NodeBrain Guide

Release 0.9.03

NodeBrain Guide
December 2014
NodeBrain Open Source Project

Release 0.9.03

Author: Ed Trettevik

History

2014-02-16 Title: *NodeBrain Guide*
Author: Ed Trettevik <eat@nodebrain.org>
Publisher: NodeBrain Open Source Project

Release 0.9.03

- This document replaces *NodeBrain User Guide* first published in 2003.
- Converted to texinfo format
- Reorganized existing content and added content; e.g. *Security* and *Performance* chapters.
- The *Sample Scripts* chapter has been dropped. The *NodeBrain Tutorial* is now a better source of examples.

Preface

This guide is intended for readers seeking a general understanding of NodeBrain. Other documents are available for readers looking for a more complete understanding of the rule language, modules, C API, and kits.

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

Caboodle NodeBrain Kit - A framework for managing rules
System NodeBrain Kit - A small sample application

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

NodeBrain is a rule engine and related components intended for the construction of state and event monitoring applications. The rule engine interacts with other monitoring components to perform simple element state monitoring or complex event correlation based on user specified rules. The language is extended through the development of node modules—plugins that use NodeBrain Library functions to implement a node and interact with the rule engine. Functionality is further extended by servants—programs written in any language that interact with NodeBrain via stdin, stdout, and stderr. A small set of plugins are distributed with NodeBrain to provide commonly needed features such as peer-to-peer communication, log monitoring, event caching, and a web browser interface for administrators.

1.1 Rule Language

The NodeBrain rule engine is an interpreter of the NodeBrain rule language. The language is mostly declarative rather than procedural. This means you define a set of conditions and response actions without fussing over the order in which the conditions are evaluated or the order in which the actions are taken.

Rule conditions are organized as a hierarchy of cells. Each cell has a value and a formula that references other cells for computing the value. A constant cell is a special case where the formula references only constant values.

The following example could be used to monitor the temperature of a CPU in Celsius. This example is only intended to give you a sense of the language, so don't worry about understanding every detail.

```
define cpu node;
cpu. define temp      cell;    # Current CPU temperature
cpu. define tempMax   cell 90; # Maximum normal temperature
cpu. define tempReset cell 80; # Reset temperature for "hot" episode
cpu. define tempAlert on(temp>tempMax ^ temp<tempReset):$alarm("Hot CPU");
```

Once these rules are loaded into the engine, the state of the monitored element, CPU temperature in this case, must be reported to the rule engine. However the temperature is actually obtained, it is reported to NodeBrain as an assertion.

```
cpu. assert temp=75; # update the current cpu temperature
```

This example also allows for adjusting the alarm threshold and episode reset threshold.

```
cpu. assert tempMax=95,tempReset=85;  # change alarm thresholds
```

In this example rule set, the value of `tempMax` determines the temperature at which an alarm is generated and an episode begins. The value of `tempReset` determines the temperature at

which the episode is considered complete. This prevents the alarm from triggering multiple times when the temperature bounces up and down over and below the upper threshold.

Rule files can be run as scripts by starting the file with a `#!/usr/bin/nb` shebang and setting the executable file permission. They can also be run by issuing an `nb` command with the file name as an argument. We sometime refer to a NodeBrain rule file as a NodeBrain script, but this is not intended to imply a procedural language like most scripting languages.

1.2 Agents

An agent is a NodeBrain script that runs as a daemon to monitor one or more elements or event streams. This is the primary use of the rule engine, although it can also run in the foreground as a batch job, or as an interactive program accepting commands from a terminal.

1.3 Caboodles

A *caboodle* is a directory that contains files used by a NodeBrain application. The rule engine does not dictate the structure of subdirectories in a caboodle, but you are advised to follow the structure defined by the *Caboodle NodeBrain Kit*. By putting all files associated with a NodeBrain application in a single directory and using relative file paths when referencing one from another, your application is relocatable and you minimize interference with other NodeBrain applications. You can easily host multiple NodeBrain applications on a single server, and multiple instances of a NodeBrain application over multiple servers. If built properly, a caboodle can be archived using `tar`, and replicated by extracting the tar file in a different location on the same host, or in any location on another host.

The Caboodle Kit provides commands for operating on a caboodle. However, you are not required to use the Caboodle Kit, nor are you required to conform to the caboodle model. It is just a recommendation.

Under the caboodle model, executable NodeBrain scripts use a `#!bin/nb` shebang, where bin/nb is the relative path within the caboodle of a symbolic link to the desired version of `nb`; e.g. '`/usr/bin/nb-0.8.16`'. This enables the timing of upgrades to be managed separately for each caboodle.

1.4 Servants

A *servant* is a program or script (written in any programming language) that runs as a child to a NodeBrain script and communicates via stdin, stdout, and stderr. A servant may receive input from NodeBrain on stdin, send commands to NodeBrain on stdout, and send lines to the NodeBrain log file on stderr.

A servant may be used to dynamically generate rules, provide assertions and alerts that update the state of rule cells to which rules respond, or implement external actions.

When using the caboodle model, a servant is stored in the '`servant`' subdirectory and can assume the current working directory is the caboodle root directory. References to all other components should be made via paths relative to the caboodle root directory.

1.5 Modules

A *module* is a plugin to the rule engine that provides added capabilities using a C API provided by the NodeBrain Library. Unlike servants that are loosely coupled, modules tightly couple with the rule engine. This has both advantages and disadvantages. The obvious disadvantages are: 1) modules require more effort to build, and 2) modules may require a rebuild or even revision to take advantage of upgrades to the NodeBrain Library or the OS. The advantages are: 1) modules can provide better performance in many cases, and 2) modules can participate as cells, not just as command processors and generators.

1.6 Kits

A NodeBrain *kit* is a collection of components (rule files, servants, etc.) used within a caboodle for a particular NodeBrain application. Kit developers may use the framework provided by the Caboodle NodeBrain Kit, or may develop their own framework.

The rule engine, `nb`, provides a hook for implementing kit frameworks. This is provided by the `nbkit` command, which is just an alias for `nb`. The `nbkit` command lets you give short names to your caboodles. These caboodle names normally identify the application.

```
... define the bobo caboodle to nbkit
$ nbkit bobo link $HOME/bobo
```

A kit framework can implement any set of commands for operating on a caboodle. This only requires creating an `nbkit_` executable and placing it in the '`bin`' subdirectory of the caboodle. The `nbkit` command provided by the rule engine can then be used to invoke kit framework commands that can assume the caboodle root directory is the current working directory. This means frameworks can be provided entirely within caboodles without requiring the installation of additional packages on the system.

```
... The nbkit command you issue from any current working directory
$ nbkit bobo command arguments
... The command nbkit issues with $HOME/bobo as current working directory
$ bin/nbkit_ bobo command arguments
```

The Caboodle NodeBrain Kit provides a framework using this scheme. You can use it, or develop your own kit framework.

1.7 Plans

A NodeBrain `plan` is a document that defines a set of rules in an abstract representation not known to the rule engine. The Caboodle NodeBrain Kit uses XML documents conforming to a specific schema to represent plans. Another framework could use a different XML schema, or another document format not based on XML. Plan compilers are required to translate a plan into a set of rules in the syntax of the NodeBrain rule language. This approach to managing NodeBrain rules enables rule developers to work at a higher level of abstraction, and provides leverage for adapting rules to take advantage of new rule engine features without updating plans.

1.8 C API

The NodeBrain Library provides the rule engine and C API functions that can be used to embed the rule engine in a C program or create a plugin to the rule engine. The nodebrain package includes the library, the `nb` command which is actually just a small main routine the uses the C API to start the engine, and a collection of plugins that use the C API to extend the engine.

The C API is described by the NodeBrain Library manual.

2 Installing

NodeBrain is packaged using the GNU Build System—autoconf, automake, and libtool. The make file supports the creation of Linux RPM files with a simple `make rpm` command, and files are included for building Debian based packages (not yet as simple as `make deb`).

2.1 Installing from Package Repositories

The easiest way to install NodeBrain is from a package repository when available. It is not currently included in major Linux distributions, but may be pulled from other repositories. If you work for a company or organization that has internal package repositories, check to see if nodebrain is available.

2.1.1 yum

On GNU/Linux distributions that use the RPM packaging system, you may be able to install NodeBrain using `yum`. The chance is slim currently that NodeBrain is in one of your configured repositories, but when it is, this is the best option for installing.

```
# yum install nodebrain
```

If you are experimenting on a test machine and feel comfortable installing from an unofficial repository, you can browse to the following url for instructions on how to install from the author's repository at software.opensuse.org.

http://software.opensuse.org/download.html?project=home:trettevik&package=nodebrain

2.1.2 apt-get

On GNU/Linux distributions that use the Debian packaging system, you may be able to install NodeBrain using `apt-get`. Currently NodeBrain is not likely to be available from your configured repositories, but this is your best option when it works.

```
# apt-get install nodebrain
```

If you are experimenting on a test machine and feel comfortable installing from an unofficial repository, you can browse to the following url for instructions on how to install from the author's repository at software.opensuse.org.

http://software.opensuse.org/download.html?project=home:trettevik&package=nodebrain

2.2 Installing from GNU Source Distribution File

This method is appropriate if you have a build machine where the needed development packages exist or can be installed.

Browse to http://nodebrain.org and select [Downloads] on the left side menu. Then click on the Rule Engine nodebrain-*version*.tar.gz file link to download the GNU source distribution file from SourceForge.net.

2.2.1 Installing as root to /usr/local

As root, issue the following command on the downloaded tar.gz file.

```
# tar -xf nodebrain-version.tar.gz
# cd nodebrain-version
# ./configure
# make
# make check
# make install
```

2.2.2 Installing to Your Home directory

If you don't have root access on the box, and want to install NodeBrain in your home directory, specify a `DESTDIR` option on the `make install` command.

```
$ tar -xf nodebrain-version.tar.gz
$ cd nodebrain-version
$ ./configure
$ make
$ make check
$ make install DESTDIR=$HOME
$ export PATH=$PATH:$HOME/usr/local/bin
$ export NB_MODULE_PATH=$HOME/usr/local/lib/nb-<spec-level>>
```

If you have success installing to your home directory and the `nb` command works, add the two export commands to your profile so they are applied each time you login.

2.2.3 Excluding OpenSSL and Dependant Features

If you don't need any NodeBrain features that depend on OpenSSL (e.g. Webster and Message modules), you may exclude these feature from the build by specifying `--without-tls` on the `./configure` command. Use it in combination with whatever other arguments you specify.

```
$ ./configure --without-tls
```

2.2.4 Excluding Node Modules (Plugins)

To exclude any of the modules provided by the NodeBrain package, include `--disable-nb_`*module* as an argument to `./configure`. For example, to exclude the Peer module, the command would look as follows.

```
$ ./configure --disable-nb_peer
```

To get a complete list of `./configure` options, use `--help`.

```
$ ./configure --help
```

2.2.5 Creating an RPM file

To make an RPM to be installed with `yum` or `rpm` on another system, simply issue `make rpm`. This will invoke an `rpmbuild` command.

2.2.6 Creating a GNU Binary Distribution file

If you need to install on a machine that doesn't use the RPM or Debian packaging systems, and you are not able to use the native packaging system, you can use a GNU binary distribution file.

```
$ tar -xf nodebrain-version.tar.gz
$ cd nodebrain-version
$ ./configure
$ make
$ make check
$ make install DESTDIR=$HOME/nodebrain
$ cd $HOME/nodebrain
$ tar -czf nodebrain-version-platform.tar.gz *
```

2.3 Installing from GNU Binary Distribution File

If you have a GNU binary distribution file created as shown in the previous section for the target platform from source, you can install it on a like system by simply extracting the tar file.

2.3.1 Installing as root to /usr/local

To install to /usr/local as root, change to the root directory and extract the file.

```
# cd /
# tar -xzf nodebrain-version-platform.tar.gz
```

2.3.2 Installing to User Home Directory

To install to the home directory of a regular user, change to the home directory and extract the file. Then set the PATH and NB_MODULE_PATH environment variables to execute from your home directory.

```
$ cd ~
$ tar -xzf nodebrain-version-platform.tar.gz
$ export PATH=$PATH:$HOME/usr/local/bin
$ export NB_MODULE_PATH=$HOME/usr/local/lib/nb
```

If you have success installing to your home directory and the `nb` command works, add the two export commands to your profile so they are applied each time you login.

2.4 Installing from Git Repository

You can download the NodeBrain git repository from SourceForge or GitHub using the protocol that works best for you.

```
$ git clone http://git.code.sf.net/p/nodebrain/nb nodebrain-nb
  -or-
$ git clone git://git.code.sf.net/p/nodebrain/nb nodebrain-nb
  -or-
$ git clone https://github.com/trettevik/nodebrain-nb.git nodebrain-nb
  -or-
$ git clone git@github.com:trettevik/nodebrain-nb.git nodebrain-nb
```

After downloading the repository you can checkout the version you want to build using the version tag. Skip this step if you want to build from the latest code on the master branch. Then execute the additional commands shown below starting with `./autogen.sh`. You will need the autoconf, automake and libtool packages installed. The `./autogen.sh` command will install some "autostuff" components that are included in the GNU source distribution tar.gz file, but not included in the source repository. It will then build the `configure` script. From there you build using any of the options discussed above for building from a source distribution file, staring from the `./configure` command.

```
$ git tag
$ git checkout version
$ ./autogen.sh
... use any build options you want from here ...
$ ./configure
$ make
$ make check
$ make install
```

3 Running

The `nb` program may be invoked a variety of ways. This chapter describes some of the options.

3.1 Command Line

The `nb` program may be run at the command line using one or more command files.

```
$ nb cmdset1.nb cmdset2.nb
```

3.2 Script

To create a NodeBrain script, place a *shebang* line at the top of a NodeBrain command file and give it the executable file permission.

```
#!/usr/bin/nb
...commands...
```

If the file above is called '`cmdset.nb`', you can execute it like any other executable script.

```
$ ./cmdset.nb
```

3.3 Relocatable Caboodle

Within a caboodle, you should use relative file paths so your application can be easily cloned or moved. We can assume that all commands within a caboodle are executed with the working directory set to the root directory of the caboodle. You can also assume that bin/nb is a symbolic link to the version of nb we want to used within the caboodle.

Now invoking `nb` at the command line, perhaps within a script, looks like this.

```
$ bin/nb cmdset1.nb cmdset2.nb
```

Similarly, a script designed to run in a caboodle looks like this.

```
#!bin/nb
...commands...
```

3.4 Daemon

To run `nb` as a daemon (background process), use the `-d` or `--daemon` option.

On the command line it looks like this.

```
$ nb -d cmdset1.nb cmdset2.nb        [Outside a caboodle]
  -- or --
$ bin/nb -d cmdset1.nb cmdset2.nb    [Within a caboodle]
```

In a script it looks like this.

```
#!/usr/bin/nb -d   [Outside a caboodle]
  -- or --
#!bin/nb -d        [Within a caboodle]
...commands...
```

When started as a daemon, `nb` first executes all the referenced commands, which usually results in defining a set of rules. It then moves to the background, enables nodes, and then switches users if starting as root and a different user was specified. If networking nodes are included in the rules, they may establish connections or listening sockets when enabled. The daemon is not attached to a terminal and writes output to a configured log file.

3.5 System Startup

See the *System NodeBrain Kit* manual for information on running `nb` at system startup time. The System Kit provides a startup script, `nodebrain.service`, and sysconfig file `nodebrain.sysconfig` that can be used to implement any number of agents that start when the system boots up. [Those components will probably be moved to the base nodebrain package as nb.service and nb.sysconfig in a future release.]

Use the `--user=`*user* option to avoid running your agent as `root` when possible. When running an agent as `root`, you should not use modules that enable remote connections.

4 Rule Engine

NodeBrain rule syntax is covered in the *NodeBrain Language* manual. This chapter is intended to provide a mental picture of how the rule engine works.

4.1 Spreadsheet Analogy

Imagine the cells of a spreadsheet. Each cell can hold a constant value, or a formula that computes a value from the values of other cells. Cells are referenced by absolute or relative row and column. When you update the value or formula in a cell reference by other cells, the values of the other cells are evaluated. If their values change, the values of cells referencing them are evaluated. This continues until there are no more cells to evaluate. Since most people have used a spreadsheet program, this model should be a familiar one.

NodeBrain also has cells that can have a simple value or a formula for computing a value based on the value of other cells. Unlike a spreadsheet, NodeBrain cells are identified by their formula, not by rows and columns. Multiple references to the same formula are references to the same cell. NodeBrain also has special cells called *terms* that reference a single cell, providing a name to reference a cell.

Updates to cells are made via `assert` and `alert` commands, which update the value or formula associated with terms. This is like updating cells in a spreadsheet, and NodeBrain's response to these changes is also like a spreadsheet.

A *rule* is another special type of cell. Like a term, a rule cell references a single cell, but also has an action. When the referenced cell transitions to a true state, the rule "fires" and the action is taken.

4.2 Expert Analogy

The spreadsheet analogy above applies when NodeBrain is running in the most common mode, where it simply reacts to new information provided by assertions and alerts. However, NodeBrain can also operate is a different mode that is more analogous to an expert system. In this mode, NodeBrain tries to solve for unknowns until every rule has been decided. In NodeBrain's system of logic, cells can have true, false, unknown, and disabled values. To solve for unknowns, rules must be augmented to provide commands that NodeBrain can issue to find the value of terms whose value is unknown. Like an expert, NodeBrain will not invoke commands to solve for unknown terms when their value is not required to solve any rule condition. For example, if a rule has a condition `A or B` and both `A` and `B` are unknown, NodeBrain will first issue the command to solve for `A`. If true, the value of `B` is not required to know the value of `A or B` is true.

This approach is similar to that of a doctor diagnosing a patient's illness, or a car mechanic troubleshooting a car problem. In both cases, the most expensive tests are avoided if the less expensive tests can provide an answer. A fun way to experiment with this mode is to write a set of rules that describe types of trees or animals based on different values for common terms representing attributes. Then run the rules in interactive mode, where NodeBrain will prompt for the values of unknown terms until it identifies the type of tree or animal described by your answers.

The modes that correspond to the spreadsheet and expert analogies can be used in combination. Rules running in reactive mode may detect a condition that cause a diagnostic rule set to be launched to troubleshoot the cause of the condition.

4.3 Inputs and Outputs

Without plugin modules, the rule engine has only two ways to obtain new state or event information or to take external actions in response: 1) by sourcing rule files with optional arguments, and 2) by invoking shell commands. A servant script or program can be invoked as a shell command to either: 1) obtain new state or event information and report it to the rule engine as commands on stdout, or 2) take external actions in response to new state or event information.

Modules can implement additional ways to obtain new information and/or respond. For example, the Syslog module enables NodeBrain to receive syslog messages and parse them, translating the foreign text into NodeBrain commands. The syslog module also provides an interface for NodeBrain to send syslog messages in response to monitored conditions. A Pipe module enables external scripts to report state changes or events to the rule engine via a named pipe. It also enables the rule engine to send messages to external processes via named pipes.

With the ability to write servants and modules to provide interfaces to the monitored environment, there should be no limit to the extensions that can be built to enable the rule engine to monitor the state of any set of elements or any event stream. That is not intended to imply the NodeBrain rule engine will always be the best choice for a monitoring application—only that it has the flexibility to adapt to a wide variety of monitoring applications.

4.4 Check Scripts

A NodeBrain check script is used for regression testing new releases and verifying builds, ports and installs. You may also find this feature useful for regression testing custom node modules. One first builds a rule file to test a set of features. The `show` command is used as necessary to make sure internal results are as expected. Then the test script is run in check mode to create a check script by prefixing the file name with a tilde, `~`.

```
$ nb ~testname
```

The generated check script is named *testname*`~`. This script is a copy of the one executed, but with check lines included. A check line is identified by a `~` in column 1. The check operation is specified in column 2.

```
' ' - Match column 3 to end of line with a line of output.

        e.g. "~ text to match goes here"

'^' - Reset the check buffer as if we matched everything.

        e.g. "~^ "

'#' - Check comment---no operation

        e.g. "~# Checking to make sure ..."
```

A check script is run in check mode to make sure we get identical results later, based on output.

```
$ nb -b ~testname~
```

The bail option "-b" in check mode will only bail out on the first check error, not on a normal error. So a check script can verify that normal error messages are produced.

A check line is "hidden" when a check script is run in normal mode, and displays when running in check mode.

If you write custom node modules, experiment with this feature as a simple method of performing regression tests. Check out the samples provided in the check directory of a NodeBrain distribution.

4.5 Cron Schedules

NodeBrain supports time expressions which create cells that are true during scheduled intervals of time, and false outside the scheduled intervals. Schedule rules specified in a crontab, the Unix system scheduling daemon, can be translated into schedule rules in the NodeBrain language. This section explains how to perform the translation.

Utility	**Configuration File Format**
crontab	*minute hour mday month wday command*
NodeBrain	define *term* on(~(*timeExpression*)):=*command*

A *timeExpression* is composed using time functions. The following table shows how each cron field translates into a NodeBrain scheduling function.

	minute	**hour**	**mday**	**month**	**wday**
Range	0-59	0-23	1-31	1-12	0-6
Value	*m*	*h*	*d*	*n*	*x*
Function	m(*m*)	h(*h*)	d(*d*)	n(*n*)	su,mo,tu,we,th,fr,sa

The values in crontab fields can be in any of the following forms, and translate into NodeBrain scheduling function attributes as shown.

crontab	NodeBrain	Description
x	x	A specific value
x,y,z	x,y,z	A set of values
x-y	x..y	All values from x to y inclusive
*		All possible values

To form the full NodeBrain time expression for a given crontab entry, string the equivalent scheduling function for each field together, left to right, separating the functions with a period. This is illustrated in the following table with examples.

minute	hour	mday	month	wday	Time Expression
0	0	*	1-5	*	~(m(0).h(0).n(1..5))
10	*	*	*	*	~(m(10))
0	0	*	*	0	~(m(0).h(0).su)
20	4	1-15	*	*	~(m(20).h(4).d(1..15))
5	10	1,7,9	*	1	~(m(5).h(10).d(1,7,9).mo)
5	10,14	1,7,9	*	1	~(m(5).h(10,14).d(1,7,9).mo)

NodeBrain scheduling expressions enable possibilities not supported by cron as illustrated by the following example.

```
# 14:00 on the Tuesday of the week of the last Friday of the month
define projectReview cell ~(h(14).tu.w.fr[-1]n);
```

Since NodeBrain schedule conditions are cells, objects that have a formula and value, they can be combined with other cell expressions to form a more complex condition. In the following example we include a reference to the number of items scheduled for review. A reminder is sent to all team members only if there is more than one item to review.

```
# 14:00 on the Tuesday of the week of the last Friday of the month
define projectReview cell ~(h(14).tu.w.fr[-1]n);
define projectReviewNotice on(projectReview and items4Review>1)\
  :$RemindTeam("Project Review Starting")
```

See the *NodeBrain Language* manual for more information on time conditions.

5 Networking

NodeBrain may be used by an application that does not require network communication. If this is the case for your application, no need to read this chapter.

However, the *Node* in the name NodeBrain is based on both internal and external notions. There are rule nodes within a NodeBrain agent, and an agent may be a node within a network of agents. A network of agents may be distributed over multiple servers, requiring network communication.

In the earliest versions of NodeBrain, the notion of running as a network of agents was so much the intent that a form of communication now implemented by the Peer module was built-in to the interpreter, and had special supporting commands. That eventually seemed like a bad idea, and the functionality was moved to the Peer module as just one option available, and one that relied on the same command syntax provided for every other module.

You are free to use any method of network communication available to address the needs of your application. This may be done using a module provided with NodeBrain, or by using available alternatives.

5.1 Networking Modules

There are modules provided in the nodebrain package that support network communication of various types. Some are intended to connect an agent to other applications using standard protocols. A couple, Peer and Message, are intended for communication between two or more NodeBrain processes. Because these do not implement standard protocols (although based on standard lower level protocols) you should not attempt writing your own code to communicate with them. Such an attempt would not be harmful, just wouldn't be a productive use of your time. Instead, consider using these modules for their intended internal purpose, and use modules that implement standard protocols for communication with external components.

You can review the list of modules online at http://nodebrain.org to see if any meet your needs. If not, you might consider writing your own module if tight integration with NodeBrain is required. Where looser integration is sufficient, it is normally easier to implement.

5.2 Networking Servants

You can write servants in any language you like to enable network communication with a NodeBrain agent. Your servant can communicate with NodeBrain via stdin, stdout, and stderr, while communicating with a remote application component via any method you choose to implement. Please consider the security risks when doing this, and select a protocol that meets your authentication and encryption requirements.

5.3 SSH and SCP Commands

Don't overlook the possibility of addressing your networking requirements with `ssh` and `scp` commands. You can invoke these commands directly in NodeBrain rule files, and from within scripts invoked by rule files.

5.4 SSH Tunnels

An SSH tunnel may be helpful to secure network communication that uses an insecure protocol. The following command establishes a tunnel between ssh daemons on the local host and host B. A connection to port A on the local host is serviced by a connection to port B on host B.

```
ssh -2 -f -N -L portA:localhost:portB hostB
```

5.5 IPSec Tunnels

You may also consider using an IPSec server-to-server tunnel as a method of securing communication that would otherwise be insecure.

6 Security

Because NodeBrain can issue any shell command and supports a variety of methods of accepting shell commands from external sources, one can easily introduce security vulnerabilities. It is important to understand the potential risks and take steps to avoid or reduce them.

6.1 Code Scanning

The Coverity Scan service is used to perform static analysis on NodeBrain source code. This does not ensure defect free code, but enables many defects to be discovered and resolved before release, often defects that would not be discovered by other forms of testing.

If you write custom NodeBrain modules, you should scan your code with the best source and/or binary scan tools available to you. This will reduce the number of security vulnerabilities your modules introduce to your application.

6.2 File Permissions

Because NodeBrain reads rules from files and executes shell commands specified in these files, it is important to manage the file permissions on your rule files to minimize the security risk. It is best to have all files within a caboodle owned by a single user, with permission to write granted only to the owner. This requires that all NodeBrain agents associated with a caboodle run as the owning user.

6.3 Identity and Rank

Because NodeBrain can receive commands from outside a caboodle via node modules, it uses *identity* and *rank* to control access to rules within memory. Every node is owned by an identity and every command is issued by an identity. Node modules that accept commands from outside the caboodle are required to associate each command with an authenticated identity. Updates to cells are restricted to the owner of the node containing the cell, or identities ranked with global permission to update cells. Commands issued to a node are also restricted to the owning identity or identities with global permission.

Warning: This control mechanism has not been rigorously tested. Before relying on this feature, you should test it with your own rule set. If you discover a bug or weakness in this feature, please report it by sending an email to bugs@nodebrain.org.

6.4 Module Security

Modules can introduce security risks, particular those that accept commands over the network or from other users on the local system. Read module documentation and follow any security recommendations.

6.5 Root Agents

If you run an agent under the root user, you should not use any module that enables commands to come from outside your caboodle. All rule files used by the agent should be owned by root and either read-only or write restricted to root.

You may be able to avoid running an agent as root by using sudo to issue commands requiring root permissions. This enables you to use the sudo configuration file to limit the commands that can be issued with root permissions by your agent.

For agents that need to listen to restricted ports below 1024, you can use the `--user` option to start as root and then switch to another user after the listening sockets have been established.

7 Performance

NodeBrain rule engine performance varies significantly depending on your rule set and the computing resources provided by the host system. The numbers listed in this chapter provide only a rough estimate of the performance to expect. The measures reported here were obtained on a virtual private server, so this is unscientific. If you have dedicated hardware and skills in performance testing, please consider contributing your own results.

7.1 Test Platform

The following platform was used to obtain the test results shown in this document.

OS	CentOS 6.4 GNU/Linux, Kernel 2.6.32
CPU	Intel Xeon E5645 2.40GHz
RAM	2GB

Because NodeBrain is not multi-threaded, performance tests using a single NodeBrain process will not improve with multiple processors. These measures are based on the utilization of a single processor core. In a real NodeBrain application where high performance is needed, the monitoring problem is distributed over multiple NodeBrain processes which collectively make use of the available CPU's.

7.2 Test Conditions

To measure only the performance of the rule engine, output was suppressed during tests to minimize disk activity. This is done with the `use (hush)` directive.

Rule files of various sizes for different rule set characteristics are generated by a script distributed as '`bin/nbgenperf`' within the build directory of the GNU source distribution starting in release 0.9.01. For each rule set, a set of `assert` or `alert` commands is generated so that each command will trigger one of the rules in the set. Similar command files are generated where no assertion triggers a rule, and where every assertion reports unchanged values. Predictably, assertions that don't fire rules or don't change values have better performance than rules that change values and fire rules. While all of these cases are important to test, not all are equally interesting, so only the results for the rule-triggering command sets are reported here.

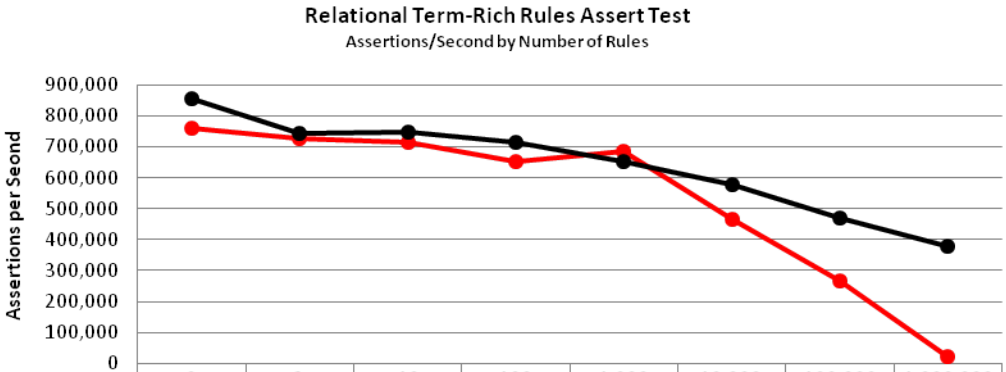
In this set of tests, a flat name space has been used. In other words, rules are not partitioned into multiple nodes. The partitioning of rules and terms into nodes generally improves performance.

7.3 Relational Term-Rich Rules Assert Test

The *relational term-rich rules assert* test uses `assert` commands on a set of `on` rules with far more terms than values in relational conditions.

```
# Rules - where X=0,1,2,...,N-1
define rX on(!aX and bX<>"X");
# Assertions - 4 million times, where X=0,1,2,...,N-1 repeating
assert !aX,bX="X+1";
```

The number of rules, `N`, is shown on the horizontal axis in the chart below.



	0	2	10	100	1,000	10,000	100,000	1,000,000
0.8.15	761,904	727,272	715,563	653,594	686,106	465,657	268,276	20,369
0.9.00	854,700	744,878	749,063	713,012	653,594	577,200	468,933	380,589

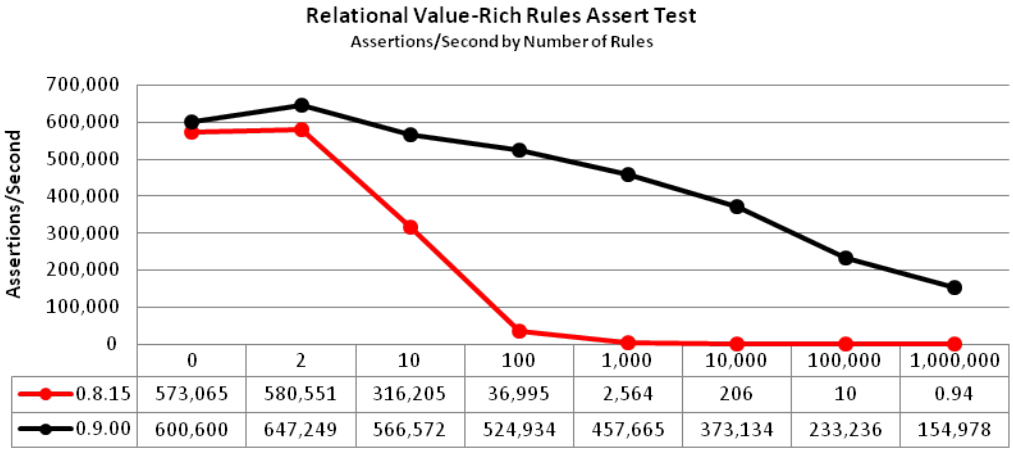
Earlier versions of NodeBrain handled this type of rule set well at 10,000 rules and below. Performance dropped off above 10,000 rules more quickly. Version 0.9.00 introduced hash tables that grow with the number of objects, enabling it to achieve higher performance when the number of rules becomes outrageous.

7.4 Relational Value-Rich Rules Assert Test

The *relational value-rich rules assert* test uses `assert` commands on a set of `on` rules with far more values than terms in relational conditions.

```
# Rules - where X=0,1,2,...,N-1
define rX on(a=X and b<>"X");
# Assertions 4 million times, where X=0,1,2,...,N-1 repeating
assert a=X,b="X+1";
```

The number of rules, `N`, is shown on the horizontal axis in the chart below.



	0	2	10	100	1,000	10,000	100,000	1,000,000
0.8.15	573,065	580,551	316,205	36,995	2,564	206	10	0.94
0.9.00	600,600	647,249	566,572	524,934	457,665	373,134	233,236	154,978

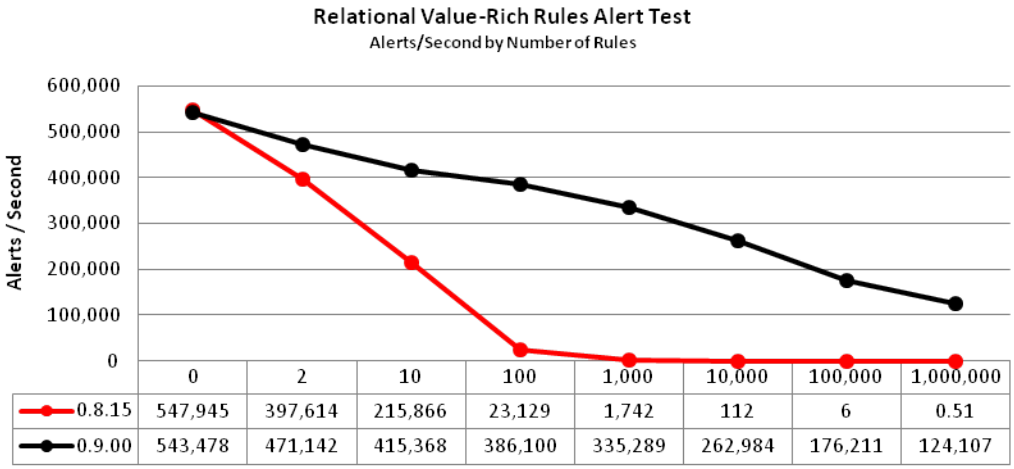
Relational value-rich rule sets were not anticipated until version 0.9.00. The performance of prior versions on this test was quite poor. This is because the rule engine operated like described in the *Spreadsheet Analogy* section of the *Rule Engine* chapter. Version 0.9.00 introduced a shortcut to bypass many evaluations of cells with relational operators that reference both a variable cell and a constant cell. Every cell has an axon tree that identifies all referencing cells. When a large number of the referencing cells have a relational operator and the other operand is a constant, an axon accelerator cell is injected to minimize evaluation on change. Since nodebrain ensures there is only one cell with a given formula, only one referencing cell with the `=` operator and a constant as the other operand can be true at any given time. A change turns it to false, and a binary search can quickly find the one false referencing cell that turns to true, if any. With 1,000,000 rules, this enhancement improved the performance from 0.94 to 154,978 assertions per second.

7.5 Relational Value-Rich Rules Alert Test

The *relational value-rich rules alert* test uses `alert` commands on a set of `if` rules with far more values than terms in relational conditions.

```
# Rules - where X=0,1,2,...,N-1
define rX if(a="abcX" and b="defX" and c="xyzX");
# Asserts - 4 million times, where X=0,1,2,...,N-1 repeating
alert a="abcX",b="defX",c="xyzX";
```

The number of rules, `N`, is shown on the horizontal axis in the chart below.



	0	2	10	100	1,000	10,000	100,000	1,000,000
0.8.15	547,945	397,614	215,866	23,129	1,742	112	6	0.51
0.9.00	543,478	471,142	415,368	386,100	335,289	262,984	176,211	124,107

A good portion of the change in performance on this test from 0.8.15 to 0.9.00 is the same as explained for the previous test. However, `if` rules had another problem when a large number were specified for a single node. An `if` rule fires on every `alert` when true, unlike `on` and `when` rules that fire only when they transition to true. This requires scheduling them to fire under a different mechanism. The scheduling mechanism for `if` rules was enhanced in 0.9.00 to achieve better performance.

Index