DDS
DYNAMIC DEPLOYMENT SYSTEM

Andrey Lebedev (GSI)
Anar Manafov (GSI)

September 13, 2018
GRID 2018 @ JINR, Dubna

Motivation

ALICE and FAIR experiments are moving from traditional one process to multiprocessing tools for simulation and reconstruction with topology and graphs.

In order to manage such an environment we need to create a system, which is able to **spawn and control hundreds of thousands** of different tasks which are tied together by a **topology**. It can run **on online clusters** or **computing clusters**, which use different **resource management systems (RMS)** or even on a **laptop** and can be **controlled by external tools**.

DDS is being developed within the **ALFA** framework (an **ALICE-FAIR** project).

Basic concepts

- **A single responsibility principle** command line **tool-set and API**;
- users' task is a black box – it can be an **executable or a script**;
- **watchdogging;**
- **rule-based execution of tasks**;
- **plug-in system** to abstract from RMS including **SSH** and a **localhost** plug-ins;
- **doesn't require pre-installation and pre-configuration** on the worker nodes;
- private facilities on demand with **isolated sandboxes**;
- key-value propagation and messaging.

The contract

The system takes so called "topology file" as the input.
Users describe desired tasks and their dependencies using this file.
Users are also provided with a Web GUI to create topologies.

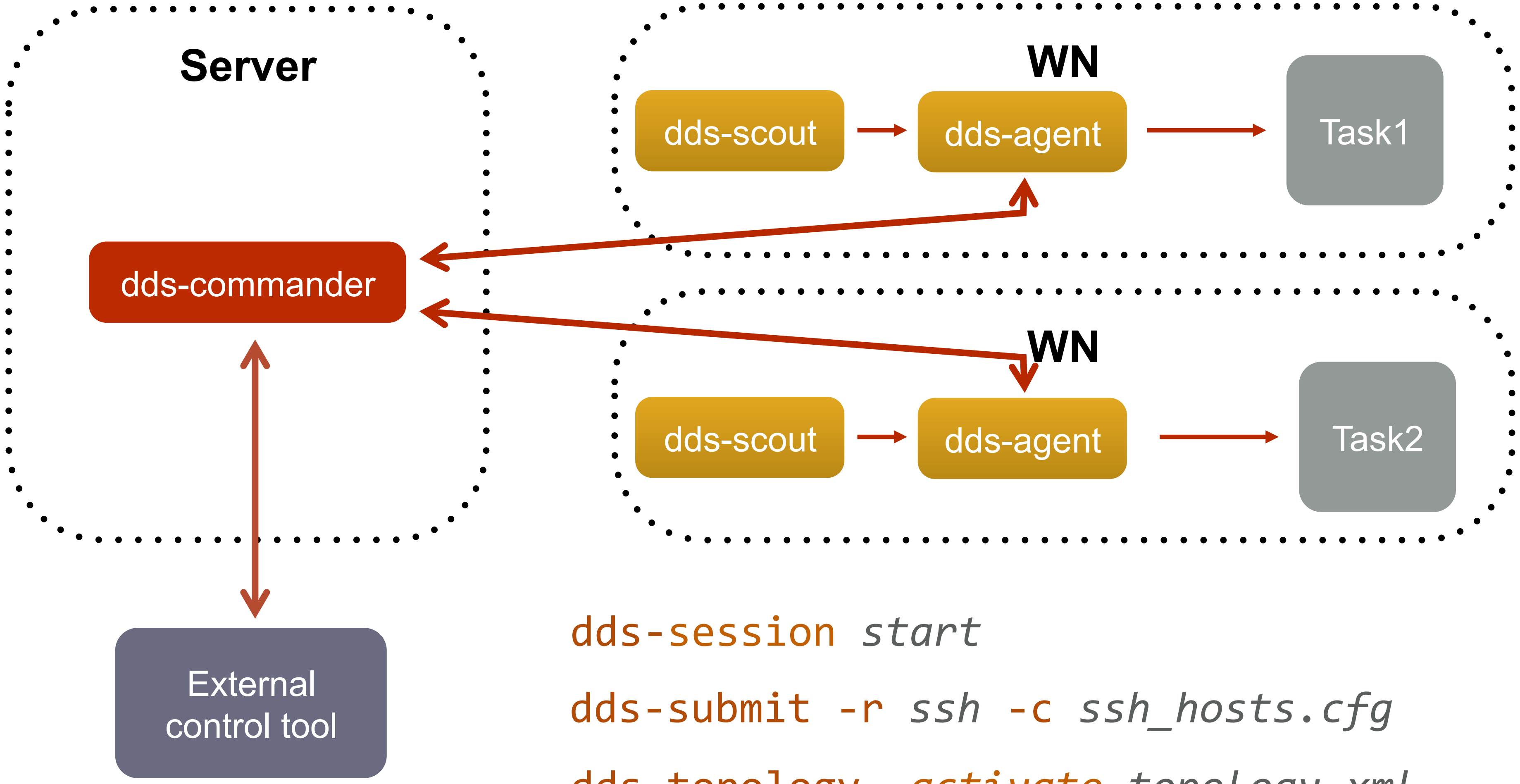
```
<topology id="myTopology">

  <decltask id="task1">
       <exe reachable="false">/Users/andrey/Test1.sh –l</exe>
   </decltask>

   <decltask id="task2">
       <exe>/Users/andrey/DDS/Test2.sh</exe>
   </decltask>

   <main id="main">
       <task>task1</task>
      <task>task2</task>
   </main>

</topology>
```

Declaration of user tasks. Commands with command line argument are supported.

Main block defines which tasks has to be deployed to RMS.

More info: http://dds.gsi.de/doc/nightly/topology.html

DDS workflow



```
dds-session start
dds-submit -r ssh -c ssh_hosts.cfg
dds-topology –activate topology.xml
dds-topology –update new_topology.xml
```

Highlights of the DDS features

- key-value propagation,
- Messaging (custom commands) for user tasks and ext. utils,
- RMS plug-ins,
- Watchdogging
  - … many more other features
    - more details here: https://github.com/FairRootGroup/DDS/blob/master/ReleaseNotes.md

Property propagation

The feature allows user's tasks to **exchange and synchronize the configuration (key-value)** dynamically at runtime.

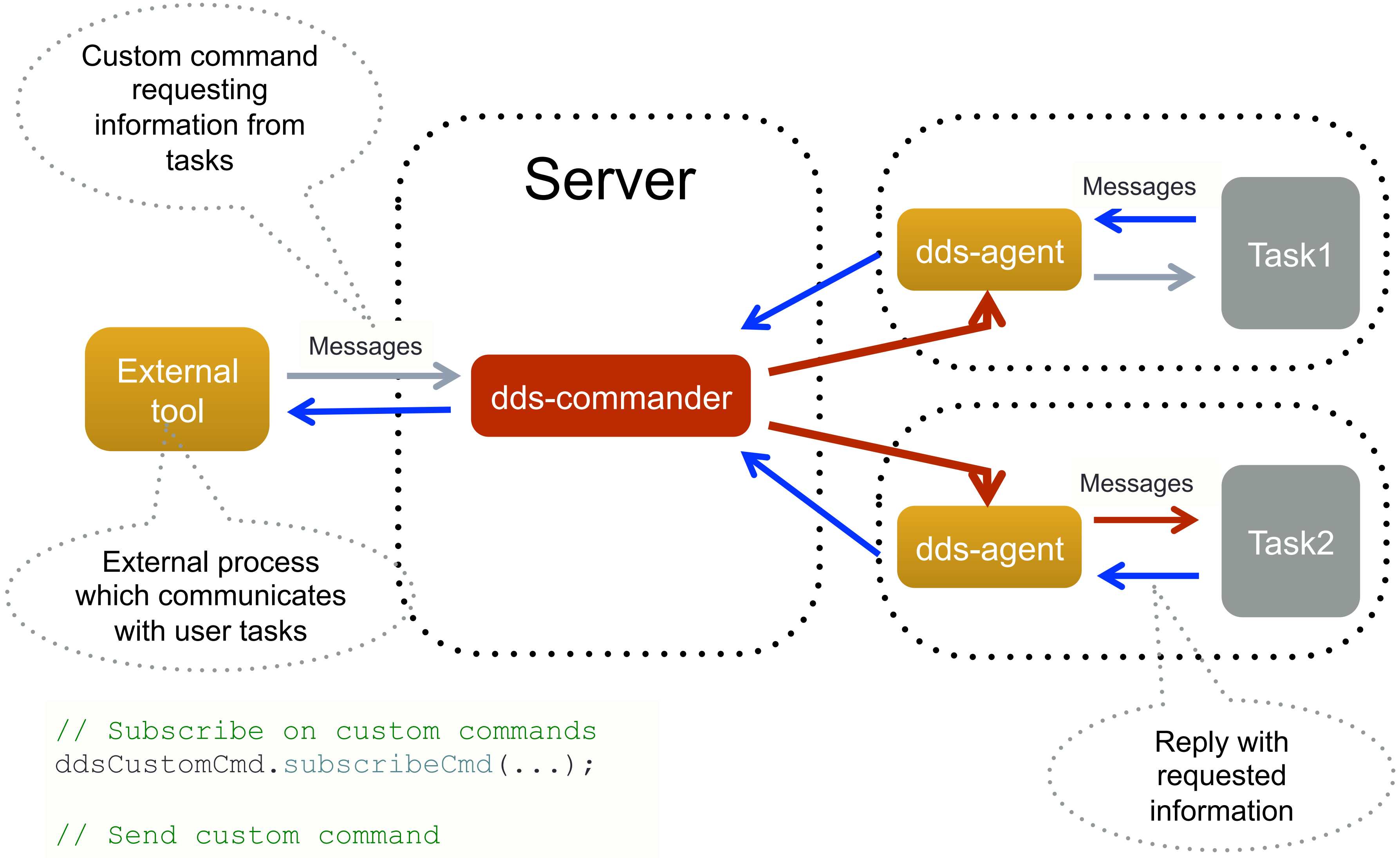
2 tasks → static configuration with shell script
Many tasks → dynamic configuration with DDS

Use case:
synchronize the startup of the user's tasks, for example, multiprocessing reconstruction based on FairMQ framework.

DDS is **highly optimized** for massive key-value transport and has a **decentralized architecture**:

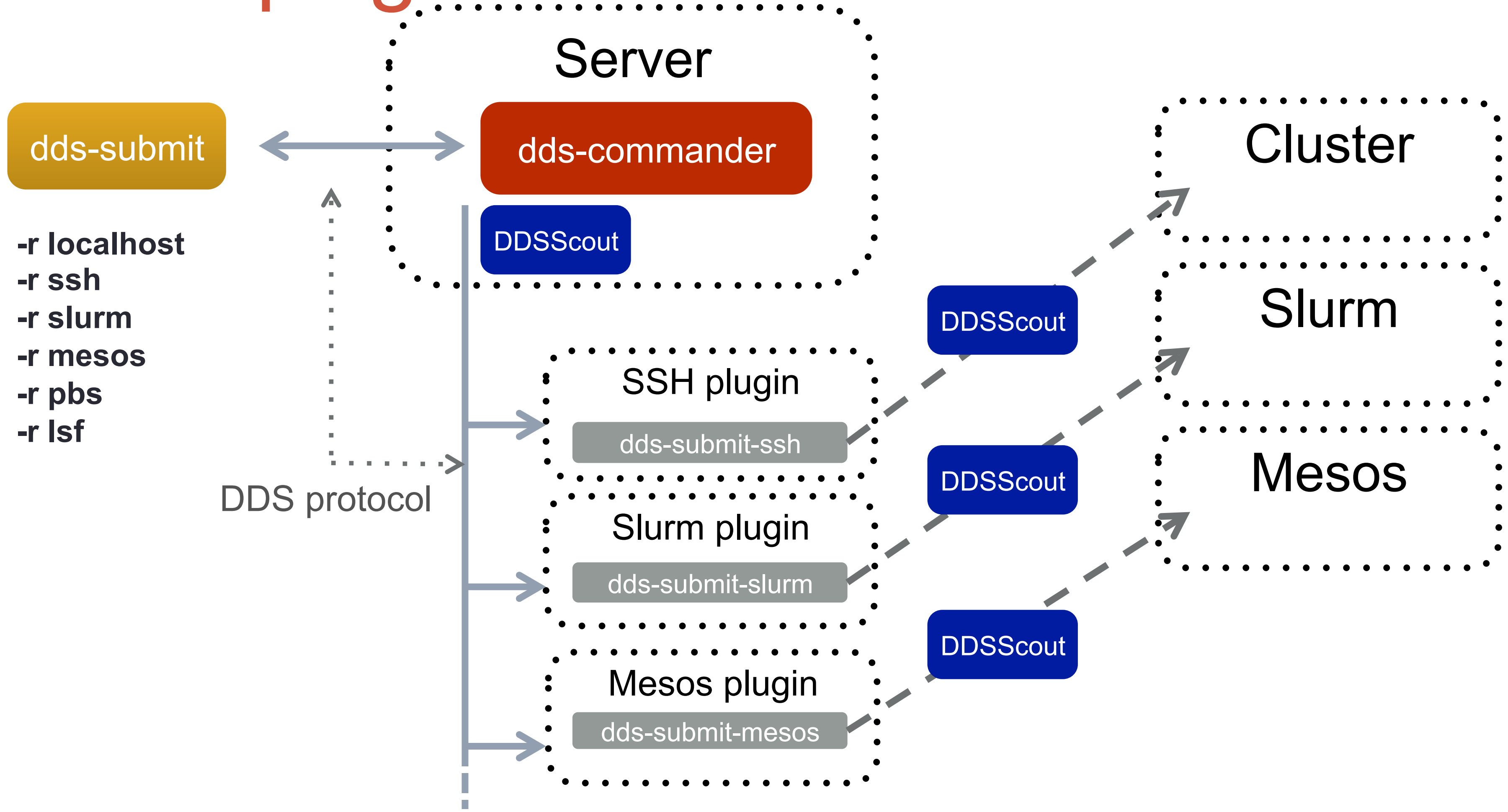
- *Internally small key-value messages are accumulated and transported as a single message;*
- *DDS agents use shared memory for local caching of key-value properties.*

Messaging (custom commands)


Custom command requesting information from tasks
Server
Messages
Messages
dds-agent
Task1
External tool
Messages
dds-commander
dds-agent
Messages
Task2
External process which communicates with user tasks
Reply with requested information

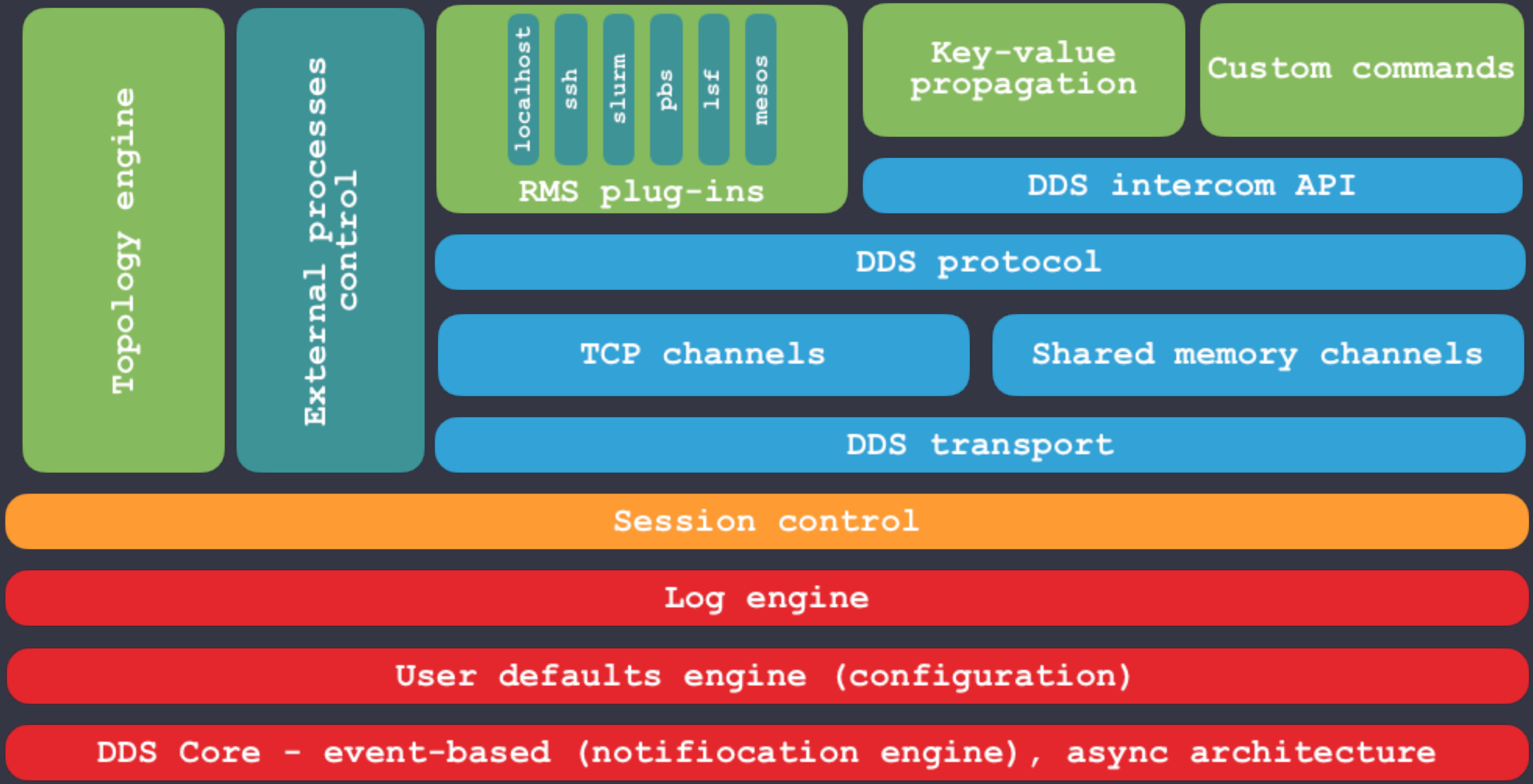
```
// Subscribe on custom commands
ddsCustomCmd.subscribeCmd(...);

// Send custom command
ddsCustomCmd.sendCmd(...);
```

RMS plug-in architecture



1. dds-commander starts a plug-in based on the dds-submit parameter,
2. plug-in contact DDS commander server asking for submissions details,
3. plug-in deploy DDSScout fat script on target machines,
4. plug-in execute DDSScout on target machines.

10000 feet view



From user's perspective

Topology

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

CLI tools

```
dds-session
dds-agent-cmd
dds-custom-cmd
dds-info
dds-prep-
worker
dds-server
dds-stat
dds-submit
dds-test
dds-topology
dds-user-
defaults
```

Intercom API

```
CIntercomService service;
CKeyValue
keyValue(service);

// Subscribe on key
update events
keyValue.subscribe([](
        const string&
_propertyID,
        const string& _key,
        const string& _value)
{…});

// Start listening to
events we have subscribed
on
service.start();
```

Usage scenarios

- Online
  - Single (multiple) DDS session managing many processes;
  - Goal ~100k processes
    - Tested up to 20k processes due to limited hardware
- Offline
  - Many DDS sessions
    - Each session manages small (~100) number of processes;
  - Computing farms with Slurm, PBS etc.;
  - GRID (AliEn)
    - Each GRID job is a DDS session;
- Local
  - on a laptop, for development and debugging.

DDS v2.0

- Releases - DDS v2.0
  - http://dds.gsi.de/download.html
- DDS Home site:
  - http://dds.gsi.de
- User's Manual:
  - http://dds.gsi.de/documentation.html
- Continues integration:
  - http://demac012.gsi.de:22001/waterfall
- Source Code:
  - https://github.com/FairRootGroup/DDS
  - https://github.com/FairRootGroup/DDS-user-manual
  - https://github.com/FairRootGroup/DDS-web-site
  - https://github.com/FairRootGroup/DDS-topology-editor

Backup slides

Shared memory communication

- Shared memory channel
  - Exactly the same event-based API as DDS network channel;
  - Duplex and many-to-many communication;
  - Asynchronous read and write operations;
  - Caching of messages in the queue for guaranteed delivery;
  - dds-protocol;
  - Efficient message forwarding.

- Implementation
  - **boost::message_queue**: message transport via shared memory;
  - **dds-protocol**: message definition, encoding, and decoding;
  - **boost::asio**: the proactor design pattern and an efficient thread pool.

Communication channels

- Network and shared memory channels;
- Unified event-based API for application and protocol events;
- Compile time check for errors where possible;

Subscribe to messages

```
client->registerHandler<cmdUPDATE_TOPOLOGY>(
    [](const SSenderInfo& _sender,
       SCommandAttachmentImpl<cmdUPDATE_TOPOLOGY>::ptr_t _attachment) {
       // User’s code
});
```

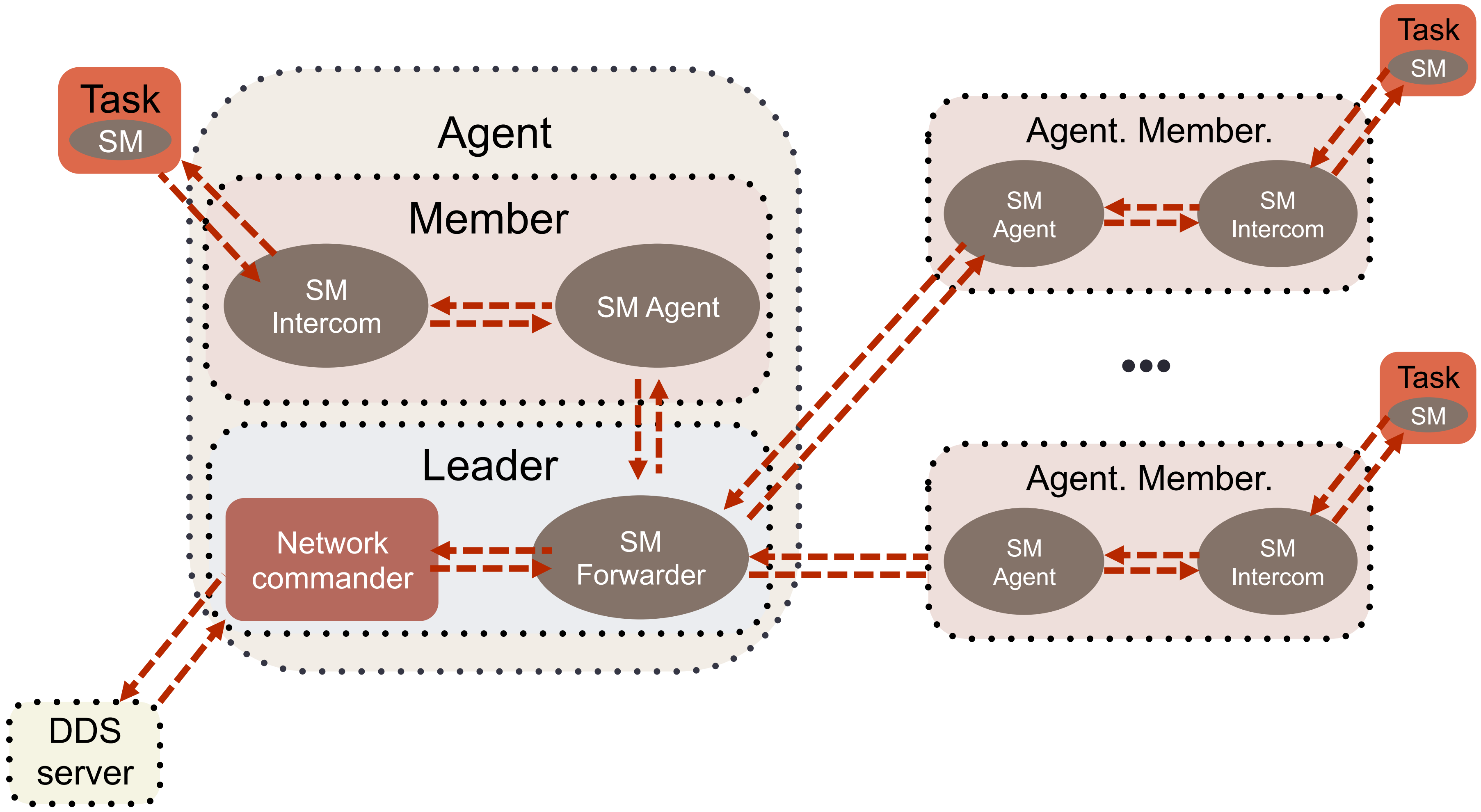
Subscribe to channel events

```
client->registerHandler<EChannelEvents::OnConnected>(
    [](const SSenderInfo& _sender) {
       // User’s code
});
```

Subscribe to messages

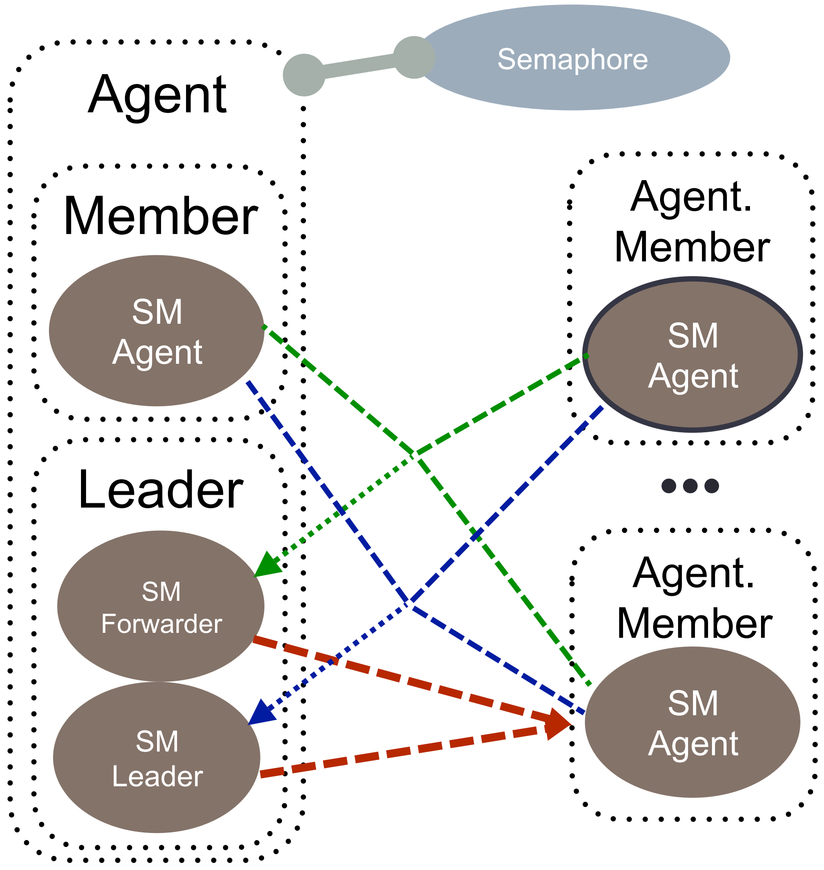
```
BEGIN_MSG_MAP(CInfoChannel)
    MESSAGE_HANDLER(cmdREPLY_PID, on_cmdREPLY_PID)
    MESSAGE_HANDLER(cmdREPLY_AGENTS_INFO, on_cmdREPLY_AGENTS_INFO)
END_MSG_MAP()
```

A lobby


Task
SM
Agent
Member
SM Intercom
SM Agent
Leader
Network commander
SM Forwarder
DDS server
Task
SM
Agent. Member.
SM Agent
SM Intercom
•••
Task
SM
Agent. Member.
SM Agent
SM Intercom

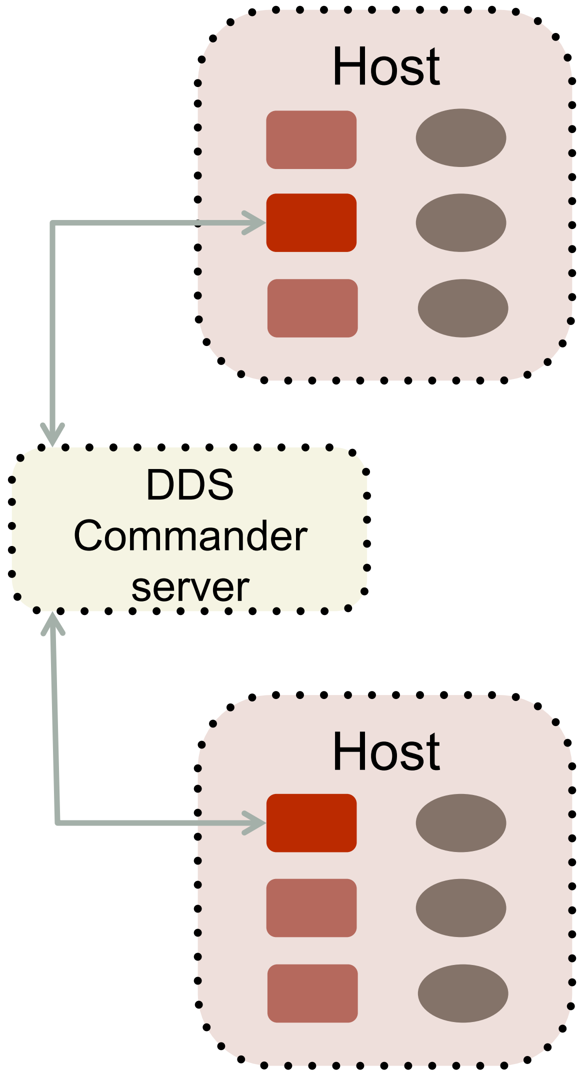
Lobby leader election

A lobby leader election: “First in takes all”.



- A leader is the one who first owns a SID semaphore;
- Each lobby member sends a special message to the leader with its connection information;
- The leader opens a channel and sends back a confirmation;
- Than a member sends a “lobby member handshake” message to Commander via SM Forwarder channel of the leader;
- Commander adds the agent to the list of approved agents;
- The communication is established.

Lobby based deployment



- One network connection per host;
- Local communications only via DDS shared memory channels;
- Unified agents and an unified lobby leader election;
- Efficient message forwarding;
- dds-protocol via network and shared memory channels;
- Handshake- and token-based authentication;