DTNbox (0.10.0)

DTNbox is a new program for directory synchronization between DTN nodes. DTNbox differs from other synchronization programs in because it is peer-to-peer and is fully integrated with the DTN protocols. These characteristics make it well suited for challenged networks, including a future Interplanetary Internet. DTNbox is released as free software under the GPLv3 license.

It is part of the DTNsuite, which consists of several DTN applications making use of the Abstraction Layer, plus the Abstraction Layer itself.

Being built on top of the Abstraction layer, DTNproxy is compatible with all the most important free bundle protocols implementations (DTN2, ION, IBR_DTN).

This document aims at providing the user with:

- release notes
- building instructions
- a brief overview and a concise user guide.

Authors: Marco Bertolazzi (marco.bertolazzi3@studiio.unibo.it, Nicolò Castellazzi (nicolo.castellazzi@studo.unibo.it) , Carlo Caini (Academic supervisor, (carlo.caini@unibo.it)

Credits: Marcello Ballanti, Flavio Poli, Mirco Nani, Imerio Spagnuolo, Andrea Rossetto for preliminary releases.

1 DTNBOX RELEASE NOTES

1.1 DTNBOX 0.10.0 (APRIL 2020)

Use of Abstraction Layer ver.1.7.0 (a few changes in the API syntax), which offers full compatibility with both BPv7 and BPv6 in ION.

1.2 DTNBOX 0.9.0 (29 NOVEMBER 2018)

This is still a development release, although almost stable. Some features, such as the optional passwords are not implemented yet.

Known bugs (work is on to fix them)

- Changes are aggregated in an update command every 5s. The update command consist of a string with triples (path of a file, timestamp, action). If the total length of the string exceeds a fixed dimension (say 2kB) we have a segmentation fault.
- Updates of the database are slow (only about 2 updates per second)

1.3 COMPILATION

DTNbox can be compiled with exactly the same modality as the other application of the DTNsuite. It however requires the presence of the SQLite package. The instructions, common to all DTNsuite applications, are given below.

2 BUILDING INSTRUCTIONS (FOR ALL DTNSUITE APPLICATIONS)

The DTNsuite package consists of a main directory, called “dtnsuite” with one subdirectory for each application, plus one for the Abstraction Layer. For example, the DTNperf code is in “dtnsuite/dtnperf”. As all applications are based on the Abstraction Layer API, before compiling them it is necessary to compile the AL. For the user convenience it is possible to compile the AL and all applications at once (the drawback is that in this case it is more difficult to interpret possible error messages). Both ways are detailed below, starting from the latter, which is preferable for normal users.

Note that the compilation of DTNbox requires the presence of the SQLite package on the host. If not present, you must install it before compilation (“sudo apt-get update, sudo apt-get install libsqlite3-dev”, in Ubuntu). Otherwise, you can comment the DTNbox compilation line in the /dtnsuite/Makefile.

2.1 INTEGRATED COMPILATION

It is possible to compile both AL and all DTNsuite applications in the right sequence with just one simple command. To this end, the Makefile in the “dtnsuite” directory calls the AL Makefile and the DTNsuite application Makefiles in sequential order.

The commands below must be entered from the “dtnsuite” directory. It is necessary to pass absolute paths to DTN2, ION and IBRDTN. The user can compile for one or more ION implementations as shown in the help:

for DTN2 only:

```
make DTN2_DIR=<DTN2_dir_absolute_path>
```

for ION only:

```
make ION_DIR=<ION_dir_absolute_path>
```

for IBR-DTN (>=1.0.1) only:

```
$ make IBRDTN_DIR=<IBRDTN_dir_absolute_path>
```

for all:

```
make DTN2_DIR=<DTN2_dir_absolute_path> ION_DIR=<ion_dir_absolute_path>
IBRDTN_DIR=<IBRDTN_dir_absolute_path>
```

The AL libraries and the DTNsuite applications will have either the extension “_vION” or “_vDTN2” or _vIBRDTN, if compiled for one specific implementation, or no extension if for all.

It is also possible to compile for just two BP implementations, by passing only two paths in the command above.

Finally, the user needs to install the program in the system directory (/usr/local/bin) with the commands (with root permissions)

```
make install
ldconfig
```

Example:

```
<path to>/dtnsuite$ make DTN2_DIR=<path to>/sources/DTN2 ION_DIR=<path
to>/sources/ion-open-source IBRDTN_DIR=<path to>/sources/ibrdtn
<path to>/dtnsuite$ sudo make install
```

```
<path to>/dtnsuite$ sudo make ldconfig
```

For recalling the help:

```
make
```

2.2 SEQUENTIAL COMPILATION

For a better control of the compilation process (e.g. if for whatever reasons it is necessary to change the AL or the specific application Makefiles), it is possible to compile AL and one or more DTNsuite applications sequentially, with independent commands.

2.2.1 Abstraction Layer

The Abstraction Layer (AL) must be compiled first; the AL compilation can be performed by means of the "make" command Entered from the AL directory.

The possibilities are the same as those shown for the integrated compilation and will not reported here for brevity. Even in this case the AL will have either the extension "_vION" or "_vDTN2" or _vIBRDTN, if compiled for one specific implementation, or no extension if for all.

By default the AL library are static. If the default is overridden to dynamic (by modifying the AL Makefile), then the user needs to install the library in the system directory with the command (with root permissions):

```
make install
ldconfig
```

Example:

```
<path to>/dtnsuite/al_bp$ make DTN2_DIR=<path to>/sources/DTN2 ION_DIR=<path
to>/sources/ion-open-source IBRDTN_DIR=<path to>/sources/ibrdtn
<path to>/dtnsuite/al_bp# sudo make install
<path to>/dtnsuite/al_bp# sudo ldconfig
```

2.2.2 Applications

Once the AL has been compiled (and installed if dynamic), each specific application can be compiled in an analogous way. It is just necessary to add the path to the AL-BP (in bold in the example below, for DTN2 only):

```
make DTN2_DIR=<DTN2_dir> AL_BP_DIR=<al_bp_dir>
```

Note that it is necessary to have previously compiled AL_BP for exactly the same ION implementations, as the library with the corresponding extensions must be already present in AL_BP_DIR (vION" or "_vDTN2" or vIBRDTN, or no extension at all if the support is for all implementations).

Finally, the user needs to install the program in the system directory (/usr/local/bin) by Entering the same commands used for both the integrated compilation and the AL.

```
make install
make ldconfig
```

Example (for DTNperf)e:

```
<path to>/dtnsuite/dtnperf$ make DTN2_DIR=<path to>/sources/DTN2 ION_DIR=<path
to>/sources/ion-open-source AL_BP_DIR=<path to>/UniboDTN/al_bp
<path to>/dtnsuite/dtnperf$ sudo make install
<path to>/dtnsuite/dtnperf$ sudo ldconfig
```

The “dtnperf” executable will have either the extension “_vION” or “_vDTN2” or _vIBRDTN, if compiled for one specific implementation; no extension if compiled for all, as always.

3 OVERVIEW

3.1 BILATERAL SYNCHRONIZATIONS.

DTNbox is a relatively complex application, hence let us start from the simplest use case.

Bilateral synchronizations and synchronization modes.

In the following, a synchronization that involves just two nodes is called “bilateral”. One of these nodes is the “owner” of the directory synchronized, the other has a local copy. This synchronization can be either bidirectional or unidirectional (Figure 1).

In bidirectional synchronizations all the changes made on either of the two paired directories are mirrored in the other, which is useful when two nodes, e.g. A and B, want to share the content of a directory. By contrast, a unidirectional synchronization is designed to prevent changes on B being propagated backwards to A, and is more suitable for backing up directories, or for distributing content from A to B. In both cases, unidirectional synchronization prevents node B from altering or deleting files on node A. Once established, node A is in “push” mode, i.e. ready to send updates, and B in “pull”, i.e. ready to receive.

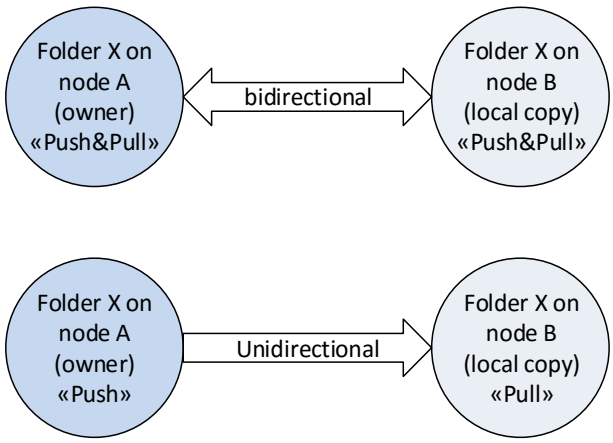

FIGURE 1: BILATERAL SYNCHRONIZATIONS: A) BIDIRECTIONAL AND B) UNIDIRECTIONAL.

3.1.1 Paired directories

All directories paired by DTNbox must be subdirectories of a directory named after the original owner. Let us assume that a user on node A (EID: dtn://nodeA.dtn) wants to synchronize a directory with node B (EID: dtn://nodeB.dtn). The first step is to launch DTNbox on node A, which creates the “/DTNbox/foldersToSync/nodeA” directory, if not yet present. Then the user on node A ask DTNbox to add the subdirectory that it wants to synchronize, e.g. “myphotos” (Figure 2). After that, it sends a synchronization request to node B, where DTNbox is already running. If node B accepts, a local copy “/DTNbox/foldersToSync/nodeA/ myphotos” is created in node B file system and paired with the original on node A.

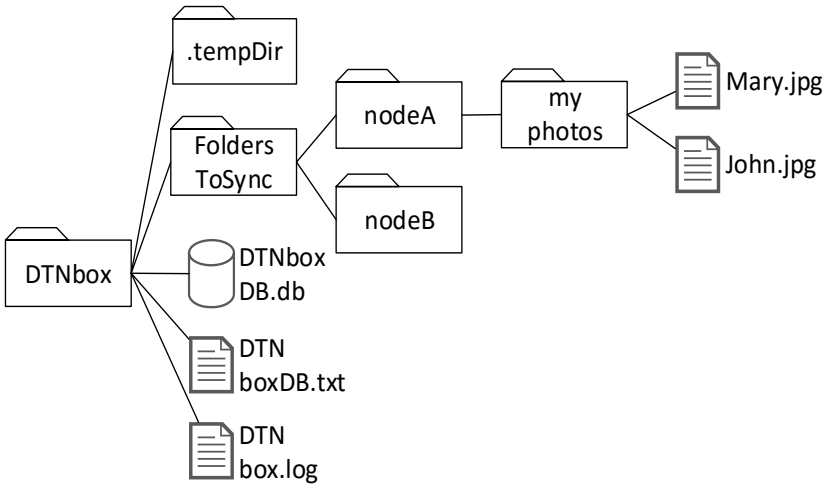

FIGURE 2: DTNBOX FILE SYSTEM ARCHITECTURE. THE HIDDEN DIRECTORY .TEMPDIR IS USED BY DTNBOX INTERNAL PROCESSES. FOLDERSTOSYNC CONTAINS "NODE" DIRECTORIES; MYPHOTOS IS THE DIRECTORY, OWNED BY NODE A, ON WHICH THE SYNCHRONIZATION IS ACTIVE. DTNBOXDB.DB CONTAINS THE SQLITE DATABASE; FINALLY, DTNBOXDB.TXT AND DTNBOX.LOG ARE TWO AUXILIARY FILES FOR DEBUG PURPOSES.

3.1.2 Synchronization requests

To establish a bidirectional synchronization, the owner can send a "push&pull_out" request to another node (from node A to B in our example). The synchronization can also be requested from the owner by another node, by means of a "push&pull_in" (from B to A). Once established both nodes work in "push&pull" mode.

To establish a unidirectional synchronization, the owner (node A) can send a "push" request to another node (B); The synchronization can also be established as a result of a "pull" request sent to the owner (A) by another node (B). Once established, the owner works in "push" and the other in "pull".

3.2 MULTILATERAL SYNCHRONIZATIONS

Starting from an ongoing bilateral synchronization, it is possible to establish multilateral synchronizations, i.e. involving more than two nodes.

3.2.1 Extensions from the owner node

The first step is always to establish a bilateral synchronization between two nodes, e.g. nodes A and B, as in the previous example. Let us assume that node A wants to synchronize his "myphotos" directory with node C. If C accepts the synchronization request sent by A, the result will be a trilateral synchronization, with a local copy of "myphotos" on node C too. This of course can be extended to other nodes as well, as shown in Figure 3.

It may also happen that a third node (C) wants to Enter into an existing synchronization, from A and B. If C asks node A (owner) and A accepts, the synchronization starts.

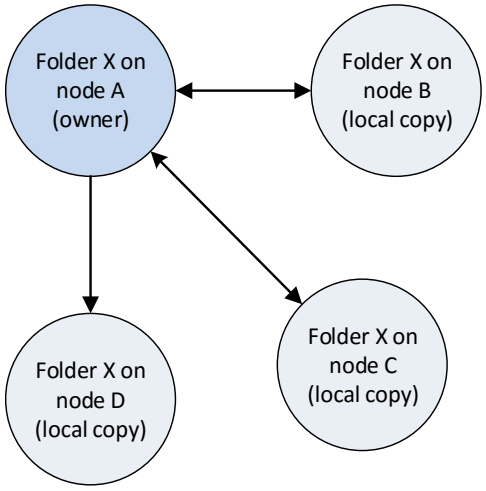

FIGURE 3: MULTILATERAL SYNCHRONIZATIONS OBTAINED BY EXTENSIONS FROM THE OWNER NODE.

3.2.2 Extensions from other nodes

If the ongoing synchronization is bidirectional, the paired node (B) can also invite other nodes to join (e.g. C), or another node (C) wants to Enter an existing synchronization from A and B by asking B, who is not the owner. A possible result is the multilateral synchronization shown in Figure 4.

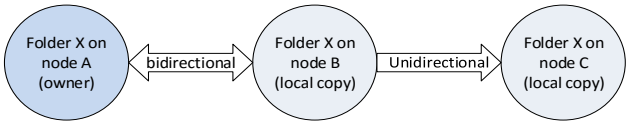

FIGURE 4: MULTILATERAL SYNCHRONIZATION OBTAINED BY EXTENSION FROM A PAIRED NODE

3.2.3 Synchronization trees

As a result of multiple extensions, complex multilateral synchronizations can be established, as that shown in Figure 5. Note that the layout is a tree, as loops must be avoided. To this end, DTNbox sets automatic constraints on possible extensions, preventing the user from involuntarily building loops. Of course, such complex layouts are not recommended unless strictly dictated by specific needs.

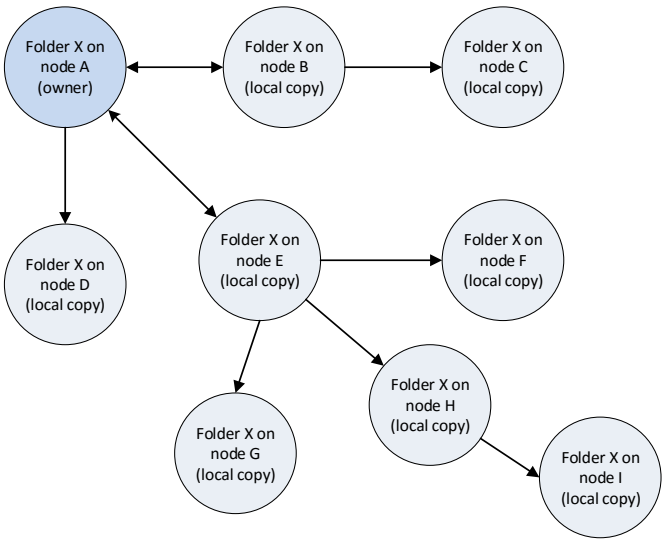

FIGURE 5: MULTILATERAL SYNCHRONIZATIONS: COMPLEX SYNCHRONIZATION TREES.

3.3 DTNBOX DATABASE

DTNbox relies on an external relational database to maintain the state of synchronizations, based on SQLite (**Errore. L'origine riferimento non è stata trovata.**). Every synchronization necessarily involves an exchange of information (not just files, but also commands and command acknowledgments) between pairs. If the network is challenged, this exchange is just more difficult than usual, because it is necessary to reduce chattiness to a minimum. To this end, at regular intervals (e.g. 5s) DTNbox aggregates commands (including file updates) and acknowledgments in one .tar file, to be sent to the pair and in turn encapsulated in one bundle.

The synchronization mechanism is based on synchronization states recorded in the database, and being quite complex is not reported here for brevity. Let us just present here the figures of the synchronization states for files and the few selected commands that require confirmation and resending (Figure 6 and Figure 7).

The retransmission timeout (loop of "Pending" state in Figure 6) is set to twice the bundle lifetime and the total number of transmissions, as the lifetime, can be set according to the destination. DTNbox retransmissions, if considered redundant in a specific deployment, can be disabled by setting "numTx"=1 to all nodes. In all BP implementations, bundle parameters, as defined in the BP "Primary Block" processing flags, e.g. priorities, custody option, status reports, etc., can be set by the user. At present these can be changed only in the DTNbox code.

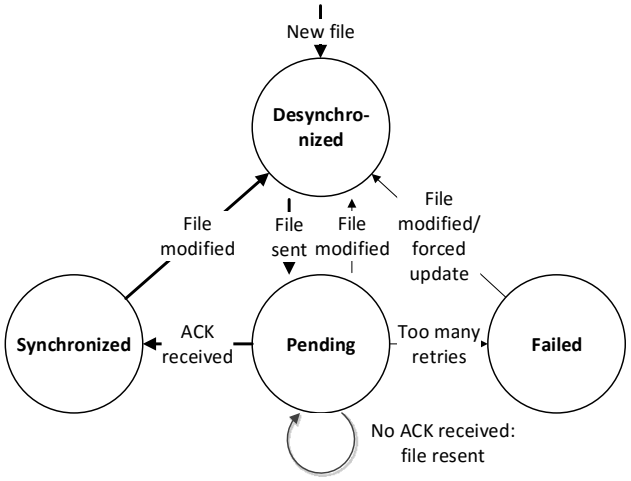

Figure 6: State diagram of the synchronization process of one file. Arrows in bold denote the normal transitions between Desynchronized, Pending and Synchronized states.

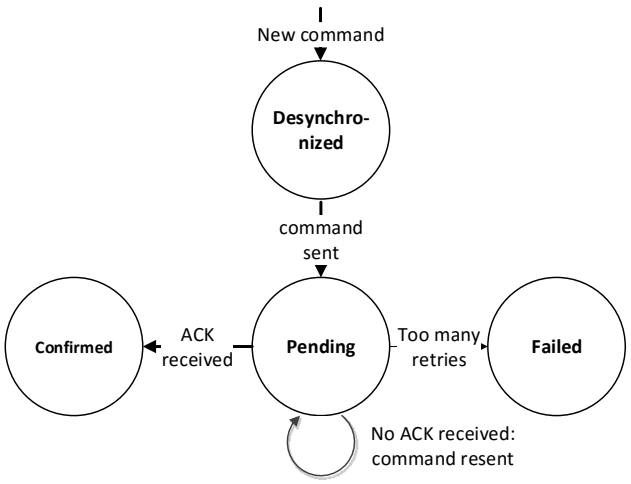

Figure 7: State diagram of commands requiring both to be acknowledged and (possibly) retransmitted. Arrows in bold denote the normal transitions between Desynchronized, Pending and Confirmed states.

4 EXAMPLE OF USE

We report here the commands to be Entered to reproduce the 3 node layout shown in Figure 4, as an example. Each node is implemented as one VM, running ION (whichever BP implementation supported by the Abstraction Layer would be possible as well).

After starting ION and DTNbox on all nodes, we create the first bidirectional synchronization between vm1 (identified as "ipn:1.0" and vm2 (ipn:2.0), by Entering the following DTNbox commands and parameters (in bold) from a vm1 terminal:

```
mainDTNbox: current user 1

userThread: waiting for commands
```

add node

- `userThread: Enter the node EID` **`ipn:2.3000`**
- `userThread: Enter the bundle lifetime for this node (in seconds)` **`60`**
- `userThread: Enter the number of Tx attemps for this node` **`2`**
- `userThread: W to add the node to the whiteList, B to the blackList:` **`W`**
  `userThread: node successfully added!`

```
userThread: waiting for commands
```

add folder

- `userThread: Enter the folder name (e.g. photos)` **`myphotos`**
  `userThread: folder successfully created and added!`

```
userThread: waiting for commands
```

add sync

- `userThread: Enter the folder owner:` **`1`**
- `userThread: Enter the folder name:` **`myphotos`**
- `userThread: Enter the EID of the synchronization pair:` **`ipn:2.3000`**
- `userThread: Enter synchronization mode:` **`PUSH&PULL_OUT`**
  `userThread: synchronization request created!`
  `receiveThread: received bundle /tmp/ion_2509_0 from ipn:2.3001`
  `syncAckCommand_receiveCommand: sync request has been accepted`

On vm2, as soon as the request is accepted, node "ipn:1.0" is added to the "nodes" table of the local database (with the same lifetime of the bundle containing the request); a new synchronization entry is created in the "synchronizations" table, and a local copy of the folder is created in the local file system.

To extend the synchronization to vm3 ("ipn:3.0") we Entered on vm3:

```
mainDTNbox: current user 3

userThread: waiting for commands
```

add node

- `userThread: Enter the node EID` **`ipn:2.3000`**
- `userThread: Enter the bundle lifetime for this node (in seconds)` **`60`**
- `userThread: Enter the number of Tx attemps for this node` **`2`**
- `userThread: W to add the node to the whiteList, B to the blackList:` **`W`**
  `userThread: node successfully added!`

`userThread: waiting for commands...`

`add sync`

- `userThread: Enter the folder owner:` **`1`**
- `userThread: Enter the folder name:` **`myphotos`**
- `userThread: Enter the EID of the synchronization pair:` **`ipn:2.3000`**
- `userThread: Enter synchronization mode:` **`PULL`**
  `userThread: synchronization request created!`
  `receiveThread: received bundle /tmp/ion_4403_0 from ipn:2.3001`
  `syncAckCommand_receiveCommand: sync request has been accepted`

After having established the synchronizations, we can add/update/remove files in the myphotos directory. Changes on vm1 will propagate to vm2 and in turn to vm3; on vm2 propagated to vm1 and vm3; on vm3 did not propagate.

Eventually, we can delete the synchronization by Entering on vm1:

`userThread: waiting for commands...`

`delete sync`

- `userThread: Enter the folder` **`1`**
- `userThread: Enter the folder name` **`myphotos`**
- `userThread: Enter the EID of the node with which you want to terminate the synchronization` **`ipn:2.3000`**
- `userThread: warning: by deleting the synchronization with the parent (PULL or PUSH&PULL_IN) your local folder will be deleted, are you sure? (YES, upper case)` **`YES`**
  `userThread: synchronization deleted, the node will be notified!`

As a result, the synchronization entry is deleted on all nodes, as were all local copies of the “myphotos” directory, as expected.