DTNfog (1.1.0)

DTNfog is a program that aims to implement a “fog” architecture in a DTN environment. The DTNfog node receives a .tar file containing a file and a command to be applied on the file by a node of lower hierarchical level. If the DTNfog node is able to execute the command it gives the response back, otherwise it forwards the request to the upper layer, and so on. Files can be transferred either by TCP or by Bundle Protocol. It has been designed for challenged networks, including a future Interplanetary Internet.

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:

1) release notes
2) building instructions
3) a brief overview and a concise user guide.

Authors: Lorenzo Tullini (lorenzo.tullini@studio.unibo.it), Carlo Caini (Academic supervisor, (carlo.caini@unibo.it)

Table of Contents

1 DTNFOG RELEASE NOTES

1.1 DTNFOG 1.1.0 (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 DTNFOG 1.0.0 (23 JULY 2019)

1.2.1 Compilation

DTNfog can be compiled with exactly the same modality as the other application of the DTNsuite. The instructions, commons 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.

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):

```
<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 DTNFOG OVERVIEW AND GUIDE OF USE.

3.1 OVERVIEW

DTNfog is a DTN application that aims to extend the “Fog Computing” model to challenged networks. DTNfog allows the user to build a hierarchical architecture where each DTN node, running DTNfog, can execute a pre-defined set of commands, specific to each node. We can have multiple intermediate (“fog”) layers between nodes belonging to the lowest (IoT) and the highest layer (Cloud). Each DTNfog node can receive/transmit either via TCP (when the corresponding link is not challenged) or via BP (in the presence of challenges). In particular, the possibility of receiving via TCP, allows DTNfog nodes of the lowest fog layer to be reached even by IoT nodes unable to use the Bundle Protcol (i.e. that are not DTN nodes) because of hardware limitations.

3.2 THE DTNFOG ARCHITECTURE

The architecture that can be built by means of DTNfog is illustrated in Figure 1. To its description is useful to examine the flow of a generic request. One IoT node, being unable to locally execute a command (e.g. to calculate the frequency of characters of a text file, or its entropy) sends the request to its upper “Fog” node (e.g. DTNfog1); if this node is able to process it, it execute the command and sends back the response; otherwise, it forwards the request to its upper DTNfog node (e.g. DTNfog2), and so on until the highest node of the hierarchical structure, the Cloud node (i.e. a DTNfog node that has no upper nodes). We can distinguish between 3 kind of nodes:

- Client node: it is typically an IoT node, which originates a request being unable to locally execute a command (e.g. to calculate the frequency of characters in a text file), sends it to its reference Fog node and waits for the answer. It does not require DTNfog. In fact, it may be or may be not a DTN node.
- Fog node: it is an intermediate node, belonging to a fog layer located between the IoT layer and the Cloud. It receives a request from a client or from a lower fog node; if it can execute the request, it sends back the answer, otherwise it forwards the request to its upper node, either fog or cloud. It is a DTN node and needs DTNfog.
- Cloud node: it is the highest node of the chain, and it is the same as a fog node, buti t has no upper nodes, thus, if it cannot execute a command, it cannot further forward the request but it has to send back a negative response.

Requests and responses are contained in a .tar file, to multiplex in one file:

- the commands to be executed (e.g. the analysis of a text file)
- the data on which to execute the commands (e.g. one or more text files)
- other data necessary to the forwarding back of the answer).

This design choice is dictated by the desire to make the requests “atomic” (all is needed is in the file) and to make DTNfog “status free” (no need to register request forwarded to upper nodes), which is clearly advantageous in terms of simplicity and scalability.

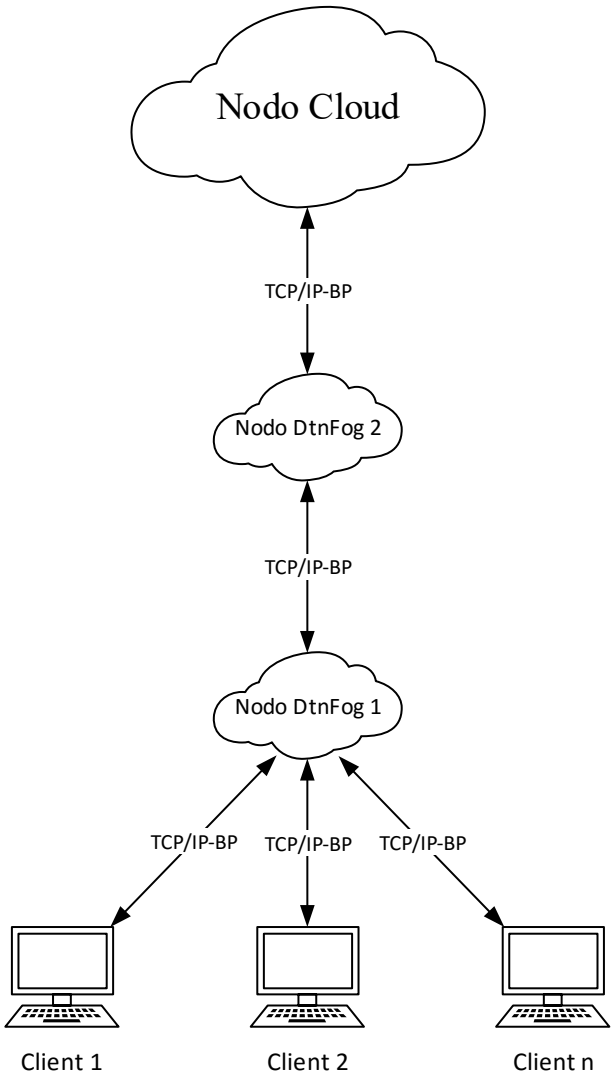

Figure1: The DTNfog architecture (two intermediate fog layers).

3.3 STRUCTURE OF THE ARCHIVE “.TAR” FILE

The communication between DTNfog nodes is carried out by means of an archive file, whose structure is presented in Figure 2It consists of the following files **:**

- command.cmd (the command to be executed),
- route.rt (the route followed by the request, i.e. the chain of DTNfog nodes, to be used in the reverse direction by the response),
- one or more files on which to execute the command (optinal), e.g. the text file to be analyzed
- report.rp (optional)
- result.rs (the result of the command, inserted after execution).

The extensions: “.rt”, ”.cmd”, “.rp” e “.rs” are reserved to DTNfog.

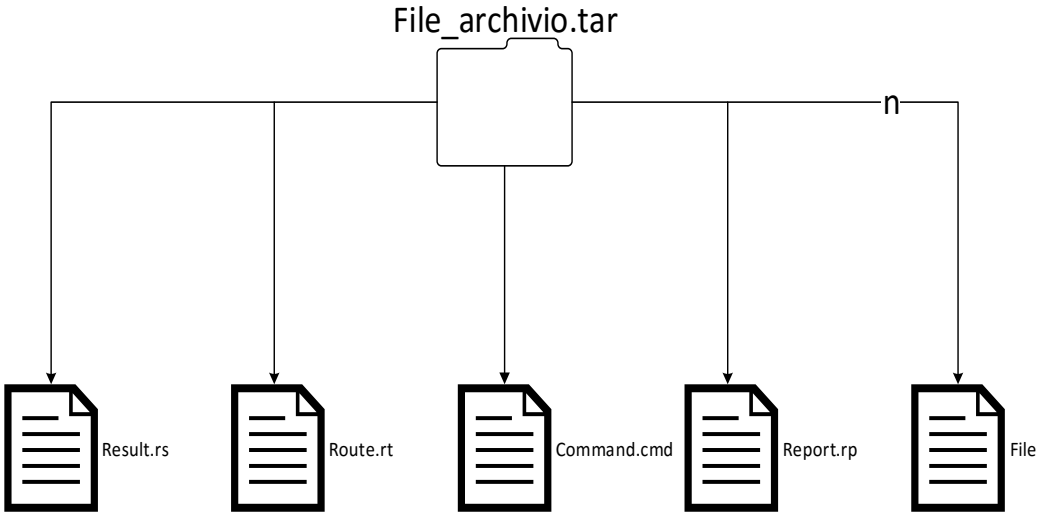

Figure 2 structure of the archive file used by DTNfog

3.3.1 The “command.cmd” file

This file is mandatory, as it contains the command whose execution is required by the IoT node. Each DTNfog node of the IoT-Cloud chain tries to execute it if the command is included in a white list specific to each node. .The command is followed by the name8s) of the file(s) necessary to its execution. For example,

```
contaCaratteri divina_commedia.txt
```

3.3.2 The “route.rp” file

This file contains the list of nodes followed by the request, to be used in the reverse direction once the command has been executed (or in case of failure). Is a sort of “record route” file, similar to the Ariadne’s thread, whose aim is to keep track of the route followed to be able to go back in the reverse direction. It allows DTNfog not to keep memory of requests received and forwarded on the local node.

In the ascending phase, each DTNfog node that does not manage to execute the command annotes the address at which it desires to receive an answer in the route.rp file; the first DTNfog node that succeeds starts the descending phase, where the last address in the file is read to send back the response, and then eliminated. Note that adresses can be IP or BP, and in the latter case following the “dtn” or the “ipn” scheme. The response will be sent back via IP or BP according to the address. The route.rt file is created and initialized by the client (IoT) node that makes the requests. Adresses must contains the port number (IP) or the demux token (BP) and must be followed by the new line character (“\n”), For example, we could alternatively have for the same node:

- @10.0.0.1:2500\n (for IP)
- @ipn:5.2000\n (for BP with ipn scheme)
- @dtn://nomedelnodo/dntfog\n (for BP with dtn scheme)

3.3.3 The “report.rp” file

This auxiliary file is optional. If inserted by the IoT client, all DTNfog nodes in both the ascending and descending chain, write debug information in it.

3.3.4 The “result.rs” file

The file results.rs is inserted into the archive file by the DTNfog node that eventually manages to execute the command. It contains a copy of the standard output and of the standard error generated by the command requested when executed. If the file is missing, DTNfog recognizes that the command has not been executed yet, i.e. that it has been received by a lower node. If the request goes up until the Cloud node, and even it cannot execute the command, a file

results.rs is inserted with a message of error. Success or failure, once a report.rt file is inserted the descending phase starts. As a results.rs phase is present, DTNfog forward back the response looking at the last address in the route.rt file, as previously described, until the IoT client originating the request is reached.

3.4 USE

3.4.1 Syntax

```
SYNTAX: DTNfog [options]
```

The syntax of DTNfog is very simple, with no compulsory parameters and only few options, all without arguments:

- "- - help"
- "- - no-header" (to omit the insertion of the DTNperf header when the archive file is sent via BP)
- "- - debug" (to have log messages on screen)

Only the second deserves an explanation. When used, the"dtnSender" thread, inherited by DTNproxy, inserts the archive file in the bundle payload as it is, i.e. by omitting the insertion of the DTNperf header.

Configuration parameters, necessary to DTNfog correct execution, are inserted not in line, but in a configuration file, specific to each node. An example is given below.

3.4.2 Configuration file

Let us consider the following example:

```
###list-commands##
contaCaratteri
#who
##end-list-commands##

##cloud-mode##
false
##cloud-mode##

##next-address##
10.0.0.11:2500
##end-next-address##

##tcp-interface##
tap0
##end-tcp-interface##

##allow-reports##
true
##end-allow-reports##
```

Configuration settings are ornaized in sections, with a syntax that looks like that used in ION .rc configuration files. The first section lists the command whose execution is both possible and allowed, i.e. the white list of commands. In the example, only the command "contaCaratteri" is allowed. The second section is used to let the node know if it is the Cloud node or an ordinary DTNfog node (ordinary in the example). The next section contains the address (eithr IP or BP) of the next higher node (IP 10.0.0.11 port 2500). The following section contains the interface at which the current DTNfog node is reachable via TCP; the corresponding IP address, to be inserted in route.rt is derived automatically by DTNfog from this information (the port is always 2500). Note that it does not exist a corresponding DTN section, as DTNfog is always able to derive its own DTN address, which is unique, automatically (one DTN node has one DTN address, even if it may have as many IP addresses as NIC cards). The tcp-interface section is thus necessary to identify

the IP address to be used, among the many possibly available. Note that DTNfog will insert its IP or DTN address in the file route.rt depending on the kind (TCP or DTN respectively), inserted in the next-address section. The last section enables the insertion of debug information in the file report.rp, if present. This feature has been inserted should an intermediate node not to be willing to disclose its identity and position in the hierarchical chain to lower nodes for security reasons. In the DTNfog architecture, a node only needs the knowledge of its direct superior, not of the full chain of higher nodes.

3.4.3 The IoT node

The IoT node is the only node that does not require the use of DTNfog. In facts, it may be or may be not a DTN node. The only requirement is that It be able to prepare a DTNfog-compliant request, i.e. an archive following the structure previously illustrated. The archive file must then sent to the next DTNfog node either via IP or BP.

3.5 AN EXAMPLE OF USE

We present an example where the request generated by the IoT node can be executed only by the Cloud node. We have 4 nodes, the IoT node (on the host), two DTNfog nodes (on VM1 and VM2, respectively) and a clod node (on VM4). The DTNfog configuration files of the last 3 nodes is presented below. (Figure 4). Note that the communication between VM1 and VM2 is via DTN, as in VM1 configuration file we have as “next address” the DTN address of VM2, while that between VM2 and VM4 is via TCP, as in the configuration file of VM2 we have the IP address of VM4. As VM4 is delared as Cloud node its next-address is irrelevant. All nodes declare eth0 as “tcp-interface”. This is useful only to automatically derive the corresponding IP address to be inserted in route.rt, should the IP protocol to be used.

```
Configurazione VM1
###list-commands##
contaCaratteri
#who
##end-list-commands##

##cloud-mode##
false
##cloud-mode##

##next-address##
ipn:2.5000
##end-next-address##

##tcp-interface##
eth0
##end-tcp-interface##

##allows-report##
true
##end-allow-reports##
```

```
CONFIGURAZIONE VM2
##list-commands##
contaCaratteri
#who
##end-list-commands##

##cloud-mode##
false
##cloud-mode##

##next-address##
10.0.0.14:2500
##end-next-address##

##tcp-interface##
eth0
##end-tcp-interface##

##allows-report##
true
##end-allow-reports##
```

```
CONFIGURAZIONE VM4
###list-commands##
reportCaratteri
##end-list-commands##

##cloud-mode##
true
##cloud-mode##

##next-address##
10.0.0.11:2500
##end-next-address##

##tcp-interface##
eth0
##end-tcp-interface##

##allows-report##
true
##end-allow-reports##
```

Figure 4:DTNfog configuration files on vm1,vm2 e vm4

The archive file must be prepared on the IoT node. It will contain the report.rp file to record debug information from all DTNfog nodes. The command.cmd file contains the following line:

```
reportCaratteri divina_commedia.txt
```

which will give as a result a table with the frequency of characters present in the divina_commedia.txt file (obviously containing all Divina Commedia by Dante) and an estimate of the corresponding entropy of the alphabet. The IoT node is assumed to be a DTN node in this case, and it specifies as its response address in route.rt the ipn address of the host, (the demux token is that typical of DTNperf server, 2000, see below):

```
@ipn:5.2000
```

On the IoT node it is launched DTNperf with the option - - server (i.e. in server mode), to receive the response via BP. It automatically registers itself as ipn:5.2000, i.e. the address specified in route.rt. After launching DTNfog on all other nodes, the archive can then be sent from the IoT node to the next DTNfog node (on VM1) alternatively either via BP, by the DTNperf client, or via TCP by the TCPclient auxiliary program of DTNproxy.

The request sent to VM1 is forwarded via BP to Vm2, as VM1 has not the command "reportCaratteri" in its white list. VM2 does the same, forwarding the request to VM4 (this time via IP), which can execute the command (is in its white list). The file results.rt is inserted and the response is sent back to the IoT node, via VM2 and VM1.

At the end, we receive on the IoT an archive file containing both the report.rp file (with debug information) and the results.rt file (Figure 12 and 13 respectively).

```
Processed by 10.0.0.11:2500 fog
     |
     V
10.0.0.11:2500: result.rs not found, tries to execute command.cmd
     |
     V
10.0.0.11:2500: says command.com not found, send next hop
     |
     V
Processed by ipn:2.5000 fog
     |
     V
ipn:2.5000: result.rs not found, tries to execute command.cmd
     |
     V
ipn:2.5000: says command.com not found, send next hop
     |
     V
Processed by 10.0.0.14:2500 cloud
     |
     V
10.0.0.14:2500: result.rs not found, tries to execute command.cmd
     |
     V
10.0.0.14:2500: command.cmd successful executed, send back to previous node
     |
     V
Processed by 10.0.0.12:2500 fog
     |
     V
10.0.0.12:2500: result.rs found, send back to previous node
     |
     V
Processed by ipn:1.5000 fog
     |
     V
ipn:1.5000: result.rs found, send back to previous node
```

Figure 5 The report.rp file of the example

```
-------------------------------------------------
|                Report caratteri               |
-------------------------------------------------
                Occorenze Caratteri             |
-------------------------------------------------
a = 13783   j = 1       s = 11377
b = 2291    k = 0       t = 11243
c = 10982   l = 11647   u = 9173
d = 9726    m = 8201    v = 6285
e = 13964   n = 12277   w = 0
f = 4159    o = 13554   x = 3
g = 5393    p = 7771    y = 1
h = 6089    q = 2755    z = 1452
i = 13555   r = 12590
-------------------------------------------------
                Frequenza caratteri             |
-------------------------------------------------
a = 0.0732079   j = 5.31146e-06 s = 0.0604285
b = 0.0121686   k = 0       t = 0.0597168
c = 0.0583305   l = 0.0618626   u = 0.0487221
d = 0.0516593   m = 0.0435593   v = 0.0333826
e = 0.0741693   n = 0.0652088   w = 0
f = 0.0220904   o = 0.0719916   x = 1.59344e-05
g = 0.0286447   p = 0.0412754   y = 5.31146e-06
h = 0.0323415   q = 0.0146331   z = 0.00771225
i = 0.0719969   r = 0.0668713
-------------------------------------------------
             Entropia Alfabeto = 4.22707        |
-------------------------------------------------
```

Figure 6 The result.rt file of the example