Message NodeBrain Module

Release 0.9.02

Message NodeBrain Module
August 2014
NodeBrain Open Source Project

Release 0.9.02

Author: Ed Trettevik

MIT License

NodeBrain License

History

2010-09-13 Title: *Message NodeBrain Module*
Author: Ed Trettevik <eat@nodebrain.org>
Publisher: NodeBrain Open Source Project

Version 0.1

- Initial prototype. This document describes the Message module as released with NodeBrain 0.8.5. We anticipate changes to this document as the module evolves to a version 1.0.

Preface

This manual is intended for users of the Message NodeBrain Module, a plug-in for broadcasting messages to a set of NodeBrain agents. The Message module was first introduced in NodeBrain 0.8.0 in early 2010. This module should be treated as a prototype. It has not yet been exercised enough to fully validate the design. We expect it to evolve as we gain experience.

This module was developed in support of a specific application requiring multi-master replication of data objects managed by a proprietary module. However, it provides a generic message server and a NodeBrain command-specific message client, making it useful to various NodeBrain applications. Custom modules can use the NodeBrain Message API to implement application-specific message clients. It is the responsibility of a message client to handle conflict resolution.

In November 2010, this module was first used in a production application supporting a form of user authentication requiring high availability.

Table of Contents

1 Concepts

The Message module is a NodeBrain plug-in for broadcasting messages to multiple peers. It uses the NodeBrain Message API, which is also available to other modules. Here we introduce some basic concepts of the API.

1.1 Message Log

A message log is a directory containing files used to manage the delivery of messages from one node to another within a set of nodes that share messages. This set of nodes is a "cabal." The relative path for a message log directory within a NodeBrain caboodle is `message/`*`cabal`*`/`*`node`*.

`name``.msg` A named message file identifies the last message received from every node within a cabal. This information is the "state" of the node and a named message file is sometimes referred to as a "state file." This file may be a symbolic link to the active numbered message file.

`nnnnnnnn``.msg` A numbered message file contains a header providing the log's state at the time the file was started. This is followed by a set of message records. The number *nnnnnnnn* starts at 00000001 and increments each time a new file is created. A new file is created when the active file reaches a specified size. All message files other than the active one end with a footer record. Every record in a message log file contains a header portion and a data portion. The header portion of each record and the header and footer records are used to manage message delivery. The data portion of a message record contains the application message to be delivered. An application message may be text or binary data.

`name``.cursor` A cursor file points to an offset within a numbered message file. Cursor files may be used by processes that read a message log and need to keep track of their position within the log. A cursor file is not required by processes that maintain state in some other way.

`~.socket` Each message log has a single "producer"—a process that writes to the log. A producer file is a local domain socket that can be used by "consumers" (processes that read from the log) to request a stream of new messages via UDP packets.

`name``.socket` A consumer file is a local domain socket used for sending messages via UDP packets from the producer to a consumer.

1.2 Producer and Consumer

A message producer is a component that writes to a message log. A message consumer is a component that reads from a message log. Consumers keep track of their position within the message log by updating a cursor file. A consumer may receive messages by polling the

message log for new messages or by requesting that the producer provide a stream of new messages via UDP.

A consumer that requests a UDP stream of messages doesn't have to read the message log, unless a missed packet is detected. The UDP packets are numbered to enable the detection of a missed packet and include the offset within the message log where the next message will be stored. This enables the message server to start reading directly at the message contained in a missed packet.

The goal of this design is to ensure rapid delivery when both the producer and server are running efficiently, but also to decouple the producer and server so either one can operate without the other, or at different speeds, at any given time.

1.3 Server and Client

A message server delivers messages from a message log to message clients. A message client is a component that connects to a message server and requests messages starting at a given state. While a client maintains a connection to a message server, messages will continue to stream in near real-time as generated by the message producer. The connection between a client and server can actually be requested by either the client or the server, meaning the client and server are peers.

Often a component is both a message client and a message producer, and the component logs messages it receives from servers as well as messages it originates. The corresponding message server is then able to serve messages originating at multiple producers. Because a server is a consumer, the communication between a producer and a server is described in the previous section. The current design allows for only one server per message log. This is because a server always uses the node name as the consumer name.

1.4 Source and Sink

A message node is a "source" when the producer is not also a client. The message log produced by such a node contains only messages it generates. A message node is a "sink" when it has a client but no server. A client of a "sink" node may replicate received messages, in which case it is a producer and may have consumers. The use of source and sink nodes is appropriate in cases where you simply want to collect messages from one or more nodes and process the messages at one or more other nodes.

1.5 Cabals, Node Numbers, and State

A message cabal is a set of nodes sharing messages for some purpose. The message log of every node with a server is assigned a number from 1 to 255, which means there can be at most 255 nodes with servers in a cabal. All nodes without servers may share a common node number of 0. Each client is required to maintain a state, which is defined as a set of the last message identifier received from each server. A client's state is passed to a message server when a connection is established, enabling the server to only deliver messages the client is not known to have already received.

1.6 Adaptive Ring, Hub, and Spoke Topology

Each node within a cabal has a strategy for maintaining connections with other nodes. This strategy depends on the type of node.

Type	Strategy
source server	accept connections from sink clients
sink client	request connections to all source servers
hub server	accept connections from hub and spoke clients, request connection to preferred hub client
hub client	accept connections from hub servers, request connection to preferred hub server
spoke client	request connection to preferred hub server

Sink and source nodes use a one-directional, many-to-many topology. Hub nodes form a ring topology, enabling efficient and scalable message broadcasting to all nodes. Spoke nodes are able to tap into any hub node, enabling access to all broadcast messages without having to be a conduit for the ring.

For hub and spoke nodes, an ordered list of all cabal nodes specifies the preferred connections: *node1*,*node2*,...,*nodeN*. This list is circular, so *node1* follows *nodeN*. A hub server prefers connecting to the first hub client following it in the circular list. If the connection fails, it continues through the list until a connection to a hub client is made, or there are no more hub clients (reached self in list). A hub client prefers connecting to the first hub server prior to it in the circular list. Like a server, it continues through the list (only backwards) until a connection succeeds or there are no more hub servers (wrapped around to self). Spoke clients use the same strategy as hub clients to search the list backwards for the first hub for which a connection succeeds. Sink clients search the list and attempt to connect to every source server.

The most interesting connections are those between hub clients and hub servers. These can be requested and accepted by both clients and servers. So using the terms "client" and "server" is a bit awkward. If you think of hub nodes as peers, they each want a connection to the first hub node ahead and behind themselves in the circular list, forming a ring. If a given node is down or unreachable it is bypassed and a smaller ring is formed.

When a node does not have a preferred set of connections, it periodically attempts to establish a preferred set of connections. Once a preferred connection is established, an alternate connection requested by the node is dropped. Extra connections requested by peers are maintained until the peers decide to drop them. In other words, the node that requests a connection is responsible for deciding when it is no longer needed.

This strategy enables replication on broadcast rings to adapt to network or system problems. For example, if a set of five nodes in a ring of ten nodes becomes isolated from the other five because of a network problem, both sets of five will form into separate rings. When the network problem is resolved, the larger ten-node ring will form again.

1.7 Delivery Sequence

Messages originating at any given node are delivered in sequence, once and only once, to every subscribing node. By "delivery," we mean complete delivery. In a scenario where

a ring adapts to a problem and is in the process of repairing itself, it is possible to have redundant transport, or partial delivery, of a message within the messaging layer. However, only messages not yet received will make it through final delivery to the application layer.

Messages originating on two different nodes can arrive at a third node in any order. Let's say node A and node B are originating messages and node C subscribes to both. Messages from A will arrive in order at C, once and only once. Messages from B will arrive in order at C, once and only once. However, a message A1 originating at node A before a message B2 originates at B, may arrive at C before or after B2. It is like shuffling a deck of cards with a stack from A in the left hand and a stack from B in the right hand. The cards from a given hand stay in order, but there are many ways the two stacks might merge.

1.8 Conflict Resolution

When using the NodeBrain Message API to implement multi-master object replication, it is the application's responsibility to perform conflict resolution. There is no one approach that will work for all applications. However, there is a general approach that will work for many applications.

- Assign a unique identifier to each object.
- Assign a unique instance number to each object store where updates originate.
- Include an instance attribute for every object. This identifies the instance that originated the last update.
- Include a version number attribute for every object.
- Include an update time stamp attribute for every object. This not required, but nice to have.
- When an object is added or updated, assign the local instance number, increment the version number, stamp the current time, and issue an `"assert"` transaction message including a complete copy (specification) of the updated object and both the instance number and version number prior to update.
- When an object is deleted, issue a `"delete"` transaction message including object identifier, instance, and version.
- When processing an `"assert"` transaction message for an object that doesn't already exist, add the object.
- When processing an `"assert"` for an existing object and the instance and version are the ones the transaction claims were updated on the original node, then apply the update. Otherwise, apply the update only if the new instance number is lower than the existing instance number.
- When processing a `"delete"` transaction message for an existing object, only delete it when the instance and version in the message match the local copy.

This approach ensures that all instances of the object store remain synchronized over time, although they can be a bit off at any given moment. This is not the same as saying all object stores will be `"correct."` We are only claiming they will be the same. A simultaneous delete and update on two different object stores will preserve the update and drop the delete at every store. This seems like a best result for many applications, but for others it might be the wrong result. Simultaneous updates at two different stores will preserve

the update from the lower numbered instance. This ensures all object stores end up the same, but which update should actually win is not clear. For a given application, there may be additional factors for making this determination, perhaps a ranking of users initiating the update, or perhaps the updates can be merged to accept both updates. So the conflict resolution logic must be designed for the application. The approach outlined above can be used for many applications and can be used as a starting point for designing others.

2 Tutorial

This section illustrates the use of the Message module with simple examples.
bingo. define alert node message.producer("alert","E1",1);

3 Commands

This section describes the syntax and semantics of commands used with the Message module.

3.1 Define

This section describes the syntax for defining Message nodes.

3.1.1 Define Producer

The `message.producer` node is used to write NodeBrain commands to a message log for processing by other members of a message cabal.

```
define node node message.producer("cabal","nodename",nodenumber)[:option[,...]];
```

Parameter	Description
cabal	Identifies a cabal subdirectory in the message subdirectory of a caboodle (root directory of NodeBrain application). This is a directory containing message logs for every node in the cabal that run within the application on the local host.
nodename	Identifies a node within the cabal and a message log (directory) within the cabal subdirectory.
nodenumber	The unique node number from 1 to 255 assigned to this message log within a cabal, or 0 for nodes with no server.

Option	Description
trace	The `trace` option is used to generate log messages for troubleshooting.

3.1.2 Define Consumer

A `message.consumer` node is used to read NodeBrain commands from a message log and pass them to the interpreter.

```
define node node message.consumer("cabal","nodename",nodenumber,"name")[:option[,...]];
```

Parameter	Description
cabal	Identifies a cabal subdirectory in the message subdirectory of a caboodle (root directory of NodeBrain application). This is a directory containing message logs for every node in the cabal that run within the application on the local host.

nodename	Identifies a node within the cabal, and a message log (directory) within the cabal subdirectory.
nodenumber	A unique node number from 1 to 255 assigned to this message log within a cabal, or 0 for nodes with no server.
name	This parameter uniquely identifies a consumer.

Option	**Description**
trace	The `trace` option is used to generate log messages for troubleshooting.

3.1.3 Define Server

A `message.server` node accepts connections from clients and delivers messages that are new relative to a client's state. A state is defined by the last message identifier received by the client from each non-zero node number within the cabal.

```
define node node message.server("cabal","nodename")[:option[,...]];
```

Parameter	**Description**
cabal	Identifies a cabal subdirectory in the message subdirectory of a caboodle (root directory of NodeBrain application). This is a directory containing message logs for every node in the cabal that run within the application on the local host. The parameter also identifies a subordinate rule node for cabal configuration. See "Cabal Configuration" below.
nodename	Identifies a node within the cabal, a message log (directory) within the cabal subdirectory, and subordinate cabal node configuration rule node. See "Cabal Configuration."

Option	**Description**
trace	The `trace` option is used to generate log messages for troubleshooting.

3.1.4 Define Client

A `message.client` node connects to one or more Message servers and requests a feed of all messages that are new relative to the client's state. A state is defined by the last message identifier received for each producer within the cabal.

```
define node node message.client("cabal","nodename")[:option[,...]];
```

Parameter	Description
cabal	Identifies a cabal subdirectory in the message subdirectory of a caboodle (root directory of NodeBrain application). This is a directory containing message logs for every node in the cabal that run within the application on the local host. The parameter also identifies a subordinate rule node for cabal configuration. See "Define Server" for a description of cabal configuration.
nodename	Identifies a node within the cabal, a message log (directory) within the cabal subdirectory, and a subordinate rule node configuring the node within the cabal. See "Cabal Configuration."

Option	Description
trace	The `trace` option is used to generate log messages for troubleshooting.

3.1.5 Cabal Configuration

Server and client nodes reference a cabal configuration rule set. These rules identify all nodes in the cabal and information needed to communicate.

A cabal configuration node identified by the cabal parameter is defined as follows.

```
define node node message.{server|client}("cabal","nodeI")[:option[,...]];
node. define cabal node;
node.cabal. source agent/cabal.nb;
```

The `agent/cabal.nb` file looks like this.

```
define ring cell "node1,...,nodeI,...,nodeN";
define node1 node;
...
#
define nodeI node;
nodeI. define id     cell I;       # instance number (e.g., 3)
nodeI. define type   cell "hub";   # hub | spoke | source | sink
nodeI. define client cell "tls://ipaddress:port"; # all nodes
nodeI. define server cell "tls://ipaddress:port"; # hub or source only
#
define nodeN node;
...
```

When all nodes are running on the same host, you may specify clients and servers using local domain sockets as "`unix://path`". If security is not an issue, you may also replace "tls" with "tcp" for remote clients and servers. In all cases, it is important to use identical cabal configuration files for every node in the cabal. The client and server entries for a given node tell it how to listen for connections as well as telling every other node how to connect.

3.2 Assertions

The Message module does not implement the `assert` and `alert` commands.

3.3 Cell Expressions

The Message module does not implement cell evaluation.

3.4 Node Commands

This section describes commands implemented by the Message module for use with defined nodes.

3.4.1 Trace

The `trace` command is used to toggle the `trace` option for troubleshooting.

```
node:trace [on|off]
```

3.5 Module Commands

The Message module provides commands for managing message logs. These commands are registered with the command interpreter so they may be invoked without first defining a node. These commands start with `message` followed by a period and then a verb.

```
message.verb body
```

3.5.1 Create

The `message.create` command is used to initialize a message log before using for the first time with a message node.

```
message.create [path] cabal nodename nodenumber [type]
```

Element	Description
path	Specifies a path to a directory containing message logs. When specified, a path must start with "/" or "./". A path starting with "./" is relative to the working directory. The default value is "./message", which assumes the command is executed with a NodeBrain caboodle as the working directory and message logs are stored in this directory.
cabal	The name of the cabal in which the new message log will participate.
nodename	The name of the node for which a new message log is requested.

nodenumber Number from 0 to 255 assigned to the specified node within the specified cabal. This number may be zero for a spoke or sink node, but must otherwise be a unique number within the cabal greater than zero. The `create` command is not aware of the numbers that may have been assigned to other nodes, perhaps on other machines, so you must manually reference a common cabal configuration file to make this assignment properly.

type When the *nodenumber* is zero (a message log with no server), specify `content` if you want messages logged, or `state` to simply maintain a node's message state. When `state` is specified, the state is stored as a message log file header record and updated as each message arrives. This option may be used by "sink" and "spoke" client nodes to keep track of their state without using up file space to store messages. There are times where it is desirable to store message content at "sink" and "spoke" client nodes for visibility or for sharing with local consumer nodes. When the *nodenumber* is non-zero, the *type* may be omitted, or a value of `content` may be specified. A value of `state` is not allowed with a non-zero *nodenumber*.

- `content` - Write messages to the log.
- `state` - Only maintain a state (header) record.

When the `content` option is specified, or used by default, a numbered message file is created with a named symbolic link. The numbered message file is initialized with a header record.

```
path/cabal/nodename/nodename.msg -> 00000001.msg
path/cabal/nodename/00000001.msg
```

When the `state` option is specified, the named message file is created as a regular file with a header record. The content of the named file is the same as the numbered file created under the `content` option.

```
path/cabal/nodename/nodename.msg
```

In either case, the named message file can be read to determine the state of the message log.

3.5.2 Export

The `export` command is used to convert a message log file into text. This may be useful for troubleshooting. In the future, we anticipate providing a corresponding `import` command, which would enable repairs to corrupted log files using a text editor.

```
message.export [path] cabal nodename nodenumber filename
```

Element	Description
path	Specifies a path to a directory containing message logs. When specified, a path must start with "/" or "./". The default path is "./message".
cabal	Specify the name of the cabal in which the message log participates.
nodename	Specify the name of the cabal node.
nodenumber	Number from 0 to 255 assigned to the specified node within the specified cabal. The *nodenumber* parameters is currently required by the NodeBrain Message API for validation. If the specified *filename* is not for the specified *nodenumber*, an error is displayed and the file is not exported. While this validation is appropriate for other uses of the Message API, a future enhancement to the API may enable the `export` command to work without requiring the *nodenumber*.
filename	Specify the name of the message file to export. The file name is of the form *`nodename`*`.msg` or *`nnnnnnnn`*`.msg`.

The full path to the message log exported is shown below.

```
path/cabal/nodename/filename
```

3.5.3 Prune

The `prune` command is used to retire message files older than specified time period.

```
message.prune [path] cabal nodename nodenumber retention
```

Element	Description
path	Specifies a path to a directory containing message logs. When specified, a path must start with "/" or "./". The default path is "./message".
cabal	Specify the name of the cabal in which the message log participates.
nodename	Specify the name of the cabal node.

nodenumber Number from 0 to 255 assigned to the specified node within the specified cabal. The *nodenumber* parameters is currently required by the NodeBrain Message API for validation. If the specified *filename* is not for the specified *nodenumber*, an error is displayed and the log is not pruned. While this validation is appropriate for other uses of the Message API, a future enhancement to the API may enable the `prune` command to work without requiring the *nodenumber*.

retension Specify a number *n* followed by a letter identifying the time unit: (d)ay, (h)our, (m)inute, (s) second. For example, `20d` specifies a retention of 20 days.

4 Triggers

A `message.client` node passes all commands received from a `message.server` to the interpreter. There are no other trigger conditions implemented by the Message module. A future enhancement may provide an option for `message.server` nodes to generate connection status change alerts, enabling rules to respond.

Index