Storage cluster management with LINSTOR

Digital Warehouse

LINSTOR is a toolkit for automated cluster management that takes the complexity out of DRBD management and offers a wide range of functions, including provisioning and snapshots. By Robert Altnoeder

LINSTOR open source software manages block storage in large Linux clusters and simplifies the deployment of high availability with distributed replicated block device version 9 (DRBD 9), dynamically provisioning storage and simplifying storage management for Kubernetes, Docker, OpenStack, OpenNebula, and OpenShift through integration. Alternatively, the software can take care of high availability with DRBD exclusively. In this article, I look into the setup and operation of the tool.

At first glance, storage automation appears to be a simple problem: It's been a long while since the tasks of creating, enlarging, and deleting storage volumes were subject to the restrictions imposed by the size of individual storage devices or the size and position of the slices or partitions created on them. Technologies such as the Logical Volume Manager (LVM) or the Zetabyte filesystem (ZFS) have long made it possible to manage storage pools in which almost any number and any size of storage volume can be created or deleted with equal ease.

At first glance, it might seem easy enough to automate these manual tasks with a few scripts. However, automation requires attention to tiny details and holds many potential pitfalls for the admin, as well. What is not noticeable in manual administration can quickly lead to problems in automation – for example, when device files do not yet exist in the `/dev` filesystem, even though the respective command for creating a volume has already reported completion, or if you want to delete volumes that are no longer mounted but are in use by the udev subsystem. If the requirement to manage an entire cluster centrally, and not just a single system, is added, it is clear that a few rough-and-ready scripts are not up to the task.

Three Components

In these cases, LINSTOR [1] from LINBIT enters into play. It is an open source software bundle comprising several components for storage cluster management. It supports not only the creation, scaling, and deletion of individual simple volumes on one of the storage cluster nodes, but also the replication of individual volumes or the replication of consistency groups of several volumes between several cluster nodes. This process is handled by DRBD replication technology. You can also use LINSTOR to configure additional functionality for volumes, such as encryption or data deduplication. Especially with more complex storage technologies like DRBD, the software automatically manages various system resources, including the necessary TCP/IP port numbers for the replication links or the "minor numbers" for the device files on the `/dev` filesystem. Calculating the storage space required for the DRBD metadata is also automated.

LINSTOR has essentially three components:

1. The LINSTOR controller manages the configuration of the entire cluster and must be executable on at least one node of the cluster. For reasons of high availability, the controller is normally installed on several nodes.
2. The LINSTOR satellite runs on each node of the storage cluster, where it performs the steps required to manage the storage space automatically.
3. LINSTOR clients complete the package. On the one hand, this could simply be the command-line application, known as the LINSTOR

client, which can be used to operate LINSTOR both manually and by scripting. On the other hand, it could be a driver that provides integration with the storage management of other products, such as virtualization environments like OpenStack, OpenNebula, or Proxmox, or container technologies like Kubernetes.

All of these components are network transparent, which means, for example, that a client does not run on the same node as the controller with which that client communicates.

Installation

To install LINSTOR, you need at least version 8 of the Java Runtime Environment (JRE), which is normally installed as a dependency when installing the distribution packages. Whether this is OpenJDK, Oracle HotSpot VM, or IBM Java VM is not important.

The package is usually installed in the directory trees for the respective type of file or directory, so that configuration data ends up in `/etc`, variable data like logs and reports in `/var/log`, and program libraries in `/usr/share/linstor-server/lib`. With command-line parameters or entries in the configuration file, LINSTOR also offers the option of installing the components in other paths.

By default, the LINSTOR controller relies on an integrated database, which is automatically initialized the first time the controller is started. For larger installations, you can also configure the controller to use a central SQL database, such as PostgreSQL, MariaDB, or DB2.

After installing from the distribution packages, start the modules installed on the respective node with systemd:

```
systemctl start linstor controller
systemctl start linstor satellite
```

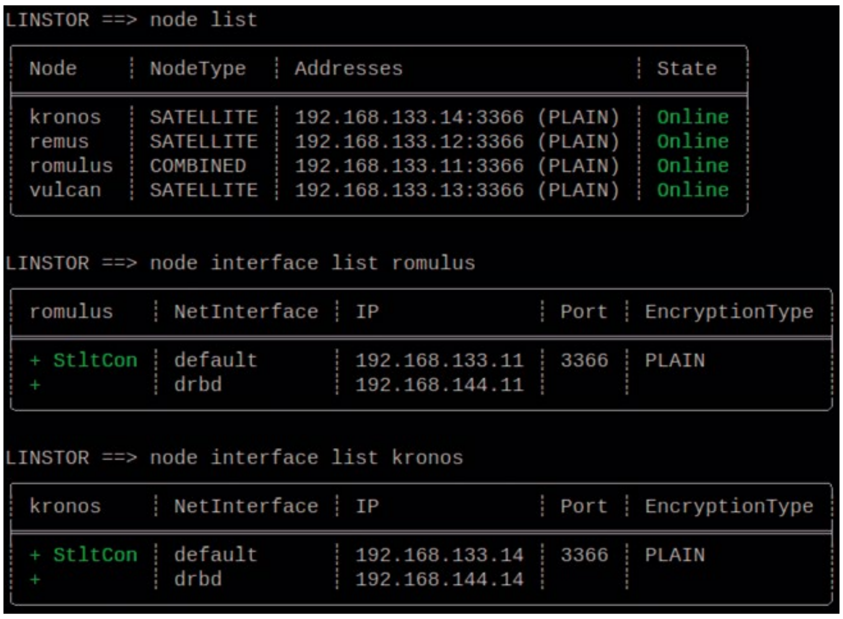
Here, I kept the default LINSTOR configuration settings for the network communication of the individual components, the resource pools to be used, TCP/IP port ranges, minor number ranges, DRBD peer slots, and the like. Also, I will not be using SSL-encrypted connections, because the setup steps required to customize all of the options would go beyond the scope of this article.

Configuring the Cluster

Before you start creating your cluster configuration and storage pools in LINSTOR, it is important to understand some basic concepts, including the various objects and the logical dependencies between them. The most important of these objects are the node, network interface, storage pool definition and storage pool, resource definition and resource, and volume definition and volume. The respective definition objects each manage information that is identical for all objects of the respective definition category throughout the cluster (i.e., on all cluster nodes). For example, a volume definition contains information that is identical on all cluster nodes, such as the size of a volume replicated across multiple cluster nodes. As a result of this principle, there are dependencies between these objects. To be able to create an initial volume, you first need to create the required objects in a sequence that reflects the dependencies. You can use the LINSTOR client to manage the controller manually for the initial configuration. The client requires the hostname or IP address of the cluster nodes on which a controller can run to be able to communicate with the active controller. Set the `LS_CONTROLLERS` environment variable before you start the LINSTOR client. If you do not specify any further parameters, the client launches in interactive mode:

```
export LS_CONTROLLERS=192.168.133.11
linstor
```

Typically, you will start the configuration work with the node objects that represent the individual cluster nodes of the storage cluster. When you create a node on the controller, you define the IP address (and the port, if necessary) that the controller uses to communicate with the satellite component on the respective cluster node. You can also define – especially for more clarity in the cluster – whether the cluster node runs a controller, a satellite, or both components. The name under which the cluster node is registered on the controller is, of course, also required. Ideally, this name should match the node name of the respective cluster node. You can define this name with the `uname -n` command. The domain name can be omitted if it is not absolutely necessary to differentiate between the individual nodes:



```
LINSTOR ==> node list
╭──────────────────────────────────────────────────────────────────╮
┊ Node    ┊ NodeType  ┊ Addresses                         ┊ State  ┊
╞══════════════════════════════════════════════════════════════════╡
┊ kronos  ┊ SATELLITE ┊ 192.168.133.14:3366 (PLAIN) ┊ Online ┊
┊ remus   ┊ SATELLITE ┊ 192.168.133.12:3366 (PLAIN) ┊ Online ┊
┊ romulus ┊ COMBINED  ┊ 192.168.133.11:3366 (PLAIN) ┊ Online ┊
┊ vulcan  ┊ SATELLITE ┊ 192.168.133.13:3366 (PLAIN) ┊ Online ┊
╰──────────────────────────────────────────────────────────────────╯

LINSTOR ==> node interface list romulus
╭─────────────────────────────────────────────────────────────────────╮
┊ romulus   ┊ NetInterface ┊ IP             ┊ Port ┊ EncryptionType ┊
╞═════════════════════════════════════════════════════════════════════╡
┊ + StltCon ┊ default      ┊ 192.168.133.11 ┊ 3366 ┊ PLAIN          ┊
┊ +         ┊ drbd         ┊ 192.168.144.11 ┊      ┊                ┊
╰─────────────────────────────────────────────────────────────────────╯

LINSTOR ==> node interface list kronos
╭─────────────────────────────────────────────────────────────────────╮
┊ kronos    ┊ NetInterface ┊ IP             ┊ Port ┊ EncryptionType ┊
╞═════════════════════════════════════════════════════════════════════╡
┊ + StltCon ┊ default      ┊ 192.168.133.14 ┊ 3366 ┊ PLAIN          ┊
┊ +         ┊ drbd         ┊ 192.168.144.14 ┊      ┊                ┊
╰─────────────────────────────────────────────────────────────────────╯
```

Figure 1: Overview of all created nodes and network interfaces.

Listing 1: LVM Logical Volumes

```
vulcan ~ # lvs drbdpool
  LV              VG       Attr       LSize   Pool     Origin Data%  Meta%  Move
  LocalData_00000 drbdpool Vwi-a-tz-- 152.00m thinpool        0.04
  thinpool        drbdpool twi-aotz-- 300.00m                 0.02          10.94
```

```
node create --node-type Combined ➘
  romulus 192.168.133.11
node create --node-type satellite ➘
  remus 192.168.133.12
node create --node-type satellite ➘
  vulcan 192.168.133.13
node create --node-type satellite ➘
  kronos 192.168.133.14
```

In addition to the required network interface, which you created when you created the node, you can now create further network interfaces (Figure 1). For example, if you want to use DRBD, you can use these additional network interfaces to distribute the replication links for different DRBD resources to these interfaces:

```
node interface create romulus ➘
  drbd 192.168.144.11
node interface create remus ➘
  drbd 192.168.144.12
node interface create vulcan ➘
  drbd 192.168.144.13
node interface create kronos ➘
  drbd 192.168.144.14
```

After you have registered all the nodes, move on to create the respective storage pools that are available on the nodes for automatic storage management. The storage pool definition is automatically created when the first storage pool is created. Nevertheless, it is important that you are aware that this storage pool definition exists, because deleting the last storage pool does not automatically delete the storage pool definition.

When you define a storage pool, you specify what type of pool it is (e.g., an *LVM Volume Group* or a *ZFS zpool*) and whether thick or thin provisioning is used. Depending on the type of pool, you specify the name of the volume group, the LVM thin pool, or the ZFS pool as a parameter for the LINSTOR storage pool driver. Of course, the storage pool object also has a unique name, which you can choose freely, observing the permitted characters and length restrictions.

The name *DfltStorPool* (Default Storage Pool) plays a special role, because the controller automatically selects this pool if you do not specify a storage pool in the various definition objects or in the resource or volume object when you create your storage resources later.

The command

```
storage-pool create lvmthin romulus ➘
  thinpool drbdpool/thinpool
```

creates a storage pool named *thinpool* with the LVM driver (Figure 2).

Working with LINSTOR

Now that at least a minimal LINSTOR configuration has been completed, you can define the initial resources and their volumes and generate them on one or more cluster nodes. It is often a good idea to start with a simple configuration first to check that all the components are working properly. Therefore, the first example only creates a single local LVM volume. As a first step, you need to create the definition for the resources whose volumes use only the storage layer and select *LocalData* as the name for the resource definition, and thus also for their resources:

```
resource-definition create ➘
  --layer-list storage LocalData
```

Now, add a single volume of 150MB to the resource definition named *LocalData*:

```
volume definition create LocalData 150m
```

Finally, on the cluster nodes *vulcan* and *kronos*, create a resource associated with these definitions with the *thinpool* storage pool for the volume of the respective resource:

```
resource create --storage-pool thinpool ➘
  vulcan kronos LocalData
```

If these steps complete without error, the list of LVM logical volumes on cluster nodes *vulcan* and *kronos* should now contain an entry for a volume named *LocalData_00000* (Listing 1). You can then format and mount this LVM logical volume in the usual way.

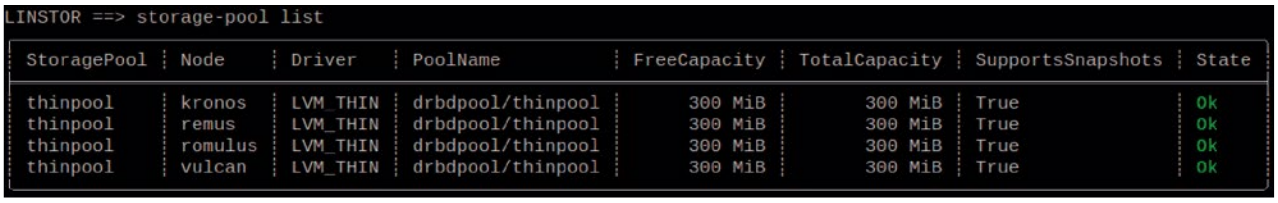
The next application example will be a bit more sophisticated: This time there will be a resource with a volume that replicates triple-redundantly on three of the cluster nodes with DRBD. When you create the resource definition, you therefore select the layers *drbd* and *storage* in the layer list, and the resource is given the name *SharedData* to match the replication of the volume:

```
resource-definition create ➘
  --layer-list drbd,storage SharedData
```

Again, add a single volume of 150MB to this resource:

```
volume definition create SharedData 150m
```

This time, leave the selection of the cluster nodes to the LINSTOR controller by specifying only



```
LINSTOR ==> storage-pool list
 | StoragePool | Node    | Driver   | PoolName          | FreeCapacity | TotalCapacity | SupportsSnapshots | State |
 |-------------------------------------------------------------------------------------------------------------------|
 | thinpool    | kronos  | LVM_THIN | drbdpool/thinpool |      300 MiB |       300 MiB | True              | Ok    |
 | thinpool    | remus   | LVM_THIN | drbdpool/thinpool |      300 MiB |       300 MiB | True              | Ok    |
 | thinpool    | romulus | LVM_THIN | drbdpool/thinpool |      300 MiB |       300 MiB | True              | Ok    |
 | thinpool    | vulcan  | LVM_THIN | drbdpool/thinpool |      300 MiB |       300 MiB | True              | Ok    |
```

Figure 2: List of created storage pools.

the required redundancy for the `--auto-place` option:

```
resource create --storage-pool thinpool ➘
                --auto-place 3 SharedData
```

The list of resources and volumes shows which nodes the controller has selected for the replicated DRBD resource and which resources are automatically available for it. In this case, the cluster nodes *kronos*, *remus*, and *romulus* were automatically selected to provide the required triple-redundancy of the volume replicated by DRBD. TCP/IP port 7000 was reserved for network communication by the DRBD resource, and the DRBD volume was assigned a minor number of *1000* (**Figure 3**). The volume can be found as the `drbd1000` entry in the `/dev` directory. The actual storage space for the data is again provided by an LVM logical volume, which appears in the output of the `lvs drbdpool` command as *SharedData_00000*.

Now that storage management is automated, retroactive modification of existing storage resources is very easy. For example, you can easily migrate one of the existing replicas to another cluster node by first adding a fourth replica and then removing one of the original replicas. If you wait for a DRBD resync, the triple-redundancy of the volume is never compromised.

As an example, migrate the *DRBD-Resource* replica between the cluster nodes *kronos* and *vulcan*. First add the replica to the *vulcan* node by assigning a LINSTOR resource to the node:

```
resource create -s thinpool vulcan ➘
  SharedData
```

After DRBD has completed the resync, you can delete the replica on *kronos*:

```
resource delete kronos SharedData
```

To remove a resource from all cluster nodes permanently, you can also delete the resource definition of this resource directly; that is, the resource is first deleted from all cluster nodes on which it was created:

```
resource definition delete LocalData
```

The resource definition, including the volume definitions it contains, is automatically deleted only after all cluster nodes have reported a successful cleanup to the controller.

Working with Snapshots

For volumes located in a storage pool based on thin provisioning (i.e., currently, storage pools with the *lvmthin* and *zfsthin* drivers), LINSTOR also provides cluster-wide snapshot functionality, not only for local storage volumes, but also for storage volumes replicated by DRBD.

To create snapshots of replicated volumes that are identical on all cluster nodes involved, LINSTOR stops I/O activity on the resource in question at the DRBD level. As a result, the data on the back-end storage volume used by DRBD does not change, and different cluster nodes can create the snapshot at different times. I/O is not released again at the DRBD level until the snapshot has been created on all cluster nodes.

Creating snapshots is similar to the process for resources. However, you do not have to take a snapshot on all cluster nodes on which the resource exists. Instead, you select the cluster nodes on which the snapshot will be created when you grab it:

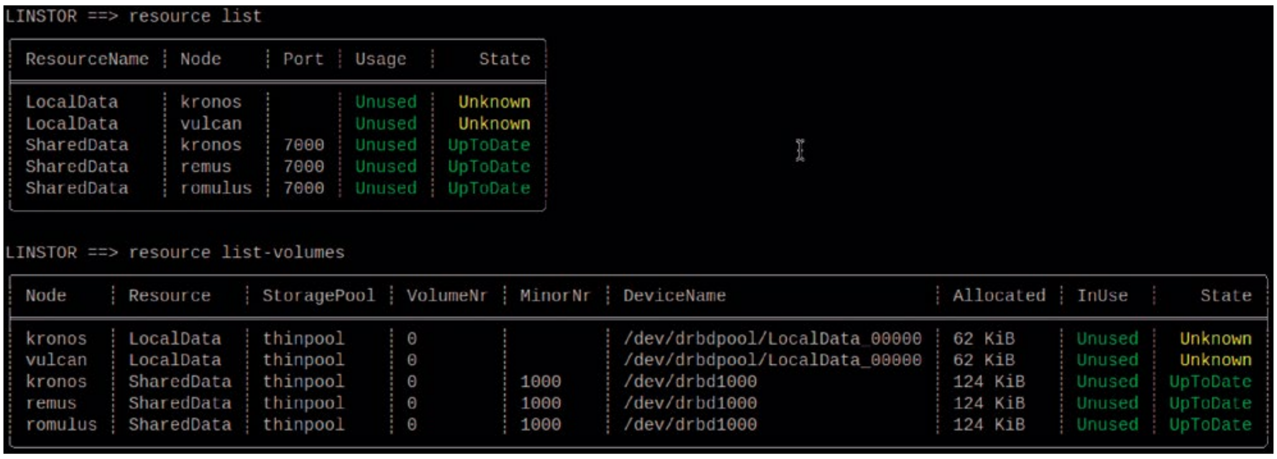
```
snapshot create romulus remus ➘
  SharedData Snap1
```

You can use a snapshot either to create a new resource based on the snapshot's dataset (`snapshot resource restore`) or to roll back the resource from which you took the snapshot to the snapshot version (`snapshot rollback`).

However, both actions can only be performed on the cluster nodes on which the snapshot is available. It is easier to restore the snapshot to a new resource. To do this, first create an "empty" resource definition (without volume definitions) for the new resource:

```
resource-definition create ➘
  SharedData_Restore
volume definition create ➘
  SharedData_Restore 150m
```

You can then restore the snapshot to the new resource:



```
LINSTOR ==> resource list
 ┊ ResourceName ┊ Node    ┊ Port ┊ Usage  ┊    State ┊
 ┊ LocalData    ┊ kronos  ┊      ┊ Unused ┊  Unknown ┊
 ┊ LocalData    ┊ vulcan  ┊      ┊ Unused ┊  Unknown ┊
 ┊ SharedData   ┊ kronos  ┊ 7000 ┊ Unused ┊ UpToDate ┊
 ┊ SharedData   ┊ remus   ┊ 7000 ┊ Unused ┊ UpToDate ┊
 ┊ SharedData   ┊ romulus ┊ 7000 ┊ Unused ┊ UpToDate ┊

LINSTOR ==> resource list-volumes
 ┊ Node    ┊ Resource   ┊ StoragePool ┊ VolumeNr ┊ MinorNr ┊ DeviceName                     ┊ Allocated ┊ InUse  ┊    State ┊
 ┊ kronos  ┊ LocalData  ┊ thinpool    ┊ 0        ┊         ┊ /dev/drbdpool/LocalData_00000  ┊ 62 KiB    ┊ Unused ┊  Unknown ┊
 ┊ vulcan  ┊ LocalData  ┊ thinpool    ┊ 0        ┊         ┊ /dev/drbdpool/LocalData_00000  ┊ 62 KiB    ┊ Unused ┊  Unknown ┊
 ┊ kronos  ┊ SharedData ┊ thinpool    ┊ 0        ┊ 1000    ┊ /dev/drbd1000                  ┊ 124 KiB   ┊ Unused ┊ UpToDate ┊
 ┊ remus   ┊ SharedData ┊ thinpool    ┊ 0        ┊ 1000    ┊ /dev/drbd1000                  ┊ 124 KiB   ┊ Unused ┊ UpToDate ┊
 ┊ romulus ┊ SharedData ┊ thinpool    ┊ 0        ┊ 1000    ┊ /dev/drbd1000                  ┊ 124 KiB   ┊ Unused ┊ UpToDate ┊
```

Figure 3: **Lists of resources and volumes.**

```
snapshot resource restore ↩
  --from-resource SharedData ↩
  --from-snapshot Snap1 ↩
  --to-resource SharedData_Restore ↩
  romulus remus
```

When restoring a snapshot, it is again possible to select a subset of the cluster nodes on which the snapshot is available.

Resetting a replicated resource from which a snapshot was taken to the snapshot version is a little more complicated if the snapshot is not available on all cluster nodes on which the resource was created. In this case, you first have to remove the resource from the cluster nodes where no snapshot is available:

```
resource delete vulcan SharedData
```

LINSTOR may leave the resource as a client resource without backend storage as a quorum tiebreaker resource if this feature is enabled. However, the tiebreaker resource can also get in the way of the dataset reset. You can disable the tiebreaker feature for this resource by manually deleting the tiebreaker resource. In this case, simply repeat the `resource delete` command. The resource is then reset to the snapshot version with the command:

```
snapshot rollback SharedData Snap1
```

Of course, after resetting the dataset, further replicas of the replicated resource can be added to the cluster by creating the respective resource again on additional cluster nodes – which, as expected, requires a resync of the dataset:

```
resource create --storage-pool thinpool ↩
  vulcan kronos SharedData
```

Snapshots are retained even if the original resource from which they were created is deleted. In the LINSTOR object hierarchy, snapshots are linked to the resource definition (Figure 4). Therefore, you cannot delete them until you remove all snapshots.

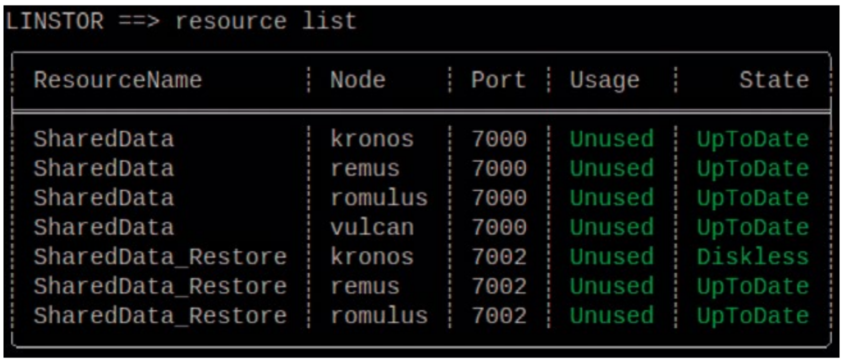
Integration with Virtualization and Container Platforms

Integration with various platforms that need to provide storage volumes automatically is more interesting for storage automation than manual operation of the storage cluster with the LINSTOR client. Included are, on the one hand, popular virtualization platforms like OpenStack, OpenNebula, or Proxmox and, on the other, container-based platforms like Kubernetes. LINSTOR can be integrated into these platforms by means of appropriate drivers so that, for example, when creating a new virtual machine (VM) in OpenNebula, the virtual system disk for this VM is automatically created according to a profile provisioned in LINSTOR. These profiles are known as *resource groups* and can be used to specify certain properties, such as which storage pool to use, a replica count for replication with DRBD, or the integration of a data deduplication layer.

The respective resource, volume definitions, and corresponding resources are created automatically, and various options are also automated. For most platforms, this means that the resources can be allocated in the easiest possible way. For example, the name of the resource definition is chosen to match the name of the respective VM. The drivers for the respective platforms are available from separate GitHub projects [2]; their names usually start with the prefix linstor- (e.g., `linstor-proxmox` and `linstor-docker-volume` <C>).

Conclusions

LINSTOR is a free package that gives administrators a toolkit for automated cluster management. It takes the complexity out of DRBD management and offers a wide range of functions, including provisioning and snapshots. Drivers are already on board for integration into existing cloud frameworks such as Kubernetes, OpenNebula, and OpenStack. A lively community [3] and commercial support from the vendor are both available to help you solve any issues. ■



```
LINSTOR ==> resource list
┊ ResourceName       ┊ Node    ┊ Port ┊ Usage  ┊    State ┊
┊ SharedData         ┊ kronos  ┊ 7000 ┊ Unused ┊ UpToDate ┊
┊ SharedData         ┊ remus   ┊ 7000 ┊ Unused ┊ UpToDate ┊
┊ SharedData         ┊ romulus ┊ 7000 ┊ Unused ┊ UpToDate ┊
┊ SharedData         ┊ vulcan  ┊ 7000 ┊ Unused ┊ UpToDate ┊
┊ SharedData_Restore ┊ kronos  ┊ 7002 ┊ Unused ┊ Diskless ┊
┊ SharedData_Restore ┊ remus   ┊ 7002 ┊ Unused ┊ UpToDate ┊
┊ SharedData_Restore ┊ romulus ┊ 7002 ┊ Unused ┊ UpToDate ┊
```

Figure 4: Status of resources after a snapshot restore and a resync.

Info

[1] LINSTOR: [https://github.com/LINBIT]

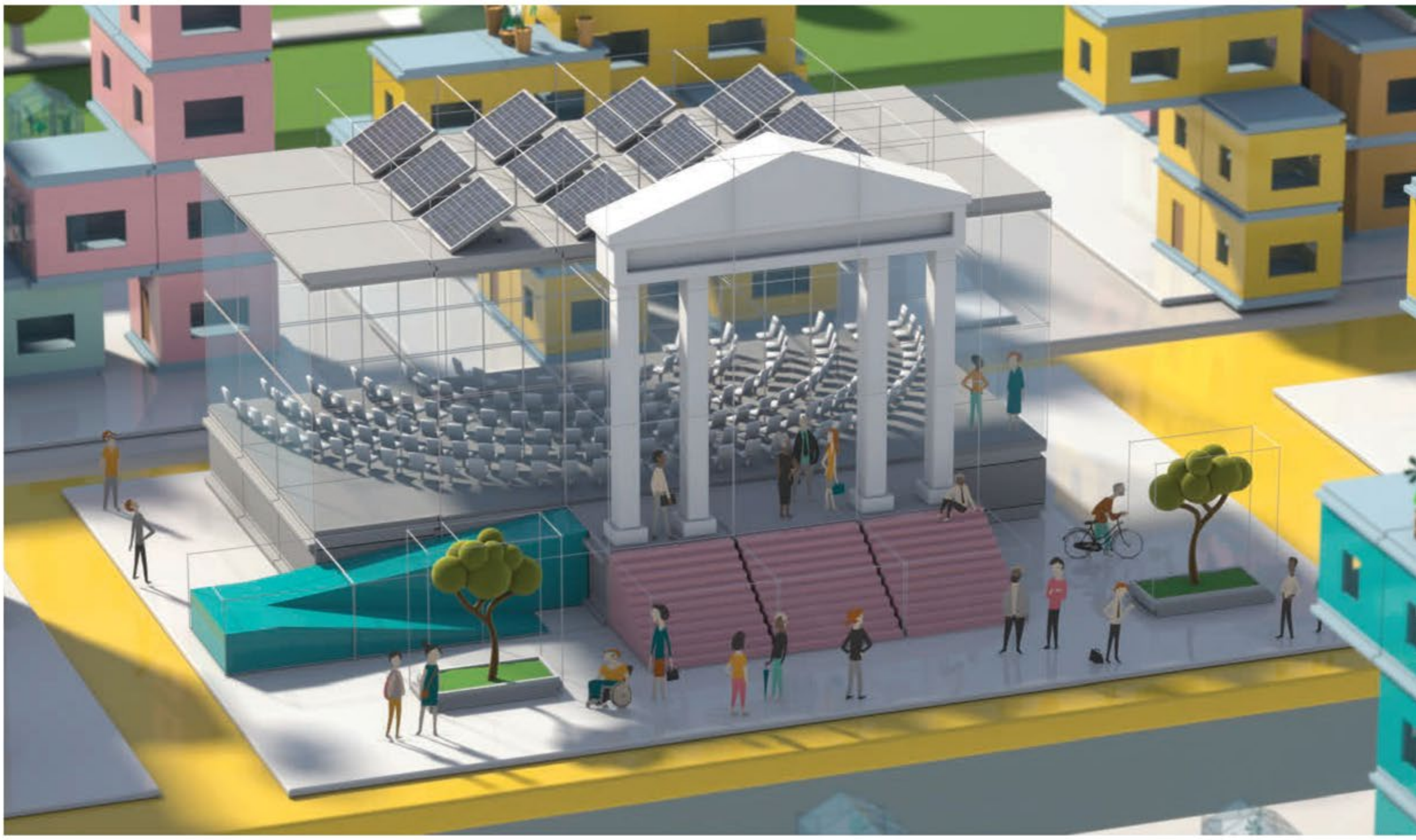
[2] Platform drivers: [https://github.com/LINBIT?q=linstor&type=&language=]

[3] Community support: [https://www.linbit.com/linbit-software-download-page-for-linstor-and-drbd-linux-driver/]

Author

Robert Altnoeder is Principal Architect at LINBIT.

Public Money
Public Code



Modernising Public Infrastructure with Free Software

fsfe Free Software Foundation Europe

Learn More: https://publiccode.eu/