2</collection>`
	`</group>`

`main`	*Parents:* `topology`
	
	Children: `task`, `collection`, `group`
	
	Attributes: `name`
	
	Description:
	
	Declares a main group.
	
	`<main name="main">`
	`<task>task1</task>`
	`<collection>collection1</collection>`
	`<group name="group1" n="10">`
	`<task>task1</task>`
	`<collection>collection1</collection>`
	`<collection>collection2</collection>`
	`</group>`
	`</main>`

`exe` (required)	*Parents:* `decltask` *Children:* No *Attributes* `reachable` *Description:* Defines path to the executable or script for the task. `<exe reachable="true">app1 -l -n</exe>`
`env` (optional)	*Parents:* `decltask` *Children:* No *Attributes:* `reachable` *Description:* Defines the path to script that has to be executed prior to main executable. `<env reachable="false">setEnv.sh</env>`
`requirements` (optional)	*Parents:* `decltask`, `declcollection` *Children:* `name` *Attributes:* No *Description*

Tag	Description

	Defines a list of requirements.
	
	`<requirements>`
	`  <name>requirement1</name>`
	`  <name>requirement2</name>`
	`</requirements>`

`properties` (optional)	*Parents:* `decltask`
	
	Children: `name`
	
	Attributes: No
	
	Description
	
	Defines a list of dependent properties.
	
	`<properties>`
	`  <name>property1</name>`
	`  <name>property2</name>`
	`</properties>`

`name` (required)	*Parents:* `properties` *Children:* No *Attributes* access *Description* Defines an ID of the already declared property. `<name>property1</name>`

Table 7.2. Topology XML attributes

Attribute	Description
`name`	*Use:* required *Default:* No *Tags:* `topology`, `property`, `declrequirement`, `decltask`, `declcollection`, `group`, `main` *Restrictions:* String with minimum length of 1 character. *Description:* Defines identificator (ID) for topology, property, requirement, task, collection and group. ID has to be unique within its scope, i.e. ID for tasks has to be unique only for tasks.

Attribute	Description
	`<topology name="myTopology">`

`reachable`	*Use:* optional
	Default: true
	Tags: `exe`, `env`
	Restrictions: true|false
	Description:
	Defines if executable or script is available on the worker node.
	`<exe reachable="true">app -l</exe>`
	`<env>env1</env>`

`n`	*Use:* optional
	Default: 1
	Tags: `group`
	Restrictions: unsigned integer 32-bit which is more or equal to 1
	Description:
	Defines multiplication factor for group.
	`<exe reachable="true">app -l</exe>`
	`<env>env1</env>`

`access`	*Use:* optional *Default:* readwrite *Tags:* `name` *Restrictions:* read|write|readwrite *Description:* Defines access type from user task to properties. `<name access="read">property1</name>`
`type`	*Use:* required *Tags:* `declrequirement` *Restrictions:* hostname|wnname *Description:* Defines the type of the requirement.

Attribute	Description
	`<declrequirement name="requirement1" type="wnname" value="wn2"/>`
`condition`	*Use:* required *Tags:* `decltrigger` *Restrictions:* TaskCrashed *Description:* Defines trigger condition. `<decltrigger name="trigger1" condition="TaskCrashed" action="RestartTask" a`
`action`	*Use:* required *Tags:* `decltrigger` *Restrictions:* RestartTask *Description:* Defines trigger action. `<decltrigger name="trigger1" condition="TaskCrashed" action="RestartTask" a`

8. How to Start

8.1. Environment

In order to enable DDS environment you need to source the `DDS_env.sh` script. The script is located in the directory where you installed PoD.

```
cd [DDS INSTALLATION]
source DDS_env.sh
```

8.2. Server

Use the dds-session command to *`start`*/*`stop`*/*`list`* DDS sessions.

```
dds-session start
```

8.3. Deploy Agents

In order to deploy agents you can use different DDS plug-ins.

8.3.1. Deploy-Agents using: SSH plug-in

DDS's SSH plug-in is the best and the fastest way to deploy DDS agents. When you don't have an RMS or you want to use a Cloud based system or even if you want just to use resources around you, like computers of your colleagues, then the plug-in is the best way to go.

First of all you need to define resources.

Then use dds-submit to deploy DDS agents on the given resources:

```
dds-submit --rms ssh -c FULL_PATH_TO_YOUR_SSHPLUGIN_RESOURCE_FILE
```

8.4. Check availability of Agents

Using dds-info you can query different kinds of information from DDS. For example you can check how many agents are already online:

```
dds-info -n
```

or query more detailed info about agents:

```
dds-info -l
```

8.5. Activate Topology

Once you get enough online agents, you can activate them. Activation of agents means, that DDS will use the given topology to distribute user tasks across available resources (agents):

```
dds-topology --activate FULL_PATH_TO_YOUR_TOPOLOGY_FILE
```

DDS will automatically check whether available resources are actually sufficient to execute the given topology.

9. How to Test

xxxx

9.1. First Section

xxxx

10. Tutorials

10.1. Tutorial 1

This tutorial demonstrates how to deploy a simple topology of 2 types of tasks (TaskTypeOne and TaskTypeTwo). By default, there will be deployed one instance of TaskTypeTwo and 5 instances of TaskTypeOne. Additionally TaskTypeTwo subscribes on key-value property from TaskTypeOne, which name is TaskIndexProperty. Once TaskTypeTwo receives values of TaskIndexProperty from all TaskTypeOne, it will set the ReplyProperty property. Number of instances can be changed in the topology file (`tutorial1_topo.xml`) using --instances option of TaskTypeOne. Please note that number of worker nodes in the SSH-plugin configuration file (`tutorial1_hosts.cfg`) has to be changed accordingly.

After DDS is installed the tutorial can be found in `$DDS_LOCATION/tutorials/tutorial1`

The source code of tasks is located in `"DDS_SRC_DIR"/dds-tutorials/dds-tutorial1`

Files of the tutorial

- task-type-one: executable of the task TaskTypeOne
- task-type-two: executable of the task TaskTypeTwo
- tutorial1_topo.xml: a topology file
- tutorial1_hosts.cfg: a configuration file for DDS SSH plug-in

10.1.1. Usage

Before running the tutorial make sure that: 1) Default working directory `~/tmp/dds_wn_test` must exist before running the tutorial. The directory can be changed in `tutorial1_hosts.cfg`. 2) SSH passwordless access to the localhost is required.

```
cd $DDS_LOCATION/tutorials/tutorial1
dds-session start --local
dds-submit -r ssh -c tutorial1_hosts.cfg
dds-topology --activate tutorial1_topo.xml
```

10.1.2. Result

To check the result, change to `~/tmp/dds_wn_test`. If the default setup was used, then there will be WN directories located: wn, wn_1, wn_2, wn_3, wn_4, wn_5.

DDS catches output of tasks and saves it in log files under names [task_name]_[date_time]_out|err.log. For example: `TaskTypeOne_2015-07-16-11-44-42_6255430612052815609_out.log`

10.2. Tutorial 2

This tutorial demonstrates how to use DDS custom commands for user task and for utility.

After DDS is installed the tutorial can be found in `$DDS_LOCATION/tutorials/tutorial2`

The source code of tasks is located in `"DDS_SRC_DIR"/dds-tutorials/dds-tutorial2`

Files of the tutorial

- task-custom-cmd: user task which receives and send DDS custom commands

- ui-custom-cmd: utility which connects to DDS commander and send custom commands to user tasks
- tutorial2_topo.xml: a topology file
- tutorial2_hosts.cfg: a configuration file for DDS SSH plug-in

10.2.1. Usage

Before running the tutorial make sure that: 1) Default working directory `~/tmp/dds_wn_test` must exist before running the tutorial. The directory can be changed in `tutorial1_hosts.cfg`. 2) SSH passwordless access to the localhost is required.

```
cd $DDS_LOCATION/tutorials/tutorial2
dds-session start --local
dds-submit -r ssh -c tutorial2_hosts.cfg
dds-topology --activate tutorial2_topo.xml
ui-custom-command
```

10.2.2. Result

To check the result, change to `~/tmp/dds_wn_test`. If the default setup was used, then there will be WN directories located: wn, wn_1, wn_2, wn_3, wn_4, wn_5.

DDS catches output of tasks and saves it in log files under names [task_name]_[date_time]_out|err.log. For example: `TaskTypeOne_2015-07-16-11-44-42_6255430612052815609_out.log`

After executing **ui-custom-command** there will be an output to the console with receiving and sending custom commands. Also check output files of tasks.

11. Command-line interface

Name

dds-session — start/stop DDS commander and manage DDS sessions
UNIX/Linux/OSX

Synopsis

`dds-session` {[[start *--mixed*] | [stop *SESSION_ID*] | [stop_all] | [list *all* | *run*]] | [set-default *SESSION_ID*] | [clean *-f*]}

Description

Using this command users can perform a set of operations on DDS sessions, such as *start*/*stop* DDS server by creating new and stopping existing sessions. Users can also *list* available sessions or *clean* expired ones.

One user can start multiple DDS sessions. Each session will have its own DDS commander instance and will be sandboxed, i.e. won't disturb other sessions of the same user.

Options

`start`
: Start a new DDS session. DDS will automatically set the newly created session as a default one.

  A single user can start as many DDS sessions as desired. Users are limited only by the resources of underlying system.

  Each DDS session spawns its own commander server. All sessions are completely isolated from each other.

  At the server start DDS will test availability of DDS WN bin. packages and download them from the DDS repository if they are missing. If the user provides *--mixed* parmeter, then WN packages for all systems (Linux, OS X) will be checked. By default DDS checks only for a package compatible with the local system only.

  To build a binary package for the local system, just issue:

```
make -j wn_bin
make -j install
```

`stop`
: Stop a given DDS session specified by *SESSION_ID*. If no *SESSION_ID* argument is provided, the command will stop the default DDS session. But in this case the command will ask user to confirm the choice.

`stop_all`
: Stop all running DDS sessions.

`list`
: List available DDS sessions. User must provide the filter criteria, either *all* or *run*

  With *all* the command will list absolutly all existing sessions, including expired ones.

  With *run* the command will list only running DDS sessions.

`set-default`
: Sets a given *SESSION_ID* as a default session ID.

  The default session ID is used by all DDS commands, when user doesn't provide a session ID explicitly in the command line arguments.

`clean`
: The command cleans DDS sessions. It will remove all session related temporary files and logs. Be careful using this command. The operation can't be undone.

  For safety reason the command confirms with the user removal of each DDS session, but you can avoid this by providing the *`-f`* argument.

Example 11.1. A dds-session command usage

```
$ dds-session start

DDS session ID: cf84e72d-a3af-4fd8-af73-4337e9434612
Checking precompiled binaries for the local system only:
 dds-wrk-bin-2.1.12.g7619ef0-Darwin-universal.tar.gz - OK
Starting DDS commander...
Waiting for DDS Commander to appear online...
DDS commander appears online. Testing connection...
DDS commander is up and running.
------------------------
DDS commander server: 60753
------------------------
Startup time: 1061.46 ms
Default DDS session is set to cf84e72d-a3af-4fd8-af73-4337e9434612
Currently running DDS sessions:
cf84e72d-a3af-4fd8-af73-4337e9434612   [2018-08-22T11:53:34Z]   RUNNING

$ dds-session list all

   cfc8e86d-157b-404e-bde8-a32f8b3c1331   [2018-08-21T13:49:35Z]   STOPPED
   5fdc6142-497c-433c-8333-721f05eabe31   [2018-08-21T14:10:39Z]   STOPPED
 * cf84e72d-a3af-4fd8-af73-4337e9434612   [2018-08-22T11:53:34Z]   RUNNING



$ dds-session stop cf84e72d-a3af-4fd8-af73-4337e9434612

Stopping DDS commander: cf84e72d-a3af-4fd8-af73-4337e9434612
Sending a graceful stop signal to Commander (pid/sessionID): 60753/cf84e72d-a3af-4fd8-a
dds-commander: self exiting (60753)...
```

Name

dds-commander — manages DDS facility
UNIX/Linux/OSX

Synopsis

`dds-commander` [[`-h, --help`] | [`-v, --version`]] {[start] | [stop]}

Description

Warning

The command must not be used directly. Please use the dds-session command instead.

Name

dds-user-defaults — get and set global DDS options
UNIX/Linux/OSX

Synopsis

`dds-user-defaults` [`-h, --help`]|[`-v, --version`]|[`-V, --verbose`]|[`-p, --path`]|[`-d, --default`]][`-c, --config` *arg*][`-s, --session` *arg*][`--ignore-default-sid`][`--default-session-id`][`--default-session-id-file`][`-f, --force`][[`--key` *arg*]|[`--wrkpkg`]|[`--wrkscript`]|[`--rms-sandbox-dir`]|[`--user-env-script`]|[`--server-info-file`]]

Description

The **dds-user-defaults** command can be used to get and set global DDS options. It also can be used to get different static settings, related to the current deployment.

Options

`-h, --help`
: Shows usage options.

`-v, --version`
: Shows version information.

`-V, --verbose`
: Causes the command to verbose additional information and error messages.

`-p, --path`
: Shows default DDS user defaults config file path.

`-d, --default`
: Generates a default DDS configuration file.

`-f, --force`
: If the destination file exists, removes it and creates a new file, without prompting for confirmation. Can only be used with the `-d, --default` options.

`-c, --config` *arg*
: This options can be used together with other options to specify non-default location of the DDS configuration file. By default the command uses `~/.DDS/DDS.cfg`.

`-s, --session` *arg*
: Use the specified DDS Session ID instead of a default one.

`--ignore-default-sid`
: Force to ignore a default sid.

`--default-session-id`
: Show the current default session ID.

`--default-session-id-file`
: Show the full path of the default session ID file.

`--key` *arg*
: Gets a value for the given key from the DDS user defaults.

`--wrkpkg`
: Shows the full path of the worker package. The path must be evaluated before use.

`--wrkscript`
: Shows the full path of the worker script. The path must be evaluated before use.

`--rms-sandbox-dir`
: Shows the full path of the RMS sandbox directory. It returns server.sandbox_dir if it is not empty, otherwise server.work_dir is returned. The path must be evaluated before use.

`--user-env-script`
: Shows the full path of user's environment script for workers (if present). The path must be evaluated before use.

`--server-info-file`
: Shows the full path of the DDS server info file. The path must be evaluated before use.

Name

dds-submimt — submits and activates DDS agents
UNIX/Linux/OSX

Synopsis

`dds-submit` [`-h, --help`]|[`-v, --version`]][`-l, --list`][`-r, --rms` *arg*][`-s, --session` *arg*] {[`-c, --config` *arg*]|[`-n, --number` *arg*]|[`-s, --slots` *arg*]}

Description

The command is used to submit DDS agents to allocate resources for user tasks. Once enough agents are online use the dds-topology command to activate the agents - i.e. distribute user tasks across agents and start them.

Options

`-h, --help`
Shows usage options.

`-v, --version`
Shows version information.

`--l, --list` *arg*
List all available RMS plug-ins.

`--r, --rms` *arg*
Defines a destination resource management system plug-in. Use "--list" to find out names of available RMS plug-ins.

`--s, --session` *arg*
DDS Session ID.

`--path` *arg*
Defines a path to the root plug-ins directory. If not specified than default root plug-ins directory is used.

`-c, --config` *arg*
A plug-in's configuration file. It can be used to provide additional RMS options.

`-n, --number` *arg*
Defines a number of agents to spawn. This option can not be mixed with "--config".

`-s, --slots` *arg*
Defines a number of task slots per agent. This option can not be mixed with "--config".

Name

dds-info — can be used to query different kinds of information from DDS commander server
UNIX/Linux/OSX

Synopsis

`dds-info` [[`-h, --help`] | [`-v, --version`]] [[`-s, --session` *`arg`*] | [`--commander-pid`] | [`--status`] | [`-n, --active-count`] | [`-l, --agents-list`] | [`--idle-count`] | [`--executing-count`] | [`--wait-count` *`arg`*] | [`--active-topology`]]

Description

The command can be used to query different kinds of information from DDS commander server.

Options

`-h, --help`
: Shows usage options.

`-v, --version`
: Shows version information.

`-s, --session` *`arg`*
: DDS Session ID.

`--commander-pid`
: Return the pid of the commander server.

`--status`
: Query current status of DDS commander server.

`-n, --active-count`
: Returns a number of online slots.

`-l, --agents-list`
: Show detailed info about all online agents.

`--idle-count`
: Returns a number of idle slots.

`--executing-count`
: Returns a number of executing slots.

`--wait-count` *`arg`*
: The command will block infinitely until a required number of agents are available. Must be used together with `--active-count`, `--idle-count` or `--executing-count`

`--active-topology`
: Returns the name of the active topology.

Name

dds-test — a DDS self-test utility
UNIX/Linux/OSX

Synopsis

`dds-test` [[`-h, --help`]|[`-v, --version`]][`-s, --session` *`arg`*][`--verbose`] {[-t, --transport]}

Description

This tool runs stress tests of DDS system.

Options

`-h, --help`
: Shows usage options.

`-v, --version`
: Shows version information.

`--verbose`
: Causes the command to verbose additional information and error messages.

`-s, --session` *`arg`*
: DDS Session ID.

`-t, --transport`
: Performs transport test.

Name

dds-topology — topology related commands
UNIX/Linux/OSX

Synopsis

`dds-topology` [[`-h, --help`] | [`-v, --version`] | [`-V, --verbose`] [[`--disable-validation`]] | [`-s, --session` *`arg`*] | [`--activate` *`arg`*] | [`--stop`] | [`--update` *`arg`*] | [`--validate` *`arg`*] | [`--topology-name` *`arg`*]]

Description

This command allows to perform topology related tasks.

Options

`-h, --help`
: Shows usage options.

`-v, --version`
: Shows version information.

`-V, --verbose`
: Causes the command to verbose additional information and error messages.

`--disable-validation`
: Switches off topology validation.

`--s, --session` *`arg`*
: DDS Session ID.

`--activate` *`arg`*
: Requests DDS to activate agents, i.e. distribute and start user tasks accoring to the given topology.

`--update` *`arg`*
: Requests DDS to update currently running topology with a new one.

`--stop`
: Requests DDS to stop execution of user tasks. Stop the active topology.

`--validate` *`arg`*
: Validates topology file against DDS's XSD schema.

`--topology-name` *`arg`*
: Get the name of the topology for a given topology file.

Name

dds-agent-cmd — send commands to agent
UNIX/Linux/OSX

Synopsis

`dds-agent-cmd` [[`-h, --help`] | [`-v, --version`] | [`command, --command` *`arg`*] | [`-s, --session` *`arg`*]] {[`getlog` *`arg`*] {[`-a, --all`]} | [`update-key` *`arg`*] {[`--key` *`arg`*] | [`--value` *`arg`*]}}

Description

This utility allows to send commands to DDS agents. Currently available commands are: getlog, update-key.

Options

`getlog` *`arg`*
: Download all log files from active agents. All files from agents' working directories with the extension "log" will be tar/zip'ed into a single file and downloaded on DDS commander server machine into the directory specified by server.log_dir DDS configuration option and placed in the subdirectory "agents" (default: `~/.DDS/log/agents`)

  Usage example:

```
dds-agent-cmd getlog -a
```

`update-key` *`arg`*
: It forces an update of a given task's property in the topology. Name of the property and a new value should be provided additionally (see `--key` and `--value`)

  Usage example:

```
dds-agent-cmd update-key --key mykey --value new_value
```

`--key`
: Defines the key to update

`--value`
: Defines a new value of the given key.

`-a, --all`
: Send command to all activer agents.

`--s, --session` *`arg`*
: DDS Session ID.

12. RMS plug-ins

12.1. For Developers

12.1.1. Basic concept

DDS offers a possibility for external developers to make their own RMS plug-ins.

Conceptually, each RMS plug-in is just an executable, which uses a simple DDS plug-in API and is able to deploy and execute a DDS worker package on a corresponding RMS.

The following is a basic workflow:

- User requests to deploy DDS agents or a given RMS using the dds-submit --rms XXXX command. Where XXXX is the name of the plug-in user wants to use.
- DDS commander server receives the request, looks for a suitable plug-in (associated with the XXXX name) and starts it. Plug-in has 2 minutes to connect back to commander to receive exact details about the submit request.
- Once plug-in is started it should contact with the DDS commander server using DDS API, receive details and deploy agents on a given RMS. That's so far it.

12.1.2. Requirements

- DDS requires each plug-in to have the name according to the following format: `dds-submit-`*`XXXX`*, where XXXX is the name of the plug-in (or name of RMS it wraps). All lower case characters.
- A DDS plug-in (executable) and all related files must be sandboxed in a dedicated folder: `path/dds-submit-XXXX/`. The folder path is provided as a commandline argument for all plug-ins. The default location of plug-ins is `$DDS_LOCATION/plugins/dds-submit-XXX/`.
- A DDS plug-in should take two command line arguments

  [--id *`arg`*]
  and

  [--path *`arg`*]
  DDS will call the plug-in with this command line arguments and will provide a unique ID and a plug-in directory path. ID must be used when ever plug-in communicates with DDS commander server (see "plug-in-id" in the API section for more info). Plug-in's directory path can be used to access related files if needed.
- Plug-ins are responsible to remove all own temporary files on exit. DDS doesn't take ownership of any file create by plug-ins.

12.1.3. API

The dds::intercom_api::CRMSPluginProtocol is a wrapper class for plug-in/"DDD commander server" communication.

Once started and ready the plug-in should subscribe on the "submit and "message" command from the DDS commander server.

```
CRMSPluginProtocol prot("plug-in-id");

prot.onSubmit([](const SSubmit& _submit) {
 // Implement submit related functionality here.
```

```
 // After submit has completed call stop() function.
 prot.stop();
});


prot.onMessage([](const SMessage& _message) {
 // Message from commander received.
 // Implement related functionality here.
});
```

onSubmit will deliver to the plugin-in the actual request dds::intercom_api::SSubmit. It can contain either a configuration file (format of the file is plug-in depended) or simply a number of agents to deploy. But it will always contain the path to the worker package, which plug-in is supposed to deploy on RMS and execute. Additionally developers can use a DDS command line tools to find out the location of the worker package: dds-user-defaults --wrkscript. This is especially useful when plug-ins use shell scripts.

Once ready the plug-in let's give a hit to DDS commander that we are online and ready for a job:

```
// Let DDS commander know that we are online and start listening for notifications.
prot.start();
```

After that commander will form a submit request and will send it back to the plug-in. By default his call will block the main thread until one of the condition is true:

- 10 minutes timeout,
- Failed connection to DDS commander or disconnection from DDS commander,
- Explicit call of the stop() function

If you do not want to stop the thread use:

```
// "false" means that we do not block the thread
prot.start(false);
```

If there are no subscribers the thread is not blocked in any case.

Once connected you can use `proto.sendMessage` to send messages. Those messages will be displayed to user while he/she waits on dds-submit command. Be advised, that once commander receives the error message it will forward it to the user and close connection as it means a failed submission.

We strongly recommend to protect CRMSPluginProtocol calls in a try/catch block, as all methods can throw std::exceptions:

```
try {
 CRMSPluginProtocol prot("plug-in-id");

 prot.onSubmit([](const SSubmit& _submit) {
  // Implement submit related functionality here.

  // report something back to a user
  proto.sendMessage(dds::intercom_api::EMsgSeverity::info, "Text of the info message");

  // After submit has completed call stop() function.
  prot.stop();
```

```
});


prot.onMessage([](const SMessage& _message) {
 // Message from commander received.
 // Implement related functionality here.
});

// Let DDS commander know that we are online and start listening for notifications
prot.start();
} catch (exception& _e) {
 // Report error to DDS commander
 proto.sendMessage(dds::intercom_api::EMsgSeverity::error, e.what());
}
```

12.2. SSH

12.2.1. Resource definition

DDS's SSH plug-in is capable to deploy DDS agents on any resource machine available for password-less access (public key, ssh agent, etc.) To define resources for the SSH plug-in we use a comma-separated values (CSV) configuration file, in case if you want to deploy agents on several computing nodes. The ssh plug-in can also spawn agents on the local machine only. In this case you don't need a configuration file - just use dds-submit -n X, where X is a desired number of agents to spawn. Fields are normally separated by commas. If you want to put a comma in a field, you need to put quotes around it. Also 3 escape sequences are supported.

Table 12.1. DDS's SSH plug-in configuration fields

1	2	3	4	5
id (must be any unique string). This id string is used just to distinguish different DDS workers in the plug-in.	a host name with or without a login, in a form: login@host.fqdn	additional SSH params (could be empty)	a remote working directory	a number of agents to spawn

Example 12.1. An example of an SSH plug-in configuration file

```
r1, anar@lxg0527.gsi.de, -p24, /tmp/test, 10
# this is a comment
r2, user@lxi001.gsi.de,,/home/user/dds,10
125, user2@host, , /tmp/test,
```

12.2.2. Example usage

Call using a given configuration file:

```
dds-submit -r ssh -c your-ssh-Resource-definition-config-file
```

Call using a local system only to spawn 10 DDS agents on it:

```
dds-submit -r ssh -n 10
```

12.3. Localhost

12.3.1. Introduction

DDS's localhost plug-in is capable to deploy DDS agents on a local machine. Unlike SSH plug-in, localhost plug-in doesn't require a password-less access (public key, ssh agent, etc.). The configuration file is not required for localhost plug-in. The plug-in spawns 1 agent with a defined number of task slots on the local machine only. Just use dds-submit --slots X, where X is a desired number of task slots.

12.3.2. Example usage

Call using a local system only to spawn 1 DDS agent with 10 task slots:

```
dds-submit -r localhost --slots 10
```

12.4. SLURM

12.4.1. Sandbox directory

If your home directory is not shared on the SLURM cluster, then you must define a sandbox directory, which DDS will use to store SLURM job script and all jobs' working directories will be also located there. Please note, that at the moment DDS doesn't clean jobs' working directories, therefore you are responsible to remove them if needed.

In order to set sandbox directory a DDS global option "server.sandbox_dir" have to be changed, which is located in the DDS configuration file DDS.cfg (default location: `$HOME/.DDS/DDS.cfg`)

12.4.2. User configuration

Using dds-submit -c My_SLURM.cfg command you can provide additional configuration options for DDS SLURM jobs. For example, the following command will submit 10 DDS agents (each with 50 task slots) and will use additional SLURM configuration options provided in the `My_SLURM.cfg`:

```
dds-submit -r slurm -n 10 --slots 50 -c My_SLURM.cfg
```

Caution

The content of the custom SLURM job configuration file can be any sbatch parameter, except "srun" and "--array".

For example, `My_SLURM.cfg` can contain:

```
#SBATCH -A "account"
#SBATCH --time=00:30:00
```

12.4.3. Usage example

Submit 10 DDS agents to SLURM cluster. On the SLURM submitter machine execute:

```
dds-submit -r slurm -n 10
```

```
dds-submit: Contacting DDS commander on lxbk0200.gsi.de:20001 ...
dds-submit: Connection established.
dds-submit: Requesting server to process job submission...
dds-submit: Server reports: Creating new worker package...
dds-submit: Server reports: RMS plug-in: /u/manafov/DDS/1.1.61.g474ddc6/plugins/dds-su
dds-submit: Server reports: Initializing RMS plug-in...
dds-submit: Server reports: RMS plug-in is online. Startup time: 17ms.
dds-submit: Server reports: Plug-in: Generating SLURM Job script...
dds-submit: Server reports: Plug-in: Preparing job submission...
dds-submit: Server reports: Plug-in: pipe log engine: Submitting DDS Job on the SLURM

dds-submit: Server reports: Plug-in: pipe log engine: SLURM: Submitted batch job 95399

dds-submit: Server reports: Plug-in: DDS agents have been submitted
```

Check the status of your SLURM jobs:

```
scontrol show job 9539993
```

Check the status of your DDS agents:

```
dds-info -ln
```

Once agents are online, use DDS as normal.