DDS

Dynamic Deployment System

Andrey Lebedev
Anar Manafov
GSI
2016-11-04

The Dynamic Deployment System

is a tool-set that automates and significantly simplifies a deployment of user defined processes and their dependencies on any resource management system using a given topology

Highlights since last meeting

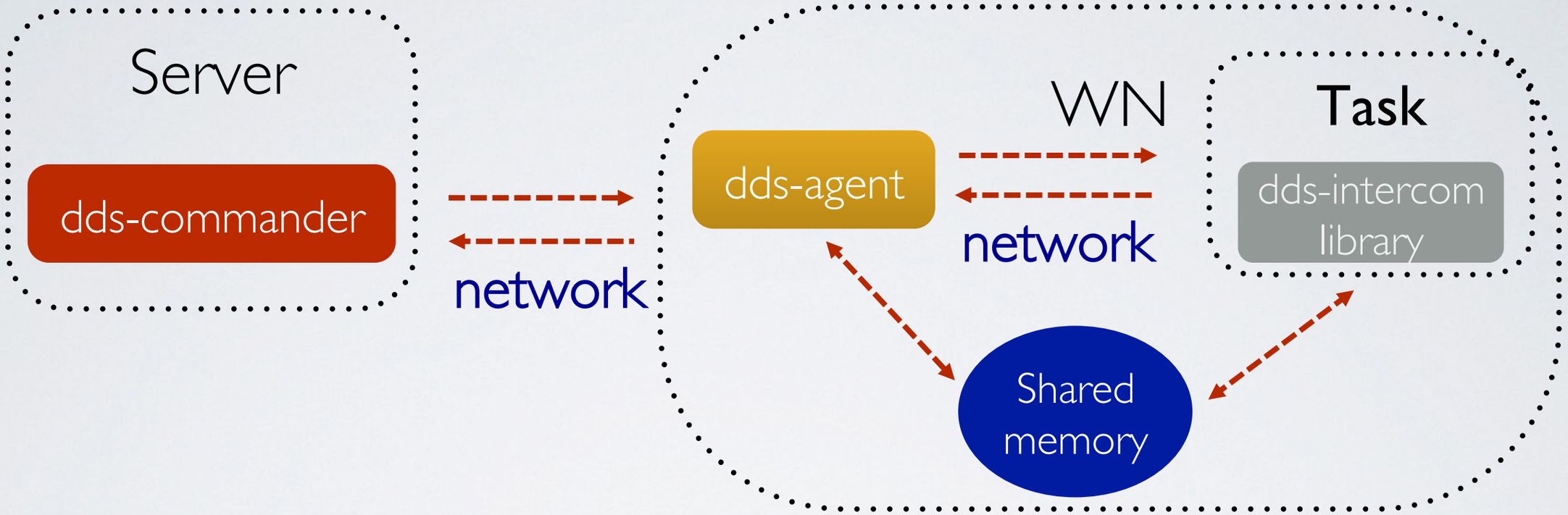
1. Shared memory communication.
2. New dds-intercom library API.
3. Versioning in key-value propagation.
4. Runtime topology update.
5. LSF and PBS plugins.
6. dds-octopus: test DDS using DDS.

… many more other fixes and stability improvements

more details here: https://github.com/FairRootGroup/DDS/blob/master/ReleaseNotes.md

Shared memory communication [1]

Motivation. Problem.



1. Because of the network connection there was no guarantee that key-value update notification will be delivered to the user.
2. That's why shared memory was used as a cache and network was used to notify the library that something was changed in the cache or to update the cache.
3. dds-intercom API was adapted to such usage pattern: getValues() method returned current values in the cache.

Shared memory communication [2]

Shared memory channel:

- Similar API as DDS network channel;
- Two way communication;
- Asynchronous read and write operations;
- dds-protocol.

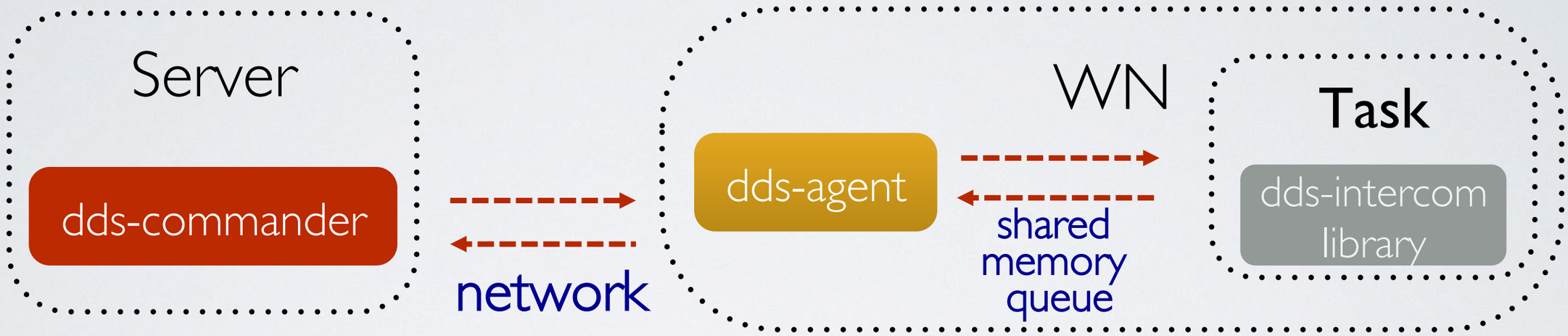
Implementation:

1. **boost::message_queue:** message transport via shared memory;
2. **dds-protocol:** message encoding and decoding;
3. **boost::asio:** proactor design pattern, thread pool.

key-value and custom commands

NEW

Shared memory communication
between dds-agent and user task



No need to cache messages in the dds-intercom-lib – we guarantee that the message will be delivered. Messages are stored directly in the shared memory and managed by the message queue.

2x better performance for our test case:
40 tasks intensively exchanging key-values on a single node with 40 logical cores.

New DDS intercom API

```cpp
#include "dds_intercom.h"

CIntercomService service;
CKeyValue keyValue(service);

// Subscribe on error events
service.subscribeOnError([](EErrorCode _errorCode, const string& _msg) {
    // Service error
});

// Subscribe on key update events
keyValue.subscribe([](const string& _propertyID, const string& _key, const string& _value) {
    // Key-value update event received
});

// Subscribe on delete key notifications
keyValue.subscribeOnDelete([](const string& _propertyID, const string& _key) {
    // Key-value delete event received
});

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

keyValue.putValue("prop_1", "prop_1_value");
```

getValues() method removed from API:

user's task has to cache key-value messages if required

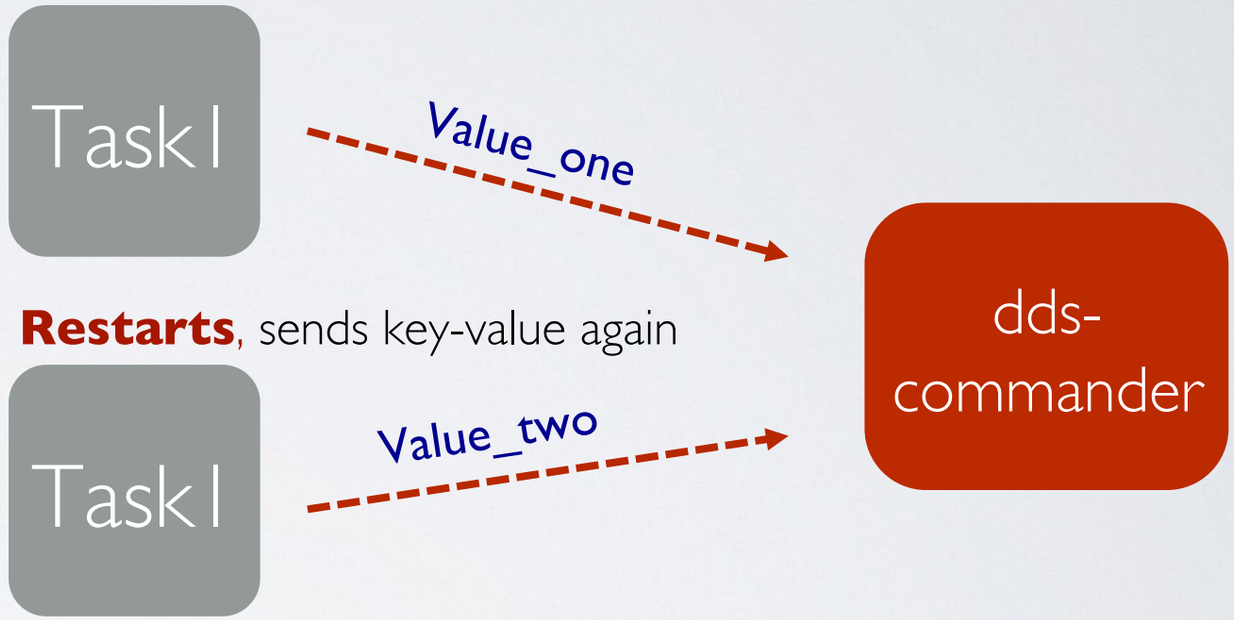
Versioning in key-value propagation [1]

Single property in the topology – multiple keys at runtime.

A certain key can be changed only by one task instance.

Why do we need versioning?

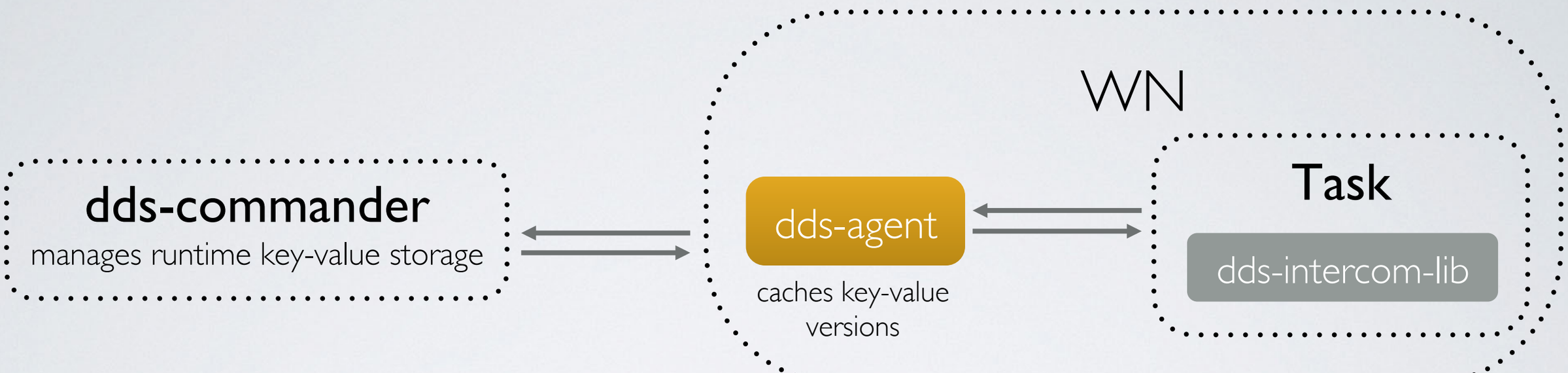
Starts, sends key-value and dies



If Value_two arrives first, than it will be overwritten by Value_one and all tasks will be notified with the wrong value.

Versioning in key-value propagation [2]

Versioning is completely hidden from the user.



1. Task sets key-value using dds-intercom-lib.
2. dds-agent sets version for key-value and sends it to dds-commander.
3. dds-commander checks version:

a) if version is correct than it updates version in storage and broadcasts the update to all related dds-agents;

b) in case of version mismatch it sends back an error containing current key-value version in the storage. dds-agent receives error, updates version cache and forces the key update with the latest value set by the user.

Runtime topology update [1]

Update of the currently running topology without stopping the whole system.

Steps:

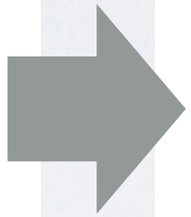
1. Get the **difference** between current and new topology. The algorithm calculates hashes for each task and collection in the topology based on the full path and compares them. As a result a list of removed tasks and collections and a list of added tasks and collections are obtained.
2. **Stop** removed tasks and collections.
3. **Schedule and activate** added tasks and collections.

Limitation:
Declaration of tasks and collections can't be changed.

```
<decltask id="task1">
    <exe>/Users/andrey/DDS/task1.sh</exe>
    <properties>
        <id access="write">property1</id>
    </properties>
</decltask>
```

Runtime topology update [2]

```
<main id="main">
    <task>task1</task>
    <collection>collection1</collection>
        <group id="group1" n="2">
                <task>task1</task>
                <task>task2</task>
                <collection>collection1</collection>
        </group>
        <group id="group2" n="3">
                <task>task1</task>
        </group>
</main>
```



```
<main id="main">
        <task>task3</task>
        <group id="group1" n="3">
                <task>task1</task>
                <task>task2</task>
                <collection>collection1</collection>
        </group>
        <group id="group2" n="1">
                <task>task1</task>
                <task>task3</task>
        </group>
</main>
```

Runtime topology update [3]

dds-topology *--update new_topology.xml*

```
dds-topology: Contacting DDS commander on visitor-16386626.dyndns.cern.ch:20001  ...
dds-topology: Connection established.
dds-topology: Requesting server to update a topology...
dds-topology: Updating topology to /Users/andrey/DDS/1.3.25.gbf0d0fc/../topology_test_diff_2.xml
dds-topology:
Removed tasks:5
1 x main/collection1/task1
1 x main/collection1/task2
2 x main/group2/task1
1 x main/task1
Removed collections:1
1 x main/collection1
Added tasks:6
1 x main/group1/collection1/task1
1 x main/group1/collection1/task2
1 x main/group1/task1
1 x main/group1/task2
1 x main/group2/task3
1 x main/task3
Added collections:1
1 x main/group1/collection1

dds-topology: Stopping removed tasks...
[==================================================] 100 % (5/5)
Stopped tasks: 5
Errors: 0
Total: 5
Time to Stop: 1.003 s
dds-topology: Activating added tasks...
[==================================================] 100 % (6/6)
Activated tasks: 6
Errors: 0
Total: 6
Time to Activate: 0.099 s
```

dds-octopus

Motivation: Growing complexity of DDS requires powerful functional tests. Unit tests can't cover all cases. Most of issues can be only detected during run-time when multiple agents are in use.

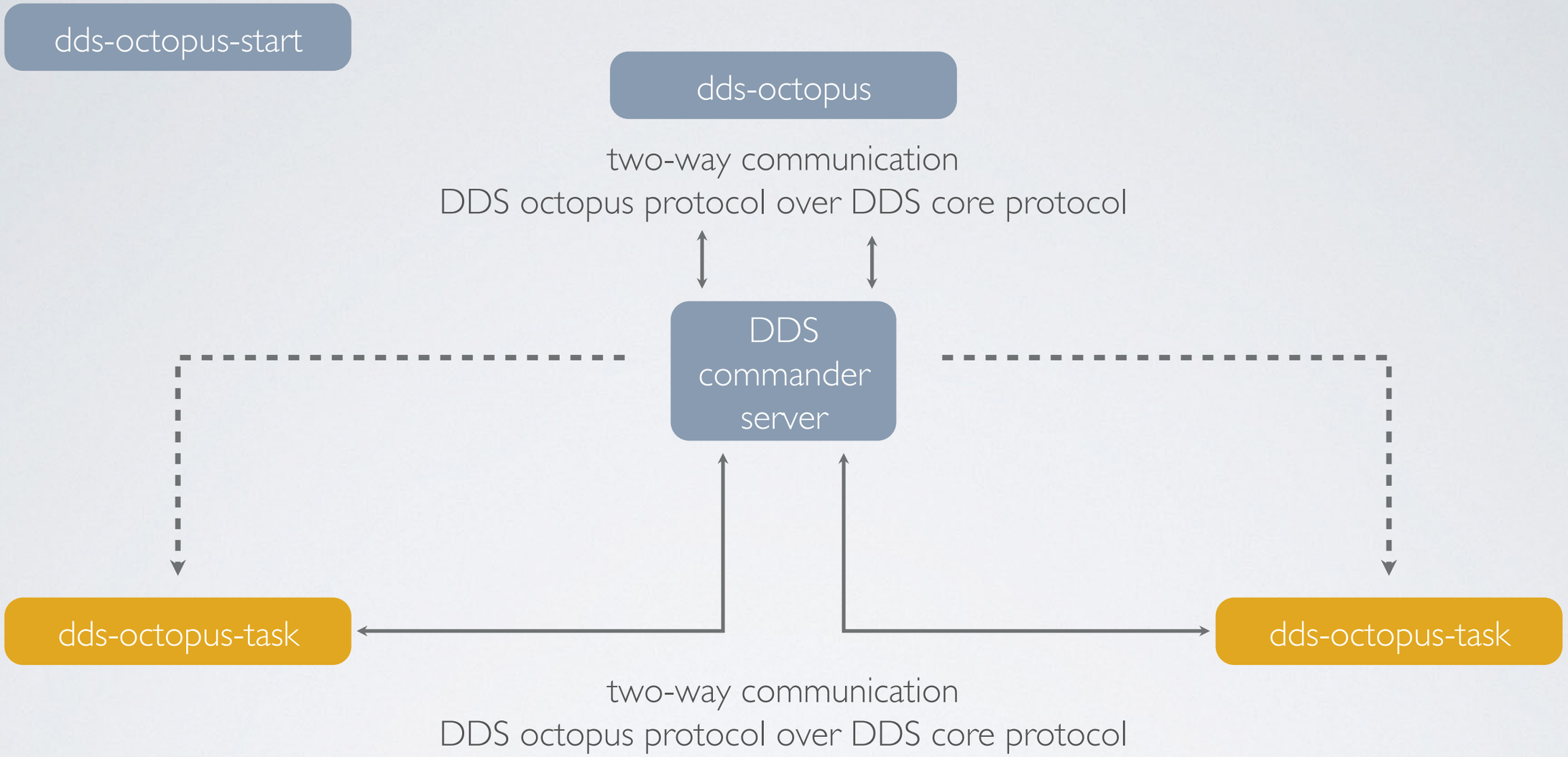
dds-octopus: A full blown functional test machinery for DDS.
Test DDS using DDS.

dds-octopus: core components

dds-octopus-start - a steering script. Starts DDS, deploys agents and activates predefined topologies. It validates return values of all used DDS commands and timeouts on each of them.
dds-octopus-task - an executable. Acts as a regular "user" task. This the same task is part of all test topologies.
dds-octopus - an executable. Acts as external test manager. It is not a part of topology.

dds-octopus architecture

deploys DDS on a localhost and initiates dds-octopus



- dds-octopus implements a set of test cases,
- tests cases send commands to tasks,
- tasks are expected to react according to some defined pattern,
- a test case fail if: a timeout is reached, task reacted with unexpected pattern.

dds-octopus key features

1. Rapid development of test cases with minimum code duplication.
2. Flexible communication protocol (json over DDS custom command API).
3. Developers when creating test cases use commands from a predefined list to form a call chain. List can be extended with new commands.
4. dds-octopus-task doesn't aware of test cases, rather it just simply knows what to do when a certain command is received. Combing different commands in a chain we can create different test cases expecting some certain behaviour from the task or the whole system.
5. Test cases use responses from tasks to decide whether the test succeeded or failed. Test cases can also use dds-commander's info API to query states of the DDS deployment from the main source.
6. Each test case must define a timeout. When dds-octopus detects that the timeout is reached it marks the test case as failed.
7. The toolchain is designed to run fully automatic without human intervention. Since DDS 1.4 it will run as a part of continues integration builds.

RMS plug-ins

1. localhost,
2. ssh,
3. slurm,
4. Mesos,
5. PBS, NEW
6. LSF. NEW

- Releases - DDS v1.4

  (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

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 rules based execution of tasks.

```
CRMSPluginProtocol prot("plugin-id");
```

(1)

```
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.
});

prot.onRequirement([](const SRequirement& _requirement) {
        // Implement functionality related to requirements here.
});
```

(2)

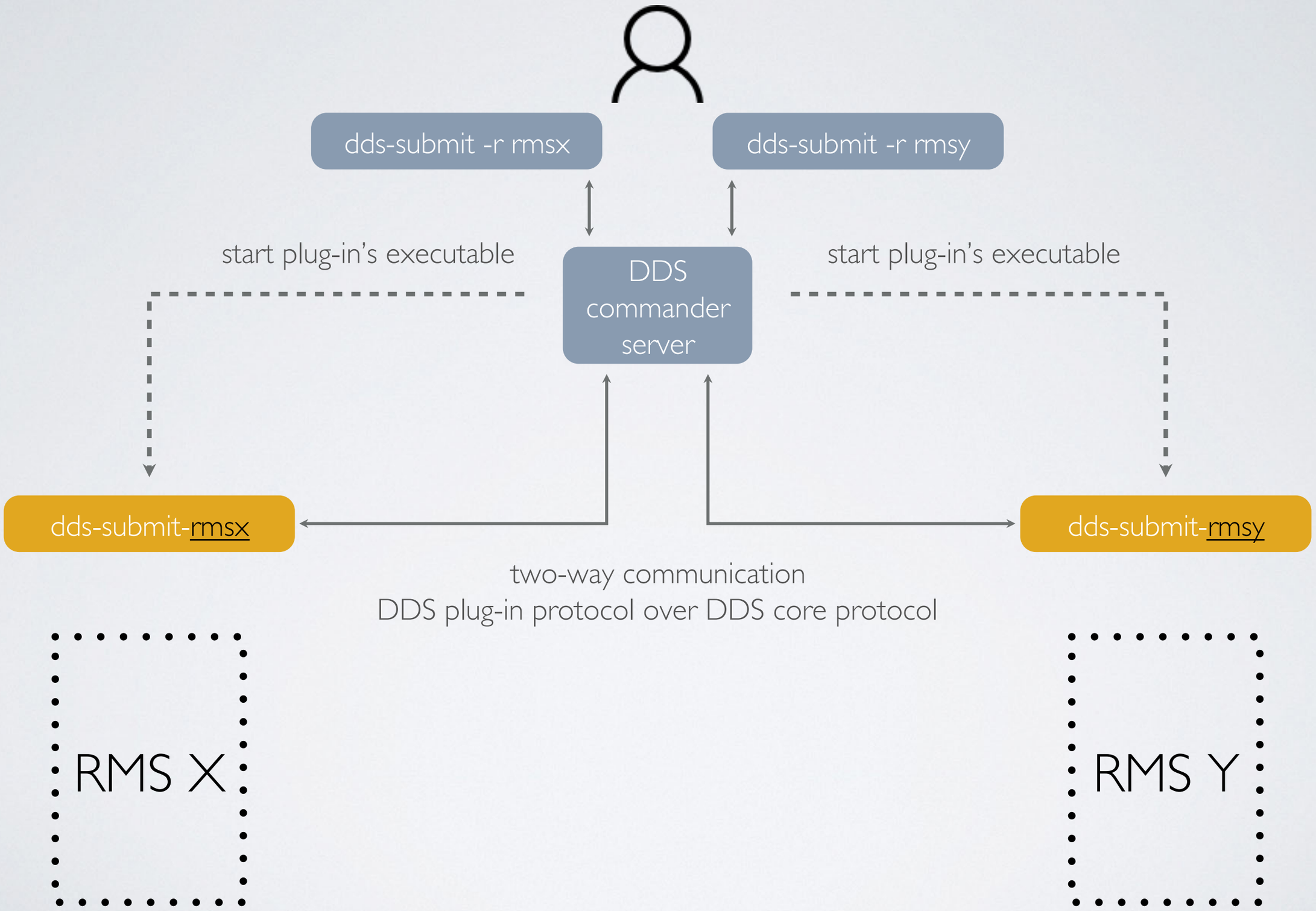
```
// Let DDS commander know that we are online and start listen for messages.
prot.start(bool _block = true);
```

(3)

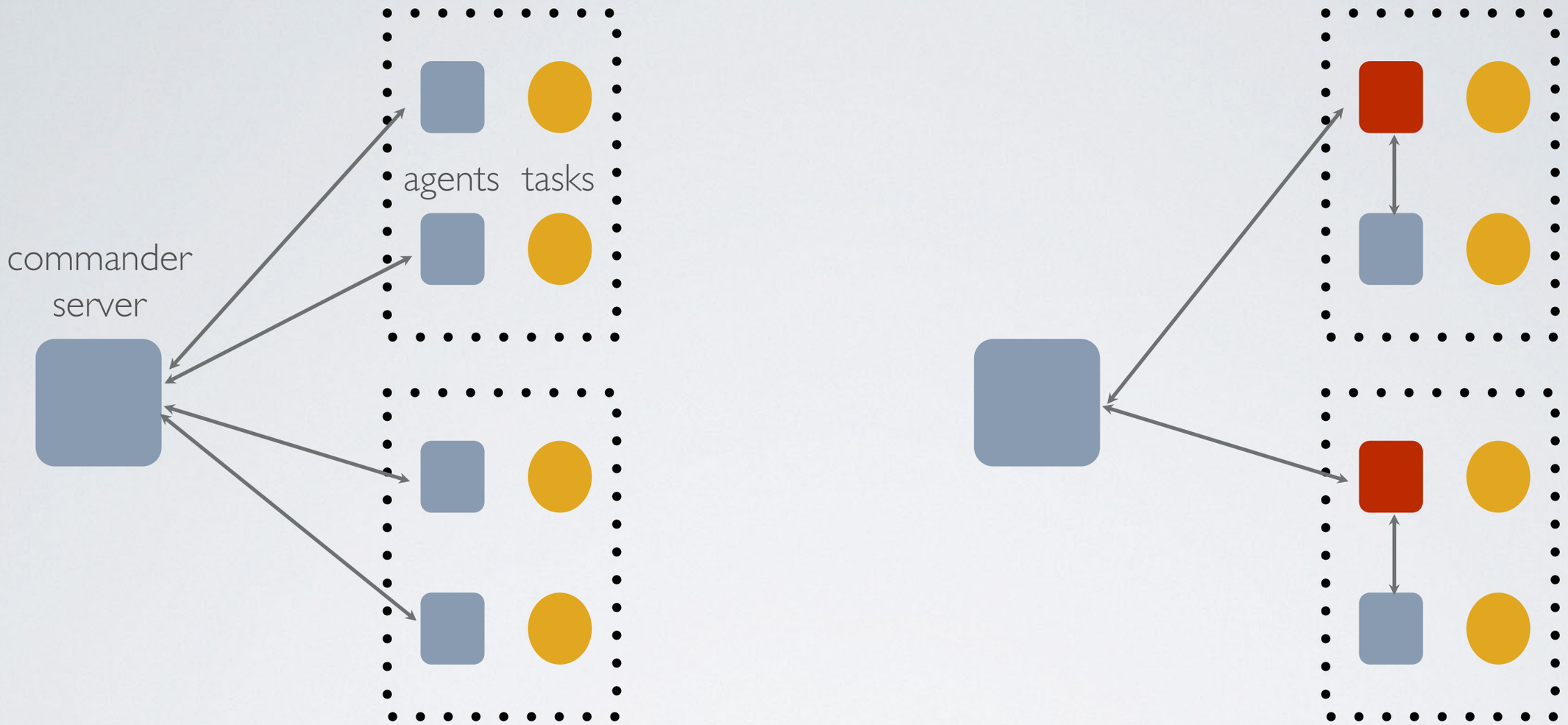
```
// Report error to DDS commander
proto.sendMessage(dds::EMsgSeverity::error, “error message here”);

// or send an info message
proto.sendMessage(dds::EMsgSeverity::info, “info message here”);
```

New RMS plug-in architecture



Lobby based deployment



1. DDS Commander will have one connection per host (lobby),
2. lobby host agents (master agents) will act as dummy proxy services, no special logic will be put on them except key-value propagation inside collections,
3. key-value will be either global or local for a collection