EPJ Web of Conferences **214**, 01011 (2019) https://doi.org/10.1051/epjconf/201921401011
CHEP 2018

DDS: The Dynamic Deployment System

Andrey Lebedev[1,*] and *Anar* Manafov[1,**]

[1]GSI Helmholtzzentrum für Schwerionenforschung GmbH, Darmstadt, Germany

Abstract. The Dynamic Deployment System (DDS) [1, 2] is a tool-set that automates and significantly simplifies the deployment of user-defined processes and their dependencies on any resource management system (RMS) using a given topology. DDS is a part of the ALFA framework [3].

DDS implements a single responsibility principle command line tool-set and API. The system treats users' task as a black box – it can be an executable or a script. It also provides a watchdogging and a rule-based execution of tasks. DDS implements a plug-in system to abstract the execution of the topology from RMS. Additionally it ships an SSH and a localhost plug-ins which can be used when no RMS is available. DDS doesn't require pre-installation and pre-configuration on the worker nodes. It deploys private facilities on demand with isolated sandboxes. The system provides a key-value property propagation engine. That engine can be used to configure tasks at runtime. DDS also provides a lightweight API for tasks to exchange messages, so-called, custom commands.

In this report a detailed description, current status and future development plans of the DDS will be highlighted.

1 Introduction

The new ALICE-FAIR concurrency framework for high quality parallel data processing and reconstruction on heterogeneous computing systems is called ALFA. It provides a data transport layer and the capability to coordinate multiple data processing components. ALFA is a flexible, elastic system which balances reliability and ease of development with performance by using multi-processing with message passing. With multi-processing, each process assumes limited communication and reliance on other processes. Such applications are much easier to scale horizontally (for example when a new process instance has to be started or a new processing node has to be added) to meet the computing and throughput demands of the experiments. However, the communication between the different processes has to be controlled and orchestrated, for example a reconstruction task is presented now by few processes that consist of a processing graph (topology) which could run on one or more nodes. To manage such an environment, we created the Dynamic Deployment System (DDS). DDS is an independent set of utilities and interfaces, providing dynamic distribution of different user processes for any given topology on any Resource Management System (RMS) or a laptop.

*e-mail: andrey.lebedev@gsi.de
**e-mail: a.manafov@gsi.de

2 Topology

A key point of DDS is the so-called topology language. DDS is a user-oriented system, i.e. the definition of topologies is simple and powerful at the same time. A 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 RMS. To describe dependencies between tasks in a topology we use properties. A property is represented as a key-value pair, where a value can be any string. DDS implements an efficient engine for property synchronization. We call it a key-value propagation feature.

Tasks can be grouped to DDS collections and DDS groups. The difference is that collections signal to the DDS topology parser that tasks of collection should be executed on the same physical machine. This is useful for cases when tasks have lots of communication or they want to access the same shared memory or any other shared resource. On the other hand groups define a multiplication factor. A set of tasks and task collections can be grouped together to form task groups. Task groups basically define how many instances of a task or a collection to be executed simultaneously.

A sketch of the topology definition is shown on Figure 1. The detailed description of the topology is available in the DDS documentation [1].

```
<topology id="myTopology">
   [... Definition of tasks, properties, and collections ...]
   <main name="main">
      [… Definition of the topology itself, including groups...]
   </main>
</topology>
```

Figure 1. A sketch of the DDS topology file. The topology file is divided into two parts. In the upper part user declares tasks, properties, collections etc. In the lower part the main block defines which tasks has to be deployed to RMS.

The DDS topology is not fixed at runtime, users always have a possibility to update it at runtime. This feature offers a possibility to make changes in a currently running topology without stopping it. The algorithm determines the difference between old and new topologies on the fly. As a result, it creates lists of tasks and collections that have to be added and/or removed. These lists are passed to DDS scheduler which then applies required changes by sending commands to corresponding DDS agents.

3 Highlights of features

In this section we shall cover the most important highlights and features of DDS, which are:

- property (key-value) propagation;
- messaging (custom commands) for user tasks and external utils;
- RMS plug-ins.

3.1 Property propagation

The feature allows user's tasks to exchange and synchronize the configuration (key-value) dynamically at runtime. For example, startup synchronization of the multiprocessing reconstruction requires tasks to exchange theirs connection strings so that they are able to connect to each other.

DDS key-value engine is highly optimized for massive key-value transport and has a decentralized architecture. Internally small packets of messages are automatically accumulated and transported as a single message. That gives a huge performance boost in case of a massive key-value exchange. DDS agents use shared memory for local transport and caching of key-value properties.

From the user perspective, the properties are declared in the topology file with a certain identifier. Tasks define a list of dependent properties with different access modifiers. When a property is received a subscribed task will be notified about the property update via the DDS API.

3.2 Custom commands

Custom commands are an easy way to communicate directly with user tasks. Also processes outside the topology can communicate with tasks inside the topology. One example is an external high level control system which coordinates tasks (e.g. experiment control). Two use cases were considered:

1. Tasks defined by the topology connect to the DDS agent, which can establish through this connection a direct communication between tasks under his supervision.

2. External utility connects to the DDS commander allowing to control user tasks from an external process.

A custom command is a standard part of the DDS protocol exposed via DDS API. From the user perspective a custom command can be any text, for example, one can build a JSON or XML based protocol on top of it. A custom command recipient is defined by a condition allowing to send a command to a particular task or broadcast the command to a group of tasks or to all tasks.

3.3 RMS plug-ins

DDS can use different job submission front-ends for deployment. In order to cover various RMS, a plug-in architecture has been implemented in DDS which gives external developers a possibility to create RMS plug-ins. The architecture guarantees isolated and safe execution of the plug-in. A plug-in is a standalone process - failures and crashes will not affect the DDS. Internally it uses the DDS protocol for communication between the plug-in and the DDS commander server - plug-ins within the DDS implement the same transport protocol for the exchange of messages and data. The RMS plug-in architecture is shown on Figure 2.

The following RMS plug-ins are currently implemented in DDS: localhost, SSH, Slurm [4], PBS [5], LSF [6], and Apache Mesos [7].

4 Implementation details

In this section, we shall give some implementation details of various DDS components.

4.1 DDS workflow

DDS command line interface is simple and user friendly. A quick start can look like the following:

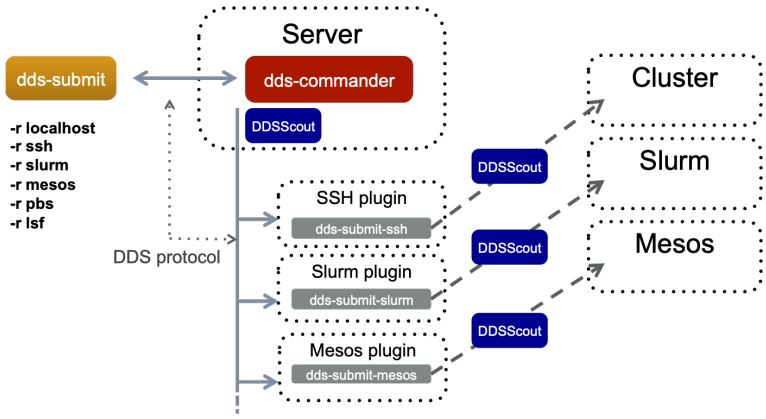

Figure 2. RMS plug-in architecture. 1. dds-commander starts a plug-in based on the dds-submit parameter. 2. plug-in contacts DDS commander server asking for submissions details. 3. plug-in deploys DDS scout fat script on target machines. 4. plug-in executes DDS scout on target machines.

```
$ dds-session start
$ dds-submit -r ssh -c hosts.cfg
$ dds-topology --activate topology.xml
$ dds-topology --update new-topology.xml
```

The first command starts a DDS session with a commander server. The second one submits DDS scouts which spawn DDS agents for the current session and act as a watchdog for them. In the given example we use SSH plug-in together with the configuration file. After the agents are submitted we are ready to activate the topology. The third command does that. If required the running topology can be updated with the new one without stopping using the fourth command.

A sketch of the DDS workflow is presented in Figure 3.

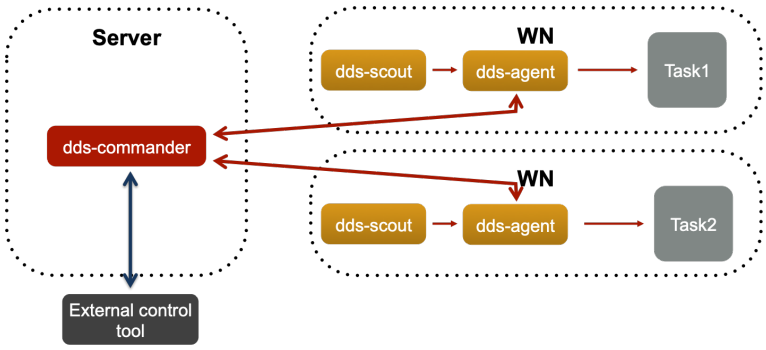

Figure 3. General DDS workflow.

4.2 A DDS protocol

DDS implements a custom lightweight protocol for messages and a variable size opaque body. The message consists of a fixed-size header and a body. The protocol API allows an

easy way to encode and decode messages to and from a byte array. The protocol is tightly integrated with the communication channels (see Section 4.3).

4.3 Communication channels

Communication between different processes in DDS is organized via so-called communication channels. There are two types of such channels: network channel and shared memory channel. Both channel types share unified event-based API for application and protocol events. Implementation utilizes modern C++ metaprogramming techniques and is done in a way that maximizes compile-time check for errors where possible. Communication channels support two-way communication, asynchronous read and write operations.

The implementation of the network channel is based on boost::asio [8] and uses the DDS protocol. Network channels are used for communication between DDS commander, agent, and for communication with external utilities which connect to DDS commander.

The implementation of the shared memory channel is based on the boost::message queue library [8], on the DDS protocol which is used for message encoding and decoding and on the boost::asio library [8] for thread pooling and implementation of the proactor design pattern. The shared memory channel is used for communication between agents when they are on the same host and between agents and their tasks. This architecture significantly improves performance and simplifies the implementation. All messages are stored directly in the shared memory and managed by the message queue.

4.4 Lobby-based deployment

The DDS must be able to handle hundreds of thousands of user processes. In DDS world each user process is controlled by a DDS agent (watchdog). Having all agents connecting back to a central DDS server (commander) would be extremely resource consuming. Hence, we implemented a so-called lobby-based deployment feature. DDS agents of a given user on one host represent a lobby. The lobby leader is the only agent, which has a direct network connection to the commander. The lobby leader is elected locally on each host. The election process is a local negotiation between agents and no connection to the commander is required. All other agents are lobby members and communicate with the commander via the leader. Agents of a given lobby communicate with each other via shared memory channels. Forming a lobby is a fully automated process which doesn't require additional configuration and human intervention. See Figure 4 for details.

4.5 DDS session

When performing offline data analysis, use cases exist, where a single user needs to run multiple different topologies hosting DDS commander on the same host. Moreover, for the Grid it is useful to run DDS in a batch mode. For all these use cases it is necessary to support multiple runtime topologies per user per host. In order to cover these use cases, a DDS session feature has been introduced. This feature offers users the possibility to run multiple commanders on the same host. Each new commander instance creates a DDS session. Sessions are sandboxed and isolated, therefore can't disturb each other. Sessions can be operated (listed, cleaned, sorted, etc) using the new dds-session command.

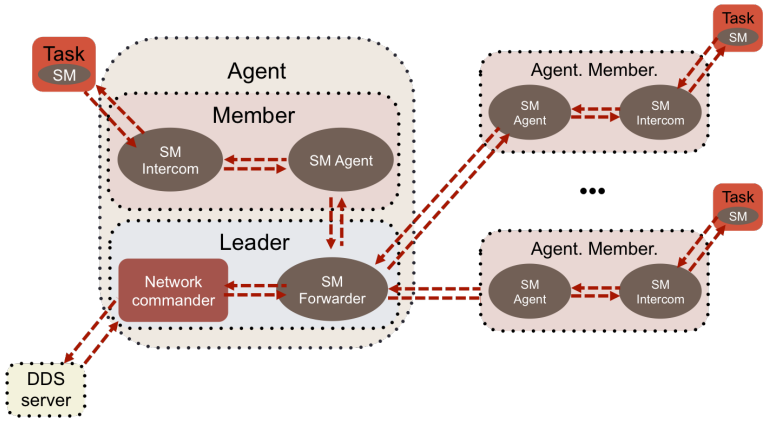

Figure 4. Schematic view of the lobby-based deployment

4.6 Automated testing

The growing complexity of DDS requires powerful functional tests, as most of the issues can only be detected during runtime when multiple agents are in use. Unit tests can't cover all edge cases. We, therefore, introduced a DDS octopus toolchain - a full-blown functional test machinery for DDS. DDS octopus features a rapid development of test cases with minimum code duplication. The toolchain is designed to run fully automatic without human intervention. It runs as a part of continues integration builds.

4.7 Development tools

DDS is being actively developed using modern development tools, C++11, and Boost libraries [8]. As the continuous integration framework, we use BuildBot [9]. The website and DDS's users manual are based on DocBook [10]. We developed and maintain a unique Git workflow to simplify and secure the development [11].

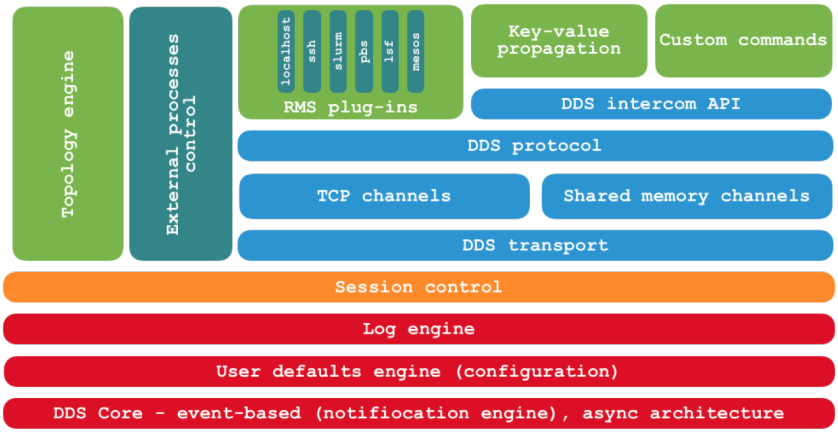

Figure 5. Overview of DDS components

The schematic view of various DDS components described in Figure 5.

5 Summary

DDS is equally good for work with different user requirements. For example, online cases in which a single DDS instance has to manage many ($\sim 100k$) processes. In offline scenarios each of the many DDS instances manage relatively small (~ 100) number of processes. Each DDS session can be a separate job which makes it a perfect tool for computing farms with RMS (Slurm, PBS etc.) as well as for the GRID (AliEn). One can even run DDS locally on a laptop for development or debugging purposes.

References

[1] DDS: The Dynamic Deployment System - http://dds.gsi.de
[2] DDS: The Dynamic Deployment System. Git repository - https://github.com/FairRootGroup/DDS
[3] M. Al-Turany et al, ALFA: ALICE-FAIR new message queuing based framework, presented on CHEP 2018, same proceedings
[4] Slurm - https://slurm.schedmd.com/
[5] PBS. Portable Batch System - http://www.pbspro.org/
[6] LSF. Platform Load Sharing Facility.
[7] Apache Mesos - http://mesos.apache.org/
[8] Boost C++ Libraries - https://boost.org
[9] Buildbot. The Continuous Integration Framework - http://www.buildbot.net
[10] DocBook - http://www.docbook.org
[11] A. Manafov, Git Workflow - https://github.com/AnarManafov/GitWorkflow