Peer Module Module

Release 0.9.02

Peer NodeBrain Module
August 2014
NodeBrain Open Source Project

Release 0.9.02

Author: Ed Trettevik

MIT License

NodeBrain License

History

2005-10-12 Title: *Peer NodeBrain Module*
Author: Ed Trettevik <eat@nodebrain.org>
Publisher: NodeBrain Open Source Project

2012-06-18 Release 0.8.10

- Revised to reflect changes since release 0.7.3.

Preface

This manual is intended for users of the Peer NodeBrain Module, a plug-in for communication between NodeBrain processes. This module is expected to change a bit by release 1.0, primarily by the addition of an option to use TLS in addition to, or instead of, the current authentication and encryption features.

The Message module is now the preferred method of exchanging high volume messages between NodeBrain processes. The Peer module is still preferred in low volume cases where it is helpful to see a response to individual commands.

See www.nodebrain.org for more information and the latest update to this document.

Documents

NodeBrain Guide - Information on using `nb`
NodeBrain Tutorial - A gentle introduction to `nb` and the rule language
NodeBrain Language - Rule language syntax and semantics
NodeBrain Library - C API

Document Conventions

Sample code and input/output examples are displayed in a monospace font, indented in HTML and Info, and enclosed in a box in PDF or printed copies. Bold text is used to bring the reader's attention to specific portions of an example. In the following example, the first and last line are associated with the host shell and the lines in between are input or output unique to NodeBrain. The `define` command is highlighted, indicating it is the focus of the example. Lines ending with a backslash \ indicate when a command is continued on the next displayed line. This is supported by the language within source files, but not for other methods of command input. If you copy an example of a command displayed over multiple lines, you must enter it as a single line when used outside the context of a source file.

```
$ nb
> define myFirstRule on(a=1 and b=2) mood="happy";
> assert mood="sad";
> show mood
mood = "sad"
> assert a=1,b=2,c=3,d="This is an example of a long single line that",\
    e="we depict on multiple lines to fit on the documnet page";
2008/06/05 12:09:08 NB000I Rule myFirstRule fired(mood="happy")
> show mood
mood = "happy"
> quit
$
```

Table of Contents

1 Concepts

The peer module provides support for external communication between NodeBrain nodes within the same process, in two separate processes on the same system, or two processes running on different systems. This includes TCP/IP communication using network or local domain sockets and communication via peer queue files. Both methods can be combined to provide store-and-forward transmission of messages to ensure delivery when interrupted by network or agent outages.

The peer module implements the NodeBrain protocol (NBP) originally built into the NodeBrain Interpreter.

1.1 Identity

NodeBrain associates permissions with identities, which are nothing more than names to which permissions are be granted. An identity may map to something less abstract, like a person, account, application, component, or role. NodeBrain makes no assumptions about how you map an abstract identity to something specific.

Modules that enable NodeBrain commands to be executed remotely, as the Peer module does, must map incoming commands to an identity. The Peer module does this by mapping authentication keys to identities. Each end of a connection (client and server) claim an identity and each is authenticated by the other using keys. Commands issued locally on behalf of the peer, are issued under the peer's authenticated identity.

1.2 Keys

The peer module uses private and shared keys. Shared keys are chryptographically known as public keys, except here they are not intended to be widely published. Instead they are shared within a small circle of users, often the same user on multiple servers. Keys are stored in a key file named '`$HOME/.nb/nb_peer.keys`', a file readable and writable only by the owner.

A key file with identities `tigger` and `lion` is illustrated below.

```
lion   3.e57f2e4dbe5d8bc4.0.0;                 # Shared key for lion
tigger 3.be5d8bc4465d3aa7.bd6107c786f72c15.0;  # Private key for tigger
```

The difference between a shared and private key can be seen in the third component, which is `0` in a shared key.

A new private key can be generated using the `identify` Peer module command.

```
nb :"peer.identify bear"
```

The `identify` command appends a new private key to the key file for the specified identity.

```
lion   3.e57f2e4dbe5d8bc4.0.0;                  # Shared key for lion
tigger 3.be5d8bc4465d3aa7.bd6107c786f72c15.0;  # Private key for tigger
bear   5.65d3aa7be5d8bc44.6f72c15bd6107c78.0;
```

2 Tutorial

> *The more elaborate our means of communication, the less we communicate.*
> —Joseph Priestly (1733–1804)

The peer module is more elaborate than some of the other modules used for communication. The goal is to communicate less, or at least to be more selective in with whom you communicate. This selectivity is accomplished with encryption and key-based authentication.

The files for this tutorial are in the `tutorial/Peer` directory. The NodeBrain scripts are executable and I have left off the `.nb` suffix for fun.

2.1 Peer Module Identity Keys

Before you communicate using the peer module, you must create keys to be used by clients and servers. To reduce complexity for this example, you'll create a single key and use it for both the client and the server as a "shared secret" key. Here's a script called `genkey` that creates a key for an identity called `buddy`.

```
#!/usr/local/bin/nb
# File: tutorial/Peer/genkey
define myService node peer.service;
myService:identify buddy;
```

The peer module provides a skill called `service` that supports some helpful commands associated with peer communication. Use the `identify` command here. This command generates a key and places it in a key store for later reference.

Unix and Linux: `~/.nb/nb_peer.keys`
Windows: `USER_PROFILE/ApplicationData/NodeBrain/nb_peer.keys`

The key store is readable only by the owning user and looks like this.

```
$ cat ~/.nb/nb_peer.keys
foo 3.3788e45e8f64776b.0.0;
bar 7.b49ad9bfb68a97b8.fab67908064b5cb3.0;
buddy 3.be5d8bc4465d3aa7.bd6107c786f72c15.0;
```

2.2 Peer Module Server Node

> *Be alert to give service. What counts a great deal in life is what we do for others.* —Anonymous

A peer server node is alert to give service, in fact, it can even give service to alerts. It provides a method for a peer client to send any command to a NodeBrain agent: alerts, assertions, new rule definitions, and so on. It accepts TCP/IP socket connections from clients and issues received commands in the context of the server node. The `server` file in the `tutorial/Peer` directory looks like this.

```
#!/usr/local/bin/nb -d
# File: tutorial/Peer/server
-rm server.log
set out=".",log="server.log";
declare buddy identity owner;
define myServer node peer.server("buddy@./socket");          # Unix domain
#define myServer node peer.server("buddy@127.0.0.1:12345");  # local
#define myServer node peer.server("buddy@0.0.0.0:12345");    # remote
myServer. define r1 on(a=1 and b=2);
```

Declare an identity `buddy` that you rank as `owner`. This is all the interpreter knows about `buddy`. Use this same name, `buddy`, in the specification of the peer server node. The peer node module requires that a key exist in the key store for this identity, which is why you created it in the previous section.

Let's start up the peer server.

```
$ ./server
2008/06/11 10:27:58 NB000I Argument [1] -d
2008/06/11 10:27:58 NB000I Argument [2] ./server
> #!/usr/local/bin/nb -d
> # File: tutorial/Peer/server
> -rm server.log
[20642] Started: -rm server.log
[20642] Exit(0)
> setout=".",log="server.log";
2008/06/11 10:27:58 NB000I NodeBrain nb will log to server.log
> declare buddy identity owner;
> define myServer node peer.server("buddy@./socket");          # Unix domain
2008/06/11 10:27:58 NB000I Peer keys loaded.
> #define myServer node peer.server("buddy@127.0.0.1:12345"); # local
> #define myServer node peer.server("buddy@0.0.0.0:12345");   # remote
> myServer. define r1 on(a=1 and b=2);
2008/06/11 10:27:58 NB000I Source file "./server" included. size=493
2008/06/11 10:27:58 NB000I NodeBrain nb[20641,19542]daemonizing
$
```

You are using a Unix domain socket in this tutorial because you don't need to communicate with remote clients. To experiment with serving remote clients, you can comment out the active `myServernode` and uncomment the remote myServer node. But first you should experiment with the client in the next section.

2.3 Peer Module Client Node

> *He who is always his own counselor will often have a fool for his client.* — Hunter S. Thompson (1937–2005)

The peer module enables NodeBrain to play the part of both server and client. This is not foolish because I am talking about different instances of NodeBrain, different processes ("skulls"), playing the roles of client and server.

The `client` file in the `tutorial/Peer` directory looks like this.

```
#!/usr/local/bin/nb
# File: tutorial/Peer/client
declare buddy identity;
define myClient node peer.client("buddy@./socket");          # Unix domain
#define myClient node peer.client("buddy@localhost:12345"); # local
#define myClient node peer.client("buddy@myhost.mydomain:12345"); # remote
myClient:assert a=1,b=2;
myClient:stop;
```

Notice you declare the identity `buddy` and specify the client just like you specified the server in the previous section. Here there is no requirement to give `buddy` local permissions.

Because you are going to run this client on the same machine as the server and under the same user account, the client and server will use the same key store. To run the client from a different account or machine, you would have to copy the `buddy` key from the server's key store to the client's key store. You could also configure different private keys for servers and clients and copy their public keys to the key stores of their peers. That approach provides better security, but quick success in this tutorial is more important, so stick with shared secret keys.

Now you can run the client and see what happens.

```
$ ./client
2008/06/11 10:28:00 NB000I Argument [1] ./client
> #!/usr/local/bin/nb
> # File: tutorial/Peer/client
> declare buddy identity;
> define myClient node peer.client("buddy@./socket");          # Unix domain
> #define myClient node peer.client("buddy@localhost:12345"); # local
> #define myClient node peer.client("buddy@myhost.mydomain:12345"); # remote
> myClient:assert a=1,b=2;
2008/06/11 10:28:00 NB000I Peer keys loaded.
2008/06/11 10:28:00 NB000I Peer b0000=buddy@./socket
2008/06/11 10:28:00 NB000I Rule myServer.r1 fired
> myClient:stop;
2008/06/11 10:28:00 NB000I Peer b0000=buddy@./socket
2008/06/11 10:28:00 NB000I Source file "./client" included. size=450
2008/06/11 10:28:00 NB000I NodeBrain nb[20644] terminating- exit code=0
$
```

The command `myClient:assert a=1,b=2` sends the command `assert a=1,b=2` to the server specified as `buddy@./socket`. Notice the message `Rule myServer.r1 fired`. There is no `myServer.r1` defined in the client. This is what happened at the server. When a peer client issues a command to a peer server, the server lets the client listen in as it reacts to the command. You can look at it from the server's point of view by displaying the agent log, in this case called `server.log`.

```
$ cat server.log

N o d e B r a i n   0.9.02 (Columbo) 2014-02-15

Compiled Jun 12 2014 19:20:12 x86_64-unknown-linux-gnu

Copyright (C) 2014 Ed Trettevik <eat@nodebrain.org>
MIT or NodeBrain License
------------------------------------------------------------------

nb -d ./server

Date       Time     Message
---------- -------- ------------------------------------------------
2014-06-11 10:27:58 NB000I NodeBrain nb[20643:1] myuser@myhost
2014-06-11 10:27:58 NB000I Agent log is server.log
2014-06-11 10:27:58 NM000I peer.server myServer: ...
... Listening for NBP connections as buddy@./socket
2014-06-11 10:28:00 NM000I peer.server myServer: buddy@./socket
> myServer. assert a=1,b=2;
2014-06-11 10:28:00 NB000I RulemyServer.r1 fired
2014-06-11 10:28:00 NM000I peer.server myServer: buddy@./socket
> myServer. stop;
2014-06-11 10:28:00 NB000I NodeBrain nb[20643] terminating - exit code=0
$
```

At 10:27:58 the server started listening for connections. At 10:28 it received a connection and issued the command `assert a=1,b=2` in the `myServer` context. This triggered rule `myServer.r1`, which has no action. In previous tutorials, you've learned enough to modify the rules in server to provide an action and try it again.

Notice the client sends a `stop` command to the server, which stops it because you've given the client full owner permissions. This means you have to restart the server each time you run the client. Only in a tutorial would you do something this silly.

2.4 Interactive Peer Module Client

If you want to experiment further with the peer module, restart the server and try the `iclient` file instead of `client`.

```
$ ./server
$ ./iclient
```

The `iclient` script looks like this.

```
#!/usr/local/bin/nb -‘myClient:
# File: iclient
declare buddy identity;
define myClient node peer.client("buddy@./socket");         # Unix domain
```

The strange looking she-bang ("#!") line has an argument that supplies an interactive command prefix and enables an option to automatically go into interactive mode after processing all command arguments. This step causes your prompt to look like this.

```
myClient:>
```

Any command you enter is now prefixed by the value `myClient:`, causing all your commands to be directed to your peer client node, which sends them to your peer server. Enter the highlighted text when prompted to get the same results as shown below.

```
$ ./iclient
2008/06/11 16:48:39 NB000I Argument [1] -'myClient:
> #!/usr/local/bin/nb -'myClient:
> # File: tutorial/Peer/iclient
> declare buddy identity;
> define myClient node peer.client("buddy@./socket");         # Unix domain
2008/06/11 16:48:39 NB000I Source file "./iclient" included. size=209
2008/06/11 16:48:39 NB000I Reading from standard input.
---------- --------
myClient:> show r1
2008/06/11 16:48:45 NB000I Peer keys loaded.
2008/06/11 16:48:45 NB000I Peer b0000=buddy@./socket
> myServer. show r1
r1 = ! == on((a=1)& (b=2));
myClient:> assert a=1,b=2;
2008/06/11 16:49:10 NB000I Peer b0000=buddy@./socket
> myServer. assert a=1,b=2;
2008/06/11 16:49:10 NB000I Rule myServer.r1 fired
myClient:> 'foo.
foo.> '
> quit
2008/06/11 16:49:15 NB000I NodeBrain nb[23715] terminating - exit code=0
$
```

This example illustrates how a single quote at the beginning of an interactive command can be used to change the command prefix.

The little tricks illustrated in this section are features of the interpreter, not the peer module, but when combined with the peer module make it a bit easier to use NodeBrain as a primitive interactive client to a NodeBrain agent.

3 Commands

This section describes commands used with the Peer modules.

3.1 Peer Server Skill

A peer server accepts connections and serves requests from other NodeBrain processes using compatible peer node modules. The server authenticates clients and issues commands sent from the client within the context of the server node. It simply accepts commands from remote nodes and passes them on to the interpreter using the clientß s authenticated identity.

3.1.1 Peer Server Definition

Syntax

`peerDefineCmd`
::= define $ term $ node [$ peerServerDef] "

`peerServerDef`
::= peer.server(ø serverSpec Æ);

`serverSpec`
::= serverIdentity `@` socketSpec

`serverIdentity`
::= name of server identity for authentication

`socketSpec`
::= inetSocket `|` localSocket

`inetSocket`
::= [hostname `|` ipaddress] : port

`localSocket`
::= filepath

The following example defines a peer server node that accepts connections from other servers on port 50000 using identity "fred."

```
define buddy node peer.server("fred@0.0.0.0:50000");
```

On servers that have multiple network interfaces, connections may be limited to a given interface by specifying the interface address.

```
define buddy node peer.server("fred@192.168.1.100:50000");
```

Access can be limited to the local host to avoid remote connections by specifying the loopback address.

```
buddy node peer.server("fred@localhost:50000");
```

Access can also be limited to the local host by using a local domain socket.

```
define buddy node peer.server(ø fred@/tmp/mysocketÆ );
```

The Assert, Evaluate, and Command methods are not implemented for this skill.

Enable and Disable methods are implemented, so the `ENABLE` and `DISABLE` commands can be used to control when the node listens for connections.

```
> disable term;      [Stop listening for connections.]
> enable term;       [Start listening for connections.]
```

3.2 Peer Queue Skill

A peer queue provides a file-based mechanism for passing data between application components. The design is intended to be simple, rugged, and independent. It is not intended for high performance.

The file structure and locking scheme is sufficiently simple to enable non-NodeBrain application components to directly write to or read from a peer message queue. However, it is normally best to use the nb program with the peer module to send and receive messages when using a peer message queue.

3.2.1 Peer Queue Definition

Syntax

`peerQueueDefine`
::= define $ term $ node [$ peerQueueDef] "

`peerQueueDef`
::= peer.queue(ø queueSpec Æ , schedule);

`queueSpec`
::= queue directory path

`schedule` ::= cell expression—normally a time condition

The following example would be used to process commands from a queue directory with a path off `/tmp/queue/automon`. This queue would be checked every 30 seconds for new commands.

```
define input node peer.queue("/tmp/queue/automon",~(30s));
```

3.2.2 Message Directory and File Names

A message file path has several components: queue, identity, time, count, and type.

```
queue/identity/time.count.type
```

The queue and identity components are specified when a peer is defined.

```
define term node peer("identity@(queue)");
```

It is important to understand that only file permissions can prevent an unauthorized user from writing a message under any given identity. It is best when producer and consumer processes run under the same user account.

The type component of a message file name identifies the type of content and file format.

'Q' Header

'q' command queue

'c' command

't' text

Each message directory (queue/identity) has a header file used to control message file names.

```
queue/identity/00000000000.000000.Q
```

This file contains a simple 21 byte record of the form show here, where ttttttttttt is a UTC time and cccccc is a counter.

```
ttttttttttt.cccccc
```

A command queue message file contains one command per line with a special character in the first position of each line.

'>' unprocessed command

'#' processed command

A command queue message file (.q) is processes by executing each command and marking it complete by replacing ">" with "#". This enables a consumer to terminate processing in the middle of a command queue message file and restart where it left off later.

A command message file (.c) is processed by interpreting each command without marking individual commands complete. This is just like including a ".nb" source file.

The peer node module does not yet have the needed functionality to process text message files (.t) and package message files (.p), although it can write them to a message queue to be processed by a custom consumer program. An example of a text message file is an email received by the Mail node module. These files can be transmitted to a message queue on another host for parsing. The structure of a package message file has not been defined. We are simply reserving this type code for future use. We expect it to be used for managing rule file updates or NodeBrain software updates. Header information will describe the package file content and required actions.

3.2.3 Message Queue Processing

When writing a command to a queue, the peer module references the header file to find the name of the current command queue message file. If the header file contains a value like the one shown below, the current command queue message file name is this same value with ".q" appended.

'01043454323.000000'
[content of header file]

'01043454323.000000.q'
[current message file name]

There are three conditions under which the current command queue message file name is "incremented." This happens when

1. the command queue interval expires
2. a message file of a different type is written
3. a consumer begins reading from a queue

Otherwise, commands are appended to the current command queue message file with a prefix of ">".

When a queue is specified in a peer definition, a command queue interval may be specified. This will cause the peer module to start a new command queue message file whenever a command is written after the current interval has expired. For example, if the command queue interval is set at 1 hour, at least one command queue message file will be created for each hour in which a command is queued.

Any time a text (.t), command (.c), or package (.p) message file is queued, the header is updated to provide a unique file name for the message file. The time component is set to the current time. If this matches the current command queue time, the count in the header file is incremented by 2 to skip over the count used by the message file. In other words, NodeBrain interrupts the current command queue message file to allow another message file to be queued in order.

When the peer module reads a message queue, it first updates the header record to force the start of a new commend queue message file. This is similar to the process described previously, except the counter is only incremented by 1 if the time component doesnß t change. The message queue directory is then processed only through the file name that was current before the update. Each message file is locked by producers and consumers to avoid simultaneous access. A consumer will skip over a busy file and catch it on the next pass.

If a system clock is reset to an earlier time, a peer message queue must preserve file sequence to avoid attempts to overwrite existing files. This is accomplished by never reducing the time in the header file. If the count in the header file exceeds 999999, the time component is incremented and the count is set to 000000. This preserves file name sequence. There is no dependence on correct times in a message queue. However, when a message queue has multiple directories (is written using multiple identities), the message files are processed in order by the messages file names. This is only important when the queue contains several files, perhaps because the consumer stopped for a long period. It enables the consumer to process message files in the same general order they were produced. The sequencing of

commands from different identities in a backlogged queue will depend on the granularity of the message queue interval.

3.3 Peer Client Skill

The peer client skill is used to communicate with a peer server and/or a peer queue.

3.3.1 Peer Client Definition

Syntax

`peerDefineCmd`
::= define $ term $ node [$ peerDef] "

`peerDef` ::= peer.client(ø peerSpec Æ [, schedule]);

`peerSpec` ::= [[clientIdentity ~] serverSpec [{ timeout }]] [(qSpec)]

`clientIdentity`
::= name of client identity to portray

`serverSpec`
::= serverIdentity @ socketSpec

`serverIdentity`
::= name of server identity (see peer.server)

`socketSpec`
::= inetSocket | localSocket

`inetSocket`
::= [hostname | ipaddress] : port

`localSocket`
::= filepath (Must include at least one "/" - use "./" if necessary)

`qSpec` ::= qDirectoryPath [(qInterval)]

`schedule` ::= See Time Conditions

A peer node can be used to perform four different functions, depending on how it is defined and what command options are used.

1. Send commands to a remote peer server node.
2. Store commands in a peer message queue.
3. Store-and-forward command to a remote peer server node through a peer message queue.
4. Process commands from a peer message queue.

Suppose a NodeBrain agent running on yoyo.com has the following peer server node.

```
define server node peer.server("charlie@0.0.0.0:32171");
```

A NodeBrain client with the following peer node is able to send commands to the agent, provided the identities "sally" and "charlie" are properly defined on both systems.

```
define client node peer("sally~charlie@yoyo.com:32171");
```

3.3.2 Peer Client Command

Syntax

`peerCmd` ::= context[(option)][: text] "

`option` ::= 0 | 1 | 2 | 3

A peer client command is used to send a single command to a peer server, directly, or indirectly via a store and forward queue.

If no option is specified, the operation is determined by the combination of server and queue specified.

Option	Server	Queue	Operation
null	No	yes	Command text is written to queue.
null	Yes	yes	Command text is written to queue and then queue is forwarded if server is available.
null	Yes	no	Command text sent directly to server

If an option is specified, the operation is determined by the option, unless the requirements are not met, in which case the option is ignored and the operation is determined as shown above.

Option	Server	Queue	Operation
0	Allowed	Required	Command text is written to queue.
1	Required	Required	Command text is written to queue and then queue is forwarded if server is available.
2	Required	Allowed	Command text sent directly to server
3	Required	Allowed	An interactive session is established with a new ø skullÆ spawned by the server.

3.4 Peer Service Skill

The service skill supports commands related to peers that are not specific to individual client and server nodes.

3.4.1 Peer Service Definition

Syntax

`peerServiceDefine`
::= define $ term $ node [$ peerServiceDef] "

`peerServiceDef`
::= peer.service;

3.4.2 Identify Command

Syntax

`identifyCmd`
::= node: identify $ identityName [$ bitSize] [$] [; [comment]] "

`node` ::= peer service node term

`bitSize` ::= integer

The `IDENTIFY` command is used to generate a random private identity key. By default a 64-bit key is generated but you may override the bit size with any positive integer value. The generated identity key is placed in your home directory in a file named ø nb_peer.keysÆ . If the identity name is not already specified, a new entry is appended.

```
identify lilly 32;
```

The command above would generate a relatively insecure private key and append an entry like the following to ~/.nb/nb_peer.keys

```
lilly 7.f64e1e7f3.f20707fa2.0;
```

The structure of the key is as follows, where owner is generated as 0 and otherwise not used by NodeBrain.

```
pubExponent.pubModulus.privateExponent.owner
```

To protect your private keys, it is important to set nb_peer.keys file permissions to allow read/write by owner only. NodeBrain is not intended for use in publicly shared applications, so there is no requirement for a public key server. You may construct a public key from your private key by replacing the third number with 0.

```
lilly 7.f64e1e7f3.0.0;
```

A NodeBrain peer public identity key should be managed as a shared secret key. Share it only with trusted administrators of peers requiring communication with your NodeBrain application.

3.4.3 Copy Command

Syntax

`peerCopyCmd`
::= node : copy (copyToFileSpec | copyToQueueSpec) [; [comment]] "

`node` ::= peer service node term

`copyToFileSpec`
::= { a | b } $ peerFileSpec $ peerFileSpec

`copyToQueueSpec`
::= { c | q | t } $ peerFileSpec $ peerQueueSpec

`peerFileSpec`
::= [clientNode :] filename

`peerQueueSpec`
::= [clientNode :] queueNode

`clientNode`
::= peer client node term

We include documentation of the deprecated `COPY` command here to support a transition to the peer copy skill.

The `COPY` command is used for secure file transfers using NodeBrain peer authentication and data encryption.

Event monitoring applications often require file transmission for application administration (e.g., rule file update) and bulk event transfers. For administration, NodeBrain provides ascii (text) and binary file transfers. The first parameter to the `COPY` command is "a" for ascii or "b" for binary.

```
copy a /tmp/newrules.nb @goofy:/tmp/newrules.nb; # ascii file transfer
copy b /usr/local/bin/nb @goofy:/tmp/nb; # binary file transfer
```

The second and third parameters specify the source file and destination file respectively. A NodeBrain agent may be used to push or pull a file. When pulling a file, a brain is specified for the source file.

```
copy a @goofy:/opt/mymonitor/rules.nb /tmp/rules.nb; # pull
```

You may also copy files between two remote servers by specifying NodeBrain agent names on both the source and destination.

```
copy a @huey:/opt/mymonitor/rules.nb @duey:/tmp/rules.nb;
```

Use of the NodeBrain `COPY` command for application administration is optional. This feature is included for environments where other secure tools are not available.

The `COPY` command is also used to place a copy of a file into a local or remote NodeBrain message queue for processing. The first parameter specifies the type of message file to create.

‘c’ command

‘q’ command queue

‘t’ text

When copying to a queue, the destination is a brain name instead of a file name. NodeBrain will generate a unique file name in the target message queue directory.

```
copy q my_command_set.nb goofy
```

The message queue directory name is derived from the brain definition and the currently portrayed identity.

```
declare goofy brain silly@goofus.nodebrain.org:49828[/var/spool/nb];
portray huey;
copy q my_command_set.nb goofy
```

Given the brain declaration and identity shown above, the copy command would create a message queue file as follows, where ttttttttttt is the current time (UTC) and nnnnnn is a unique number for files within that second and "q" is the message file type.

```
/var/spool/nb/goofy/huey/ttttttttttt.nnnnnn.q
```

Like administrative file transfers, message queue file transfers may have a remote source and/or destination.

```
copy t @agent1:/tmp/my_command_set.nb @agent2:goofy
```

This feature is provided for convenient transmission of event data from a source system to a monitoring system. You may use other tools like ssh and scp for this purpose if preferred in your environment.

4 Triggers

This module does not implement triggers.

Index