Tarantool Enterprise Documentation

Release 1.0

Mail.Ru, Tarantool team

Jul 24, 2019

Contents

Tarantool Enterprise documentation

This product is the Enterprise edition of Tarantool software – a DBMS for deploying fault-tolerant distributed data storages.

This document primarily concentrates on distinctive features of the Tarantool Enterprise edition. The most relevant and exhaustive documentation on the Tarantool DBMS is available in the Tarantool manual.

CHAPTER 1

Installation

1.1 Package contents

The package includes the following components and features:

- Tarantool Enterprise DBMS server,
- automatic sharding (scaling-out) system for Tarantool DBMS instances,
- Tarantool DBMS automatic resharding system,
- replication system module for reducing conflicts related to master-master interaction,
- service for processing distributed requests to cluster nodes,
- cluster administration service.

The product consists of three standalone applications:

- `router` is a service that processes distributed requests to cluster nodes,
- `storage` is a data storage service,
- `orchestrator` is a cluster administration service.

The purpose of each application and component is described in a corresponding section further in this document.

Tarantool Enterprise DBMS for distributed fault-tolerant data storages is delivered as a `tar + gzip` archive.

Archive contents:

- `./artifacts/` are the binary artifacts (packages) of the delivery, e.g. `bin-tarantool-app.tar.gz` is the Tarantool Enterprise DBMS and its components,
- `./common/` are the Ansible roles from Common Installer,
- `./pack-config.sh` is the script for packaging `/config/` into artifacts,
- `./tarantool-deploy.yml` is the Ansible Playbook of the installer,
- `./tarantool-deploy.sh` is the script for launching the installer,

- `./inventory/` is the Ansible `inventory` of the installer,
- `./config/` are the configuration file templates:
  - `router.lua` is the router configuration template,
  - `storage.lua` is the storage configuration template,
  - `orchestrator.yml` is the orchestrator's main configuration template,
  - `orchestrator.logging.ini` is the orchestrator's logging configuration template,
  - `configure_zk.sh` is the script template for the initial configuration of a cluster in the ZooKeeper and orchestrator,
  - `storage-init.sh` is the init-script template for launching the storage,
  - `router-init.sh` is the init-script template for launching the router,
  - `orchestrator-init.sh` is the init-script template for launching the orchestrator.

1.2 System requirements

1.2.1 Hardware requirements

To fully ensure the fault tolerance of a distributed data storage system, at least **three** physical computers or virtual servers are required. For testing/development purposes, the system can be deployed using a smaller number of servers; however, it is not recommended to use such configurations for production.

Hereinafter, **"storage servers"** or **"Tarantool servers"** are the computers used to store and process data, and **"administration server"** is the computer used by the system operator to install and configure the product.

1.2.2 Software requirements

All servers must be running Red Hat Enterprise Linux 7 (any build). The following packages should be installed and configured in addition to the basic OS functionality:

- Tarantool servers:
  - `openssh-server` (any version) for providing remote access to the virtual server,
  - `tar`, `gzip`, `bzip2` (any version) for unpacking archives,
  - `less` (any version) for viewing log files,
  - `zookeeper-server` and `zookeeper` (version 3.4) are the ZooKeeper server and client for centralized configuration management,

  Note: This package is not available in the main RHEL 7 repository, so it should be installed from additional repositories (for example, Cloudera CDH 5), or using any other method.

  - `openssl-libs` (version 1.0.2k) is a mandatory component for launching Tarantool DBMS. It usually comes as a default system component.
- administration server:
  - `openssh-server` (any version) for remote access to the virtual server,
  - `tar`, `gzip`, `bzip2` (any version) for unpacking archives,

  - `ansible` (version 2.2 or 2.4) for automated cluster deployment,
  - `make` (any version) for task automation,
  - `vim` (any version) for editing configuration files,
  - `less` (any version) for viewing log files,
  - `curl` (any version) for testing the REST API orchestrator,
  - `tmux` (any version) or `screen` (any version) for multiplexing the terminals and improving the usability.

1.2.3 Network topology requirements

- Administration server should be able to access TCP port 22 on Tarantool servers using the SSH protocol.
- Administration server should be able to access TCP port 8080 on Tarantool servers to use the API for centralized cluster management.
- Administration server should be able to access TCP port 8000 on Tarantool servers to integrate with a monitoring system.
- Tarantool servers should be able to communicate and send traffic from any TCP port to TCP ports 3000:4000 to ensure that the Tarantool cluster is functioning properly.
- Tarantool servers should be able to communicate and send traffic from any TCP port to TCP port 2181 to ensure that ZooKeeper is functioning properly.

1.3 Setup

1.3.1 Setting up system users

1. On all Tarantool servers, create and enable an *administrator* user, on behalf of whom the product should be installed:

   ```
   $ useradd USER_NAME -m
   $ password USER_NAME
   ```

   After executing the second command, enter and remember the password.

2. On all Tarantool servers, create and enable a `tarantool` user:

   ```
   $ useradd tarantool -m -d /data/tarantool
   $ password tarantool
   ```

   After executing the second command, enter and remember the password.

3. On all Tarantool servers, create a `/data/logs/tarantool` directory and give the `tarantool` user the privilege to write to this directory:

   ```
   $ mkdir -p /data/logs/tarantool
   $ chown tarantool:tarantool /data/logs/tarantool
   ```

4. On behalf of the server administrator, enable `sudo -u tarantool` for the `tarantool` user on all Tarantool servers:

   ```
   $ echo "USER_NAME ALL=(tarantool) ALL" > /etc/sudoers.d/tarantool
   ```

5. (optional) On the administration server, generate an SSH key with no passphrase:

   ```
   $ ssh-keygen -t rsa
   Generating public/private rsa key pair.
   Enter file in which to save the key (/home/<user_name>/.ssh/id_rsa): <ENTER -␣
   ↪default file>
   Enter passphrase (empty for no passphrase): <ENTER - empty passphrase>
   Enter same passphrase again:  <ENTER - empty passphrase>
   Your identification has been saved in id_rsa.
   Your public key has been saved in id_rsa.pub.
   The key fingerprint is:
   ```

6. (optional) Add the public part of the generated key to the list of authorized keys for the `administrator` user created on the Tarantool servers:

   ```
   $ ssh-copy-id USER_NAME@Tarantool_server
   ```

After executing this command, enter the password for the `administrator` user created in step 1.

If steps 5-6 are performed correctly, then the administration server allows using the SSH protocol to connect to all Tarantool servers on behalf of the `USER_NAME`, **without entering a password**:

```
admin_server$ ssh <Tarantool_server>
Tarantool_server$ # access granted without password
```

If steps 5-6 are skipped, the administration server allows using the SSH protocol to connect to all Tarantool servers on behalf of the `USER_NAME` **after entering the password**. Both options are valid and supported by the automatic installer.

If step 3 was performed correctly, all Tarantool servers allow switching the current user from `USER_NAME` to `tarantool`:

```
Tarantool_server$ sudo -u tarantool -i
Tarantool_server$ whoami
tarantool
Tarantool_server$ pwd
/data/tarantool
```

After executing this command, enter the password for the administrator user created in step 1.

1.3.2 Setting up systemd-logind

1. Open the `/etc/systemd/logind.conf` file on each Tarantool server and change the value of the `RemoveIPC` parameter to `no`.
2. Restart the `systemd-logind` service by executing `systemctl restart systemd-logind`.

Thus you guarantee that the running services cannot be stopped by the `systemd` services after logging out (for example, when the SSH client is disconnected). For more information, please refer to the RHEL 7 user documentation.

1.3.3 Setting up name resolution

It is recommended to assign a unique hostname to each Tarantool server and specify it in the DNS or in the `/etc/hosts` file on all Tarantool servers and on the administration server. You can skip this step if IPv4 addresses are used instead of host names.

1.3.4 Setting up sysctl

It is recommended to apply the following settings for `sysctl` on all Tarantool servers:

```
$ # TCP KeepAlive setting
$ sysctl -w net.ipv4.tcp_keepalive_time = 60
$ sysctl -w net.ipv4.tcp_keepalive_intvl = 5
$ sysctl -w net.ipv4.tcp_keepalive_probes = 5
```

This optional setup of the Linux network stack helps speed up the troubleshooting of network connectivity when the server physically fails. To achieve the maximum performance, you may also need to configure other network stack parameters that are not specific to the Tarantool DBMS. For more information, please refer to the Network Performance Tuning Guide section of the RHEL7 user documentation.

1.3.5 Setting up ZooKeeper

Apache ZooKeeper service is required for centralized configuration of a distributed data storage. It is recommended to install the ZooKeeper service on all Tarantool servers, however, a dedicated ZooKeeper server can also be used. If the ZooKeeper service has already been installed for other applications, skip this step.

Note: `zookeeper-server` package is not present in the main RHEL 7 repository, so it should be installed from additional repositories (for example, Cloudera CDH 5), or using any other method.

```
$ yum install -y https://archive.cloudera.com/cdh5/one-click-install/redhat/7/x86_64/
↪cloudera-cdh-5-0.x86_64.rpm
$ yum install -y zookeeper-server java
```

Contents of the ZooKeeper configuration file should be as follows:

```
$ # ticks per millisecond
$ tickTime=2000
$ # a port for client connection
$ clientPort=2181
$ # all ZooKeeper services are specified below
$ # the first port is used by the followers to connect to the leader-service
$ # the second port is used to choose the leader
$ server.1=<IP-address of the Tarantool server 1>:3887:3888
$ server.2=<IP-address of the Tarantool server 2>:3887:3888
$ ...
$ server.n=<IP-address of the Tarantool server n>:3887:3888
```

After making changes to the configuration, set the ZooKeeper host ID:

```
$ MYID=<ID of the ZooKeeper service, c 1[a]> /etc/init.d/zookeeper-server init
```

A globally unique `MYID` must also match the `server.1`, `server.2`, `server.n` records. In other words, `MYID=1` should be set on server `<Tarantool 1 IP address>`, while `MYID=2` should be set on server `<Tarantool 2 IP address>` and so on.

ZooKeeper can be started in a regular way:

```
$ /etc/init.d/zookeeper-server start
```

To check the operability of ZooKeeper, try:

```
$ # echo "stat"| nc 127.0.0.1 2181
$ Zookeeper version: 3.4.8-1--1, built on Fri, 26 Feb 2016 14:51:43 +0100
$ Clients:
$ /127.0.0.1:59139[0](queued=0,recved=1,sent=0)
$ Latency min/avg/max: 0/6/10
$ Received: 11
$ Sent: 10
$ Connections: 1
$ Outstanding: 0
$ Zxid: 0x600000039
$ Mode: follower
$ Node count: 17
```

One of the servers should be in the `leader` mode, the others should be in the `follower` mode.

1.4 Automatic installation

The product is automatically installed using the Ansible-based installation package.

The installation process powered by the automatic installer includes the following steps:

1. *Unpacking the archive*.
2. *Setting up the inventory*.
3. *Launching the automatic installer*.
4. *Deploying the Tarantool cluster*.
5. *Launching the Tarantool cluster*.
6. *Initializing the cluster*.

All operations are performed on the administration server. The details of each step are provided further in this document.

1.4.1 Unpacking the archive

The delivered archive should be uploaded to the administration server and unpacked:

```
admin_server$ tar xvf bin-tarantool-app.tar.gz
admin_server$ cd bin-tarantool-app.tar.gz
```

1.4.2 Setting up the inventory

To automatically install and configure the package, you should specify the desired cluster topology in the `inventory` file. The `inventory` template is located in the `inventory/localhost/1-bin-tarantool.yml` file. Shared variables are located in the `inventory/localhost/group_vars` files.

Basic configuration parameters:

- **`zoo_host` contains the hostname and the ZooKeeper server port.** It is recommended to install ZooKeeper on all Tarantool servers. If ZooKeeper is present on all Tarantool servers, then `zoo_host` points to the local ZooKeeper node. If ZooKeeper is installed on a dedicated computer, then the address and port of the ZooKeeper service should be specified.

- `ansible_host` is the hostname of a physical server or a virtual machine to install the Tarantool instance,
- `tarantool_user` and `tarantool_password` are the username and password to provide access through the Tarantool client library using the binary protocol,
- `tarantool_sharding_user` and `tarantool_sharding_password` are the internal username and password to be used by the sharding and replication system,
- `tarantool_memtx_memory` is the number of bytes allocated to the in-memory storage,
- `tarantool_uuid` is the Tarantool node's UUID that should be unique for each node of the cluster,
- `tarantool_replicaset` is the UUID of a replica set in the Tarantool cluster; nodes that have the same UUID are aggregated into one replica set,
- `tarantool_metrics_host` and `tarantool_metrics_port` are the HTTP server's listen hostname and port for integration with the SNMP-INT system,
- `tarantool_admin_host` and `tarantool_admin_port` are the listen hostname and port for accessing the Tarantool administrative console,
- `tarantool_host` and `tarantool_port` are the listen hostname and port for the Tarantool binary protocol,
- `tarantool_zones` is the list of availability zones (DCs),
- `tarantool_zone` is the availability zone (DC) for a specific server,
- `app.memtx_dir` is the path to the directory containing snapshots of the in-memory engine (memtx),
- `app.vinyl_dir` is the path to the directory containing data managed by the disk engine (vinyl),
- `app.wal_dir` is the path to the write-ahead log (WAL) files.

Note: To achieve optimal performance, the `app.wal_dir` directory should be located on a separate physical hard disk drive.

It is definitely not recommended to place `app.memtx_dir/app.vinyl_dir` and `app.wal_dir` on the same physical hard disk drive and/or store `app.wal_dir` using a storage system or a network file system. Otherwise, you may encounter an increase in response time during request processing.

Note: `tarantool_host` should point to the interface address and cannot be `0.0.0.0`, `127.0.0.1` or `::1`.

Note: UUID values in the `tarantool_uuid` and `tarantool_replicaset` fields can contain an arbitrary UUID generated by the `uuidgen` system utility.

When specifying the cluster topology, the following parameters should be taken into account in the first place:

- `ansible_host` specifies the name of the host on which the Tarantool instance should be installed,
- `tarantool_replicaset` specifies the replica set for the Tarantool node,
- `tarantool_zone` specifies if the node belongs to a specific availability zone (DC).

For example, to create a cluster of two replica sets with 3 servers in each replica set, you should specify 6 servers in the `tarantool_storage` section, 3 of which have `tarantool_replicaset = X`, and the remaining 3 servers have `tarantool_replicaset = Y`, where X and Y are arbitrary UUID identifiers generated by the `uuidgen` system utility. The *auto installer* then generates all necessary settings **automatically** (see the next section).

When setting up the `inventory`, you should also make sure that the port numbers used on the same server are unique.

1.4.3 Launching the auto installer

Use the following script to launch the `auto installer`:

```
$ ./tarantool-deploy.sh:
$ ./tarantool-deploy.sh -u <USER_NAME> -c <command>
```

`USER_NAME` is the account of the administrator who installs the product.

Available commands:

- `install` automatically installs the package,
- `update` updates the package,
- `configure` updates the configuration of applications,
- `start` launches the applications,
- `stop` stops the applications,
- `restart` restarts the applications.

1.4.4 Deploying the cluster

If the `auto installer` completes successfully, the specified computers become nodes of the Tarantool DBMS cluster in accordance with the topology specified in the `inventory`. After installation, you should execute the `ps axuf` command to make sure that there is an expected number of *running* Tarantool nodes on the specified computers. If there are fewer Tarantool nodes than you expected, carefully study the errors listed in the output of the installer, as well as the contents of the Tarantool DBMS logs in the `/data/logs/` directory (`*.init.log` files).

Typically, problems on node startup are associated with configuration errors, for example, duplicate port numbers or errors in the specified host names. In such a case, you should carefully re-verify the configuration and re-install the cluster.

1.4.5 Initializing the cluster

To run the cluster, you should perform an initial setup of the ZooKeeper as well as an initial distribution of the buckets. For this purpose, the `auto installer` **automatically** generates the `configure_zk.sh` script in the root directory on the administration server. After that, the script is **automatically** executed after the start of the Tarantool cluster. The script:

1. Creates a configuration of accessibility zones (Moscow, Siberia, etc.).
2. Registers all routers and storages.
3. Creates replica sets according to the configuration specified in the `inventory`.
4. Applies (distributes) the configuration.
5. Distributes the buckets in the cluster.

The script is generated each time the `auto installer` is launched. The installer runs this script **automatically** using a special Ansible role (`tarantool_configure`). The script can be modified directly if you need to specify additional parameters for the orchestor's API (see *Appendix 1*). Execution of the script is an idempotent operation that can be performed an arbitrary number of times. When manually executing this script, you should make sure that all commands were executed successfully:

```
{"error":{"message":"ok","code":0},"data":{},"status":true}
```

On successful completion of the script, all the Tarantool nodes that were started enter the `running` state in 5-10 seconds. After that, they start creating `*.main.log` and `*.audit.log` files in the `/data/logs/tarantool/` directory. If the nodes do not enter the running state, you should review the contents of the `*.init.log` files.

Within 10 seconds after successful configuration of ZooKeeper, the `configure_zk.sh` script starts the initial allocation of buckets between all configured replica sets. The `routers` may need 1-2 minutes to get information about the buckets. As new configurations are discovered, the `unknown_buckets` metric value gradually decreases in the monitoring system until it reaches zero. You can control the initial distribution process directly in the administrative console of the router by executing the `vshard.router.info()` command:

```
HOST:PORT:3200> vshard.router.info()
...
bucket:
    unreachable: 0 // allocated, but unavailable slots
    available_ro: 0  // allocated slots
    unknown: 0 // unallocated slots
    available_rw: 3000 // allocated slots
status: 0
alerts: []
...
```

For more information on the output of the `vshard.router.info()` command, see the section *Monitoring the shards*.

The cluster is ready to start when the number of unallocated slots reaches zero.

1.5 Manual installation

Tarantool Enterprise DBMS is delivered as a portable archive for RHEL 7. To run the Tarantool interpreter, you should unpack the `bin-tarantool-app.tar.gz` archive from the `artifacts/` directory of the `auto installer` and launch the `./tarantool` application:

```
./tarantool
Tarantool Enterprise 1.7.7-225-gd4f3087
type 'help' for interactive help
tarantool>
```

For more information on the Tarantool DBMS, see the Tarantool manual.

CHAPTER 2

Administration

2.1 Managing the sharding system

2.1.1 Managing nodes via CLI

Each Tarantool node (`router`/`storage`) provides a Command Line Interface (CLI) for debugging, monitoring and troubleshooting. The CLI acts as a Lua interpreter and displays the result in the human-readable YAML format. The `tarantoolctl` connect command, as well as `telnet` or `nc` (`netcat`) utilities can be used as a client:

```
$ /data/tarantool/router_1/tarantoolctl connect 127.0.0.1:3200
$ telnet 127.0.0.1 3200
$ echo "vshard.router.info()" | nc 127.0.0.1 3200
```

To specify the host name and the port, take `tarantool_admin_host` and `tarantool_admin_port` of one of the routers from the `auto installer`'s `inventory`.

Note: The `tarantoolctl` utility is installed together with any of the Tarantool DBMS cluster applications (`router`/`storage`).

2.1.2 Controlling the cluster via API

To control the cluster, use the `orchestrator` included in the delivery package. The `orchestrator` uses ZooKeeper to store and distribute the configuration. The `orchestrator` provides the REST API for controlling the cluster. Configurations in the ZooKeeper are changed as a result of calling the `orchestrator`'s API-functions, which in turn leads to changes in configurations of the Tarantool nodes.

We recommend using a **curl** command line interface to call the API-functions of the `orchestrator`.

The following example shows how to register a new availability zone (DC):

```
$ curl -X POST http://HOST:PORT/api/v1/zone \
    -d '{
  "name": "Caucasian Boulevard"
  }'
```

To check whether the DC registration was successful, try the following instruction. It retrieves the list of all registered nodes in the JSON format:

```
$ curl http://HOST:PORT/api/v1/zone| python -m json.tool
```

`HOST:PORT` in the URL should be equal to the `orchestrator` hostname and port, according to the inventory configuration of the `auto installer`.

To apply the new configuration directly on the Tarantool nodes, increase the configuration version number after calling the API function. To do this, use the POST request to `/api/v1/version`:

```
$ curl -X POST http://HOST:PORT/api/v1/version
```

Altogether, to update the cluster configuration:

1. **Call the `POST/PUT` method of the `orchestrator`.** As a result, the ZooKeeper nodes are updated, and a subsequent update of the Tarantool nodes is initiated.
2. **Update the configuration version using the `POST` request to `/api/v1/version`.** As a result, the configuration is applied to the Tarantool nodes.

See *Appendix 1* for the detailed orchestrator API.

2.2 Changing the cluster topology

The main problem with cluster scaling is that the identifiers of its components (replica sets and Tarantool instances) are unknown. If you connect/disconnect the replica set in parts (one instance after another), this can cause data loss.

An optimal strategy for deploying a cluster and connecting new nodes to it is to independently deploy replica sets, and then add a fully functional replica set to the cluster configuration and distribute it to all cluster members. In this case, all the parameters needed to update the configuration become known before actually adding new members to the cluster. After the new replica set has been fully deployed to the cluster, and all nodes of the cluster have been notified of the changes by updating the configuration, buckets start to migrate to the new node.

The most convenient way to add both new `router` nodes and replica sets is to use the `auto installer`.

2.3 Using the auto installer to install new nodes

Installing a new node is similar to the *general cluster installation* procedure executed by the `auto installer`.

To add a new node to the `auto installer`'s `inventory`, copy the node configuration of one of the replicas of the required replicasets and update the following parameters:

- `tarantool_host`
- `tarantool_port`
- `tarantool_metrics_host`
- `tarantool_metrics_port`
- `tarantool_admin_host`

- tarantool_admin_port
- tarantool_uuid

tarantool_uuid should be globally unique for all Tarantool nodes. UUID can be generated using the **uuidgen** system utility that is available in the installation packages of RHEL 7 and other versions.

After updating the inventory, you should run the auto installer to install and launch the new node. If the installation is successful, the nodes are launched and they wait for a configuration from the ZooKeeper (the waiting for zookeeper configuration status in the list of processes). After that, the auto installer executes the configure_zk.sh script to add the node to the ZooKeeper and switch the node(s) to normal operation (the running status). For more information, please see the section on *automatic installation*.

Note: If you do not want the active nodes of the Tarantool cluster to be reinstalled after the auto installer starts, it is recommended that you use the --limit <new_node_name> option for Ansible. Otherwise, the entire cluster can be restarted as a result of running the auto installer.

The following example shows how to run the auto installer command to add the shard3_* group of nodes:

```
$ ./[g][h]-tarantool-deploy.sh -u centos -c install -e '--limit shard3_*,localhost'
```

Note: To make the auto installer automatically call the ZooKeeper configuration script, you should always add localhost to the end of the host list in the --limit parameter.

2.4 Manually registering nodes in ZooKeeper

If the auto installer failed to automatically add a node to the ZooKeeper configuration, then this node can be registered manually using the orchestrator's API.

The general procedure is as follows:

1. Install and run a node or multiple nodes using the auto installer.
2. Register new nodes in the orchestrator one by one.
3. Register the router in the list of available routers (for routers); or register the node in the replica set (for storages).
4. Apply the updated configuration using the POST request to /api/v1/version.

2.4.1 Example: Registering a node

```
$ curl -X POST \
    http://HOST:PORT/api/v1/registry/node \
    -H 'cache-control: no-cache' \
    -H 'content-type: application/json' \
    -d {
            "zone_id": "<zone_id>",
            "uuid": <node_UUID>,
            "name": "router_3",
            "uri": "sh3.i.tarantool.org:3300",
            "user": "storage:storage",
```

(continues on next page)

(continued from previous page)

```
            "repl_user": "storage:storage",
            "cfg": {
                "listen": "sh3.i.tarantool.org:3300",
                "memtx_memory": 134217728
                }
        }
```

When calling an API function, configuration parameters should be replaced with the corresponding values from `inventory`. It is recommended that you use the auto installer-generated `configure_zk.sh` file as the most relevant example showing how to call the API functions and perform registration.

Use the following command to verify that the nodes are registered successfully:

```
$ curl http://HOST:PORT/api/v1/nodes|python -m json.tool
```

It retrieves a complete list of the Tarantool nodes.

2.4.2 Example: Registering a router

```
$ curl -s -X POST \
  http://HOST:PORT/api/v1/routers \
  -H 'cache-control: no-cache' \
  -H 'content-type: application/json' \
  -d '{
        "uuid": <router_UUID>
   }'
echo
```

2.4.3 Example: Registering a replica set

```
$ curl -s -X POST \
    http://HOST:PORT/api/v1/replicaset \
    -H 'cache-control: no-cache' \
    -H 'content-type: application/json' \
    -d '{
        "weight": 100,
        "replicas": [
            {
            "uuid": <first_node_UUID>,
            "master": true
            },
            {
            "uuid": <second_node_UUID>,
            "master": false
            }
        ]
    }'
```

As with individual nodes, you can verify that the replica set is registered successfully using the following `GET` request:

```
$  curl -s -X GET http://HOST:PORT/api/v1/replicaset|python -m json.tool
```

After registration, execute the following command to update the configuration version:

```
$ curl s -X POST http://HOST:PORT/api/v1/version
```

Note: Registering nodes manually is not trivial. If you still need to perform this operation, it is recommended to run the `auto installer` to get the current version of the `configure_zk.sh` file, after which you can use this file as the basis for API calls.

For more information about which parameters to use in API calls, please see the *orchestrator API* section.

2.5 Rebalancing the data

Rebalancing (resharding) is initiated automatically once a new replica set is added and registered in the `orchestrator`.

The most convenient way to trace through the process of rebalancing is to monitor the number of active slots on the storage nodes. Initially, a newly added replica set has 0 active buckets. After a few minutes, the background rebalancing process begins to transfer slots from the other replica sets to the new one. Rebalancing continues until the data is distributed evenly among all replica sets.

To get the current number of buckets, say `vshard.storage.info()` in the administrator console of the storage nodes:

```
$ echo "vshard.storage.info().bucket|nc -N HOST:STORAGE_PORT
127.0.0.1:3201> vshard.storage.info().bucket
---
- receiving: 0
  active: 1000
  total: 1000
  garbage: 0
  sending: 0
...
```

For more information on the monitoring parameters, please see the section *Monitoring the storages*.

2.6 Regular way of switching to another master

To perform a scheduled downtime of the master from any replica set, update the configuration of the replica set by using the appropriate orchestrator API call.

The general algorithm is the following:

1. Get the UUID of the replica set and the UUID of the new master node:
   - from the `inventory` file of the `auto installer`, or
   - by calling the API function of the `orchestrator`.
2. Change the UUID of the replica set master by calling the API function of the `orchestrator`.

2.6.1 Getting UUIDs from the inventory

Check the following parameters:

- UUID of the replica set is stored in the `tarantool_replicaset` parameter

- UUID of the node is stored in the `tarantool_uuid` parameter

2.6.2 Getting UUIDs using the orchestrator

To get of the replica set and the node using the orchestrator, say this:

```
$ curl -s -X GET http://HOST:PORT/api/v1/replicaset|python -m json.tool
curl -s -X GET http://95.163.251.213:8080/api/v1/replicaset|python -m json.tool
{
    "data": {
        "1a3c88d4-a25c-4c4d-8d4f-6d224f5b4900": {
            "0808ef7c-49b8-4c2a-b46b-4e81334b4ebb": {
                "hostname": "host_name",
                "master": false,
                "name": "shard1_2",
                "off": false,
                "repl_user": "storage:storage",
                "uri": "192.168.17.221:3301",
                "user": "user:user",
                "zone_id": 2
            },
    ...
}
```

For more information, please see the *replica set reference*.

2.6.3 Switching to another master

```
$ curl -s -X POST http://HOST:PORT/api/v1/replicaset/4e29028a-27d7-4125-b69e-
↪2b1fbd2c0fed/master \
        -H 'cache-control: no-cache' \
        -H 'content-type: application/json' \
        -d '{"instance_uuid": "1a3c88d4-a25c-4c4d-8d4f-6d224f5b4900 "}'
```

In this example, `4e29028a-27d7-4125-b69e-2b1fbd2c0fed` is the UUID of the replica set, and `1a3c88d4-a25c-4c4d-8d4f-6d224f5b4900` is the UUID of the new master.

To apply the configuration on the Tarantool nodes, update the version:

```
$ curl -s -X POST http://HOST:PORT/api/v1/version
```

Please see the *replica set reference*:

- For more information on how to use the `orchestrator`'s API in order to switch to another master.
- For more information on how to change the parameters of the replica set.

When you use the regular way to switch to another master using `orchestrator`, the Tarantool cluster automatically waits for the synchronization of all data that was written to the old master on all nodes in the replica set.

2.7 Regular way of deactivating the replica set

To perform a scheduled downtime of the replica set from a cluster:

1. **Migrate all the buckets to other storages in the cluster.** For migration, the replica set should be assigned a zero weight via the `orchestrator`'s API. After a few minutes, the `rebalancer` starts transferring buckets from the source replica set to other replica sets. Wait for the migration to finish.
2. Update the configuration on all nodes.
3. Disable the replica set: remove the replica set and deactivate the Tarantool nodes using the `orchestrator`'s API.

2.8 Failure of a single node or multiple nodes

If the master of any replica set fails, it is recommended that you do the following:

1. Switch one of the replicas to the master mode for all instances of this replicaset. This will allow the new master to process all requests.
2. Update the configuration on all cluster members. As a result, all connection requests to this replica set will be redirected to the new master.

In this case, you can use any external utility to monitor the state of the master and switch the modes of the instances.

2.9 Failover of a replica set

When an entire replica set goes down, the data becomes partially unavailable:

- read-only requests are redirected to replicas,
- write requests are not processed.

The `router` makes periodic attempts to reconnect to the master of the failed replica set. Thus, the functionality of the cluster is automatically restored when the failed replica set is restored.

2.10 Setting up geo redundancy

Logically, cluster nodes can belong to some availability zone. Physically, an availability zone is a separate DC, or a rack inside a DC. You can specify a matrix of weights (distances) for the availability zones.

New zones are added by calling a corresponding API method of the orchestrator. The `auto installer` generates a `configure_zk.sh` file that creates multiple default availability zones. If you need to add zones and/or change node associations, you can edit this file directly on the server.

By default, the matrix of weights (distances) for the zones is not configured, and geo-redundancy for such configurations works as follows:

- Data is always written to the master.
- If the master is available, then it is used for reading.
- If the master is unavailable, then any available replica is used for reading.

When you define a matrix of weights (distances) by calling `/api/v1/zones/weights`, the automatic scale-out system of the Tarantool DBMS finds a replica which is the closest to the specified router in terms of weights, and starts using this replica for reading. If this replica is not available, then the next nearest replica is selected, taking into account the distances specified in the configuration.

2.11 Resolving conflicts

Tarantool has an embedded mechanism for asynchronous replication. As a consequence, records are distributed among the replicas with a delay, so conflicts can arise.

To prevent conflicts, the special trigger `space.before_replace` is used. It is executed every time before making changes to the table for which it was configured. The trigger function is implemented in the Lua programming language. This function takes the original and new values of the tuple to be modified as its arguments. The returned value of the function is used to change the result of the operation: this will be the new value of the modified tuple.

For insert operations, the old value is absent, so `nil` is passed as the first argument.

For delete operations, the old value is absent, so `nil` is passed as the second argument. The trigger function can also return `nil`, thus turning this operation into delete.

This example shows how to use the `space.before_replace` trigger to prevent replication conflicts. Suppose we have a `box.space.test` table that is modified in multiple replicas at the same time. We store one payload field in this table. To ensure consistency, we also store the last modification time in each tuple of this table and set the `space.before_replace` trigger, which gives preference to newer tuples. Below is the code in Lua:

```lua
fiber = require('fiber')
-- define a function that will modify the function test_replace(tuple)
        -- add a timestamp to each tuple in the space
        tuple = box.tuple.new(tuple):update{{'!', 2, fiber.time()}}
        box.space.test:replace(tuple)
end
box.cfg{ } -- restore from the local directory
-- set the trigger to avoid conflicts
box.space.test:before_replace(function(old, new)
        if old ~= nil and new ~= nil and new[2] < old[2] then
                return old -- ignore the request
        end
        -- otherwise apply as is
end)
box.cfg{ replication = {...} } -- subscribe
```

2.12 Monitoring the shards

2.12.1 Monitoring the storages

Use `vshard.storage.info()` to obtain information on storage nodes.

Output example

```
vshard.storage.info()
---
- replicasets:
    <replicaset_2>:
    uuid: <replicaset_2>
    master:
        uri: storage:storage@127.0.0.1:3303
    <replicaset_1>:
    uuid: <replicaset_1>
```

(continues on next page)

(continued from previous page)

```
    master:
        uri: storage:storage@127.0.0.1:3301
  bucket: <!-- buckets status
    receiving: 0 <!-- buckets in the RECEIVING state
    active: 2 <!-- buckets in the ACTIVE state
    garbage: 0 <!-- buckets in the GARBAGE state (are to be deleted)
    total: 2 <!-- total number of buckets
    sending: 0 <!-- buckets in the SENDING state
  status: 1 <!-- the status of the replica set
  replication:
    status: disconnected <!-- the status of the replication
    idle: <idle>
  alerts:
  - ['MASTER_IS_UNREACHABLE', 'Master is unreachable: disconnected']
```

List of statuses

Code	Critical level	Description
0	Green	A replica set works in a regular way.
1	Yellow	There are some issues, but they don't affect a replica set efficiency (worth noticing, but don't require immediate intervention).
2	Orange	A replica set in in a degraded state.
3	Red	A replica set is disabled.

Potential issues

- `MISSING_MASTER` — No master node in the replica set configuration.

  Critical level: Orange.

  Cluster condition: Service is degraded for data-change requests to the replica set.

  Solution: Set the master node for the replica set in the configuration using API.

- `UNREACHABLE_MASTER` — No connection between the master and the replica.

  Critical level:

  – If idle value doesn't exceed T1 threshold (1 s.) — Yellow,

  – If idle value doesn't exceed T2 threshold (5 s.) — Orange,

  – If idle value exceeds T3 threshold (10 s.) — Red.

  Cluster condition: For read requests to replica, the data may be obsolete compared with the data on master.

  Solution: Reconnect to the master: fix the network issues, reset the current master, switch to another master.

- `LOW_REDUNDANCY` — Master has access to a single replica only.

  Critical level: Yellow.

  Cluster condition: The data storage redundancy factor is equal to 2. It is lower than the minimal recommended value for production usage.

  Solution: Check cluster configuration:

  - If only one master and one replica are specified in the configuration, it is recommended to add at least one more replica to reach the redundancy factor of 3.
  - If three or more replicas are specified in the configuration, consider checking the replicas' states and network connection among the replicas.

- `INVALID_REBALANCING` — Rebalancing invariant was violated. During migration, a storage node can either send or receive buckets. So it shouldn't be the case that a replica set sends buckets to one replica set and receives buckets from another replica set at the same time.

  Critical level: Yellow.

  Cluster condition: Rebalancing is on hold.

  Solution: There are two possible reasons for invariant violation:

  - The `rebalancer` has crashed.
  - Bucket states were changed manually.

  Either way, please contact Tarantool support.

- `HIGH_REPLICATION_LAG` — Replica's lag exceeds T1 threshold (1 sec.).

  Critical level:

  - If the lag doesn't exceed T1 threshold (1 sec.) — Yellow;
  - If the lag exceeds T2 threshold (5 sec.) — Orange.

  Cluster condition: For read-only requests to the replica, the data may be obsolete compared with the data on the master.

  Solution: Check the replication status of the replica. Further instructions are given in the :ref: troubleshooting guide <admin-troubleshoot>'.

- `OUT_OF_SYNC` — Mal-synchronization occured. The lag exceeds T3 threshold (10 sec.).

  Critical level: Red.

  Cluster condition: For read-only requests to the replica, the data may be obsolete compared with the data on the master.

  Solution: Check the replication status of the replica. Further instructions are given in the :ref: troubleshooting guide <admin-troubleshoot>'.

- `UNREACHABLE_REPLICA` — One or multiple replicas are unreachable.

  Critical level: Yellow.

  Cluster condition: Data storage redundancy factor for the given replica set is less than the configured factor. If the replica is next in the queue for rebalancing (in accordance with the weight configuration), the requests are forwarded to the replica that is still next in the queue.

  Solution: Check the error message and find out which replica is unreachable. If a replica is disabled, enable it. If this doesn't help, consider checking the network.

- `UNREACHABLE_REPLICASET` — All replicas except for the current one are unreachable. **Critical level:** Red.

  Cluster condition: The replica stores obsolete data.

  Solution: Check if the other replicas are enabled. If all replicas are enabled, consider checking network issues on the master. If the replicas are disabled, check them first: the master might be working properly.

2.12.2 Monitoring the router

Use `vshard.router.info()` to obtain information on the router.

Output example

```
vshard.router.info()
---
- replicasets:
    <replica set UUID>:
      master:
        status: <available / unreachable / missing>
        uri: <!-- URI of master
        uuid: <!-- UUID of instance
      replica:
        status: <available / unreachable / missing>
        uri: <!-- URI of replica used for slave requests
        uuid: <!-- UUID of instance
      uuid: <!-- UUID of replica set
    <replica set UUID>: ...
    ...
  status: <!-- status of router
  bucket:
    known: <!-- number of buckets with the known destination
    unknown: <!-- number of other buckets
  alerts: [<alert code>, <alert description>], ...
```

List of statuses

Code	Critical level	Description
0	Green	The `router` works in a regular way.
1	Yellow	Some replicas sre unreachable (affects the speed of executing read requests).
2	Orange	Service is degraded for changing data.
3	Red	Service is degraded for reading data.

Potential issues

Note: Depending on the nature of the issue, use either the UUID of a replica, or the UUID of a replica set.

- `MISSING_MASTER` — The master in one or multiple replica sets is not specified in the configuration.

  Critical level: Orange.

  Cluster condition: Partial degrade for data-change requests.

  Solution: Specify the master in the configuration.

- `UNREACHABLE_MASTER` — The `router` lost connection with the master of one or multiple replica sets.

  Critical level: Orange.

  Cluster condition: Partial degrade for data-change requests.

Solution: Restore connection with the master. First, check if the master is enabled. If it is, consider checking the network.

- `SUBOPTIMAL_REPLICA` — There is a replica for read-only requests, but this replica is not optimal according to the configured weights. This means that the optimal replica is unreachable.

  Critical level: Yellow.

  Cluster condition: Read-only requests are forwarded to a backup replica.

  Solution: Check the status of the optimal replica and its network connection.

- `UNREACHABLE_REPLICASET` — A replica set is unreachable for both read-only and data-change requests.

  Critical Level: Red.

  Cluster condition: Partial degrade for read-only and data-change requests.

  Solution: The replica set has an unreachable master and replica. Check the error message to detect this replica set. Then fix the issue in the same way as for *UNREACHABLE_REPLICA*.

2.13 Troubleshooting

Please see the Troubleshooting guide in the Tarantool manual.

2.14 Tarantool recovery

Please see the section Disaster recovery in the Tarantool manual.

2.15 Backups

Please see the section Backups in the Tarantool manual.

2.16 Reporting issues

To report a bug or submit an update request for this document, please create an issue in our repository at GitLab

CHAPTER 3

Appendixes

3.1 Appendix A. Orchestrator API reference

3.1.1 Configuring the zones

- *POST /api/v1/zone*
- *GET /api/v1/zone/*
- *PUT /api/v1/zone/*
- *DELETE /api/v1/zone/*

`POST /api/v1/zone`

Create a new zone.

Request

```
{
"name": "zone 1"
}
```

Response

```
{
"error": {
    "code": 0,
    "message": "ok"
},
"data": {
    "id": 2,
    "name": "zone 2"
},
"status": true
}
```

Potential errors

- `zone_exists` - the specified zone already exists

GET /api/v1/zone/{zone_id: optional}

Return information on the specified zone or on all the zones.

Response

```
{
    "error": {
        "code": 0,
        "message": "ok"
    },
    "data": [
        {
            "id": 1,
            "name": "zone 11"
        },
        {
            "id": 2,
            "name": "zone 2"
        }
    ],
    "status": true
}
```

Potential errors

- `zone_not_found` - the specified zone is not found

PUT /api/v1/zone/{zone_id}

Update information on the zone.

Body

```
{
    "name": "zone 22"
}
```

Response

```
{
    "error": {
        "code": 0,
        "message": "ok"
    },
    "data": {},
    "status": true
}
```

Potential errors

- `zone_not_found` - the specified zone is not found

DELETE /api/v1/zone/{zone_id}

Delete a zone if it doesn't store any nodes.

Response

```
{
    "error": {
        "code": 0,
        "message": "ok"
    },
    "data": {},
    "status": true
}
```

Potential errors

- `zone_not_found` - the specified zone is not found
- `zone_in_use` - the specified zone stores at least one node

3.1.2 Configuring the zone weights

- *GET /api/v1/zones/weights*
- *POST /api/v1/zones/weights*

`POST /api/v1/zones/weights`

Set the zone weights configuration.

Body

```
{
    "weights": {
        "1": {
            "2": 10,
            "3": 11
        },
        "2": {
            "1": 10,
            "3": 12
        },
        "3": {
            "1": 11,
            "2": 12
        }
    }
}
```

Response

```
{
    "error": {
        "code": 0,
        "message": "ok"
    },
    "data": {},
    "status": true
}
```

Potential errors

- `zones_weights_error` - configuration error

GET /api/v1/zones/weights

Return the zone weights configuration.

Response

```
{
    "error": {
        "code": 0,
        "message": "ok"
    },
    "data": {
        "1": {
            "2": 10,
            "3": 11
        },
        "2": {
            "1": 10,
            "3": 12
        },
        "3": {
            "1": 11,
            "2": 12
        }
    },
    "status": true
}
```

Potential errors

- `zone_not_found` - the specified zone is not found

3.1.3 Configuring registry

- *GET /api/v1/registry/nodes/new*
- *POST /api/v1/registry/node*
- *PUT /api/v1/registry/node/*
- *GET /api/v1/registry/node/*
- *DELETE /api/v1/registry/node/*

GET /api/v1/registry/nodes/new

Return all the detected nodes.

Response

```
{
    "error": {
        "code": 0,
        "message": "ok"
    },
    "data": [
        {
            "uuid": "uuid-2",
            "hostname": "tnt2.public.i",
            "name": "tnt2"
        }
```

(continues on next page)

(continued from previous page)

```
    ],
    "status": true
}
```

POST /api/v1/registry/node

Register the detected node.

Body

```
{
    "zone_id": 1,
    "uuid": "uuid-2",
    "uri": "tnt2.public.i:3301",
    "user": "user1:pass1",
    "repl_user": "repl_user1:repl_pass1",
    "cfg": {
        "listen": "0.0.0.0:3301"
    }
}
```

Response

```
{
    "error": {
        "code": 0,
        "message": "ok"
    },
    "data": {},
    "status": true
}
```

Potential errors

- `node_already_registered` - the specified node is already registred
- `zone_not_found` - the specified zone is not found
- `node_not_discovered` - the specified node is not detected

PUT /api/v1/registry/node/{node_uuid}

Update the registered node parameters.

Body

Pass only those parameters that need to be updated.

```
{
    "zone_id": 1,
    "repl_user": "repl_user2:repl_pass2",
    "cfg": {
        "listen": "0.0.0.0:3301",
        "memtx_memory": 100000
    }
}
```

Response

```
{
    "error": {
```

(continues on next page)

(continued from previous page)

```
        "code": 0,
        "message": "ok"
    },
    "data": {},
    "status": true
}
```

Potential errors

- `node_not_registered` - the specified node is not registered

GET /api/v1/registry/node/{node_uuid: optional}

Return information on the nodes in a cluster. If `node_uuid` is passed, information on this node only is returned.

Response

```
{
    "error": {
        "code": 0,
        "message": "ok"
    },
    "data": {
        "uuid-1": {
            "user": "user1:pass1",
            "hostname": "tnt1.public.i",
            "repl_user": "repl_user2:repl_pass2",
            "uri": "tnt1.public.i:3301",
            "zone_id": 1,
            "name": "tnt1",
            "cfg": {
                "listen": "0.0.0.0:3301",
                "memtx_memory": 100000
            },
            "zone": 1
        },
        "uuid-2": {
            "user": "user1:pass1",
            "hostname": "tnt2.public.i",
            "name": "tnt2",
            "uri": "tnt2.public.i:3301",
            "repl_user": "repl_user1:repl_pass1",
            "cfg": {
                "listen": "0.0.0.0:3301"
            },
            "zone": 1
        }
    },
    "status": true
}
```

Potential errors

- `node_not_registered` - the specified node is not registered

DELETE /api/v1/registry/node/{node_uuid}

Delete the node if it doesn't belong to any replica set.

Response

```
{
    "error": {
        "code": 0,
        "message": "ok"
    },
    "data": {},
    "status": true
}
```

Potential errors

- `node_not_registered` - the specified node is not registered
- `node_in_use` - the specified node is in use by a replica set

3.1.4 Routers API

- *GET /api/v1/routers*
- *POST /api/v1/routers*
- *DELETE /api/v1/routers/{uuid}*

`GET /api/v1/routers`

Return the list of all nodes that constitute the router.

Response

```
{
    "data": [
        "uuid-1"
    ],
    "status": true,
    "error": {
        "code": 0,
        "message": "ok"
    }
}
```

`POST /api/v1/routers`

Assign the router role to the node.

Body

```
{
    "uuid": "uuid-1"
}
```

Response

```
{
    "error": {
        "code": 0,
        "message": "ok"
    },
    "data": {},
    "status": true
}
```

Potential errors

- `node_not_registered` - the specified node is not registred

`DELETE /api/v1/routers/{uuid}`

Release the router role from the node.

Response

```
{
    "error": {
        "code": 0,
        "message": "ok"
    },
    "data": {},
    "status": true
}
```

3.1.5 Configuring replica sets

- *POST /api/v1/replicaset*
- *PUT /api/v1/replicaset/*
- *GET /api/v1/replicaset/*
- *DELETE /api/v1/replicaset/*
- *POST /api/v1/replicaset/{replicaset_uuid}/master*
- *POST /api/v1/replicaset/{replicaset_uuid}/node*
- *DELETE /api/v1/zone/*

`POST /api/v1/replicaset`

Create a replica set containing all the registered nodes.

Body

```
{
    "uuid": "optional-uuid",
    "replicaset": [
        {
            "uuid": "uuid-1",
            "master": true
        }
    ]
}
```

Response

```
{
    "error": {
        "code": 0,
        "message": "ok"
    },
    "data": {
        "replicaset_uuid": "cc6568a2-63ca-413d-8e39-704b20adb7ae"
    },
    "status": true
}
```

Potential errors

- `replicaset_exists` – the specified replica set already exists
- `replicaset_empty` – the specified replica set doesn't contain any nodes
- `node_not_registered` – the specified node is not registered
- `node_in_use` – the specified node is in use by another replica set

PUT /api/v1/replicaset/{replicaset_uuid}
Update the replica set parameters.

Body

```
{
    "replicaset": [
        {
            "uuid": "uuid-1",
            "master": true
        },
        {
            "uuid": "uuid-2",
            "master": false,
            "off": true
        }
    ]
}
```

Response

```
{
    "error": {
        "code": 0,
        "message": "ok"
    },
    "data": {},
    "status": true
}
```

Potential errors

- `replicaset_empty` – the specified replica set doesn't contain any nodes
- `replicaset_not_found` – the specified replica set is not found
- `node_not_registered` – the specified node is not registered
- `node_in_use` – the specified node is in use by another replica set

GET /api/v1/replicaset/{replicaset_uuid: optional}
Return information on all the cluster components. If `replicaset_uuid` is passed, information on this replica set only is returned.

Body

```
{
    "name": "zone 22"
}
```

Response

```
{
    "error": {
        "code": 0,
        "message": "ok"
    },
    "data": {
        "cc6568a2-63ca-413d-8e39-704b20adb7ae": {
            "uuid-1": {
                "hostname": "tnt1.public.i",
                "off": false,
                "repl_user": "repl_user2:repl_pass2",
                "uri": "tnt1.public.i:3301",
                "master": true,
                "name": "tnt1",
                "user": "user1:pass1",
                "zone_id": 1,
                "zone": 1
            },
            "uuid-2": {
                "hostname": "tnt2.public.i",
                "off": true,
                "repl_user": "repl_user1:repl_pass1",
                "uri": "tnt2.public.i:3301",
                "master": false,
                "name": "tnt2",
                "user": "user1:pass1",
                "zone": 1
            }
        }
    },
    "status": true
}
```

Potential errors

- `replicaset_not_found` – the specified replica set is not found

`DELETE /api/v1/replicaset/{replicaset_uuid}`

Delete a zone if it doesn't store any nodes.

Response

```
{
    "error": {
        "code": 0,
        "message": "ok"
    },
    "data": {},
    "status": true
}
```

Potential errors

- `replicaset_not_found` - the specified replica set is not found

`POST /api/v1/replicaset/{replicaset_uuid}/master`

Switch the master in the replica set.

Body

```
{
    "instance_uuid": "uuid-1",
    "hostname_name": "hostname:instance_name"
}
```

Response

```
{
    "error": {
        "code": 0,
        "message": "ok"
    },
    "data": {},
    "status": true
}
```

Potential errors

- `replicaset_not_found` – the specified replica set is not found
- `node_not_registered` – the specified node is not registered
- `node_not_in_replicaset` – the specified node is not in the specified replica set

`POST /api/v1/replicaset/{replicaset_uuid}/node`

Add a node to the replica set.

Response

```
{
    "error": {
        "code": 0,
        "message": "ok"
    },
    "data": {},
    "status": true
}
```

Body

```
{
    "instance_uuid": "uuid-1",
    "hostname_name": "hostname:instance_name",
    "master": false,
    "off": false
}
```

Potential errors

- `replicaset_not_found` – the specified replica set is not found
- `node_not_registered` – the specified node is not registered
- `node_in_use` – the specified node is in use by another replica set

`GET /api/v1/replicaset/status`

Return statistics on the cluster.

Response

```
{
    "error": {
        "code": 0,
        "message": "ok"
    },
    "data": {
        "cluster": {
            "routers": [
                {
                    "zone": 1,
                    "name": "tnt1",
                    "repl_user": "repl_user1:repl_pass1",
                    "hostname": "tnt1.public.i",
                    "status": null,
                    "uri": "tnt1.public.i:3301",
                    "user": "user1:pass1",
                    "uuid": "uuid-1",
                    "total_rps": null
                }
            ],
            "storages": [
                {
                    "hostname": "tnt1.public.i",
                    "repl_user": "repl_user2:repl_pass2",
                    "uri": "tnt1.public.i:3301",
                    "name": "tnt1",
                    "total_rps": null,
                    "status": 'online',
                    "replicas": [
                        {
                            "user": "user1:pass1",
                            "hostname": "tnt2.public.i",
                            "replication_info": null,
                            "repl_user": "repl_user1:repl_pass1",
                            "uri": "tnt2.public.i:3301",
                            "uuid": "uuid-2",
                            "status": 'online',
                            "name": "tnt2",
                            "total_rps": null,
                            "zone": 1
                        }
                    ],
                    "user": "user1:pass1",
                    "zone_id": 1,
                    "uuid": "uuid-1",
                    "replicaset_uuid": "cc6568a2-63ca-413d-8e39-704b20adb7ae",
                    "zone": 1
                }
            ]
        }
    },
    "status": true
}
```

Potential errors

- `zone_not_found` - the specified zone is not found
- `zone_in_use` - the specified zone stores at least one node

3.1.6 Setting up configuration versions

- *POST /api/v1/version*
- *GET /api/v1/version*

POST /api/v1/version

Set the zone weights configuration.

Response

```
{
    "error": {
        "code": 0,
        "message": "ok"
    },
    "status": true,
    "data": {
        "version": 2
    }
}
```

Potential errors

- `cfg_error` - configuration error

GET /api/v1/version

Return the zone weights configuration.

Response

```
{
    "error": {
        "code": 0,
        "message": "ok"
    },
    "status": true,
    "data": {
        "version": 2
    }
}
```

3.1.7 Configuring sharding

- *POST /api/v1/sharding/cfg*
- *GET /api/v1/sharding/cfg*

POST /api/v1/sharding/cfg

Add a new sharding configuration.

Response

```
{
    "error": {
        "code": 0,
        "message": "ok"
    },
    "status": true,
```

(continues on next page)

(continued from previous page)

```
    "data": {}
}
```

GET /api/v1/sharding/cfg

Return the current sharding configuration.

Response

```
{
    "error": {
        "code": 0,
        "message": "ok"
    },
    "status": true,
    "data": {}
}
```

3.1.8 Resetting cluster configuration

- *POST /api/v1/clean/cfg*
- *POST /api/v1/clean/all*

POST /api/v1/clean/cfg

Reset the cluster configuration.

Response

```
{
    "error": {
        "code": 0,
        "message": "ok"
    },
    "status": true,
    "data": {}
}
```

POST /api/v1/clean/all

Reset the cluster configuration and delete information on the cluster nodes from the ZooKeeper catalogues.

Response

```
{
    "error": {
        "code": 0,
        "message": "ok"
    },
    "status": true,
    "data": {}
}
```

3.2 Appendix B. Structure of ZooKeeper directories

Note: The cluster default name is `default`.

- */tarantool/cluster/{cluster_name}/new_nodes/{host_name:instance_name}*
- */tarantool/cluster/{cluster_name}/node/{uuid}*
- */tarantool/cluster/{cluster_name}/cfg*

`/tarantool/cluster/{cluster_name}/new_nodes/{host_name:instance_name}`

The directory contains information on the Tarantool instance. This information is obtained once the instance is detected.

Example

```
{
    "hostname": host_name,
    "name": instance_name,
    "uuid": "uuid-1"
}
```

`/tarantool/cluster/{cluster_name}/node/{uuid}`

The directory contains the Tarantool instance status.

Example

```
"status": ['online'/'resharding'/'offline']
```

`/tarantool/cluster/{cluster_name}/cfg`

The directory contains the configuration of the entire cluster.

Example

```
{
    "zones": {
        1:  {
          "id": 1,
          "name": 'DC 1',
        }
    },
    "servers": {
        INSTANCE1_UUID: {
            "zone": 1,
            "hostname": "tnt1.a.public.i",  # from new_nodes
            "name": "tnt1",  # from new_nodes
            "uri": "tnt1.a.public.i:3301",
            "user": "user1:pass1",
            "repl_user": "repl_user1:repl_pass1",
            "cfg": {
              "listen": "0.0.0.0:3301",
              # ...other box.cfg{} parameters
            }
        },
        INSTANCE2_UUID: { ... },
        ...
    },

    "cluster": {
        "cfg": {
```

(continues on next page)

(continued from previous page)

```
                "vbuckets": 1000,
                # sharding config parameters
            },

            "routers": [
                INSTANCE1_UUID,
                ...
            ],

            "storage": {  # replica sets
                REPLICASET1_UUID: {  # replica set 1
                    INSTANCE1_UUID: {
                        "master": true
                    },
                    INSTANCE2_UUID: {
                        "master": false
                    },
                },

                {  # replica set 2
                  ...
                }
            }
        }
}
```

3.3 Appendix C. Useful Tarantool parameters

- `box.info`
- `box.info.replication`
- `box.info.memory`
- `box.stat`
- `box.stat.net`
- `box.slab.info`
- `box.slab.stats`

For details, please see reference on module 'box' in the main Tarantool documentation.

3.4 Appendix D. Monitoring system metrics

Option	Description	SNMP type	Units of measure	Threshold
Version	Tarantool version	DisplayString		
IsAlive	instance availability indicator	Integer (listing)		0 - unavailable 1 - available
MemoryLua	storage space used by Lua	Gauge32	Mbyte	900

Continued on next page

Table 1 – continued from previous page

Option	Description	SNMP type	Units of measure	Threshold
MemoryData	storage space used for storing data	Gauge32	Mbyte	set the value manually
MemoryNet	storage space used for network I/O	Gauge32	Mbyte	1024
MemoryIndex	storage space used for storing indexes	Gauge32	Mbyte	set the value manually
MemoryCache	storage space used for storing caches (for vinyl engine only)	Gauge32	Mbyte	
ReplicationLag	lag time since the last sync between (the maximum value in case there are multiple fibers)	Integer32	sec.	5
FiberCount	number of fibers	Gauge32	pc.	1000
CurrentTime	current time, in seconds, starting at January, 1st, 1970	Unsigned32	Unix timestamp, in sec.	
StorageStatus	status of a replica set	Integer	listing	> 1
StorageAlerts	number of alerts for storage nodes	Gauge32	pc.	>= 1
StorageTotalBkts	total number of buckets in the storage	Gauge32	pc.	< 0
StorageActiveBkts	number of buckets in the ACTIVE state	Gauge32	pc.	< 0
StorageGarbageBkts	number of buckets in the GARBAGE state	Gauge32	pc.	< 0
StorageReceivingBkts	number of buckets in the RECEIVING state	Gauge32	pc.	< 0
StorageSendingBkts	number of buckets in the SENDING state	Gauge32	pc.	< 0
RouterStatus	status of the `router`	Integer	listing	> 1
RouterAlerts	number of alerts for the router	Gauge32	pc.	>= 1
RouterKnownBkts	number of buckets within the known destination replica sets	Gauge32	pc.	< 0
RouterUnknownBkts	number of buckets that are unknown to the `router`	Gauge32	pc.	< 0
RequestCount	total number of requests	Counter64	pc.	

Continued on next page

Table 1 – continued from previous page

Option	Description	SNMP type	Units of measure	Threshold
InsertCount	total number of insert requests	Counter64	pc.	
DeleteCount	total number of delete requests	Counter64	pc.	
ReplaceCount	total number of replace requests	Counter64	pc.	
UpdateCount	total number of update requests	Counter64	pc.	
SelectCount	total number of select requests	Counter64	pc.	
EvalCount	number of calls made via Eval	Counter64	pc.	
CallCount	number of calls made via `call`	Counter64	pc.	
ErrorCount	number of errors in Tarantool	Counter64	pc.	
AuthCount	number of completed authentication operations	Counter64	pc.	

3.5 Appendix E. Audit log

Audit log provides records on the Tarantool DBMS events in the JSON-format. The following event logs are available:

- successful/failed user authentication and authorization,
- closed connection,
- password change,
- creation/deletion of a user/role,
- enabling/disabling a user,
- changing privileges of a user/role.

3.5.1 Log structure

Key	Type	Description	Example
`type`	string	type of event	`<"access_denied">`
`type_id`	number	id of event	`<8>`
`description`	string	description of event	`<"Authentication␣ ↪failed">`
`time`	string	time of event	`<"YYYY-MM- ↪DDTHH:MM:SS. ↪03f[+|-]GMT">`
`peer`	string	remote client	`<"ip:port">`
`user`	string	user	`<"user">`
`param`	string	parameters of event	see below

3.5.2 Events description

Event	Key	Parameters
user authorized successfully	`auth_ok`	`{"name": "user"}`
user authorization failed	`auth_fail`	`{"name": "user"}`
user logged out or quit the session	`disconnect`	
failed access attempts to secure data (personal records, details, geolocation, etc.)	`access_denied`	`{"name": "obj_name", "obj_type": "space", "access_type": "read"}`
creating a user	`user_create`	`{"name": "user"}`
dropping a user	`user_drop`	`{"name": "user"}`
disabling a user	`user_disable`	`{"name": "user"}`
enabling a user	`user_enable`	`{"name": "user"}`
granting (changing) privileges (roles, profiles, etc.) for the user	`user_priv`	`{"name": "user"} "obj_name": "obj_name", "obj_type": "space", "old_priv": "", "new_priv": "read,write"}`
resetting password of the user (the user making changes should be specified)	`password_change`	`{"name": "user"}`
creating a role	`role_create`	`{"name": "role"}`
granting (changing) privileges for the role	`role_priv`	`{"name": "role"} "obj_name": "obj_name", "obj_type": "space", "old_priv": "", "new_priv": "read,write"}`