# cenode.js

**Revision 1.1**

W.M. Webberley & A.D. Preece*
*Web:* cenode.io - *Mail:* info@cenode.io

## Preamble

This document descibes the concept of 'CENode' - a pure JavaScript implementation of the CEStore[1] that is under development by IBM as part of the ITA project. As with the existing CEStore, ITA CE (Controlled English) is used 'all the way down' for constructing and modifying a conceptual model, and populating it with instances.

CENode (and surrounding functionalities) is distributed as a single JavaScript file, known in this document as `cenode.js`, that is designed to work in a wide variety of settings, such as within a web app, within a JavaScript application (such as Node.js), and also as a RESTful web service. Individual devices running any type of instance of CENode are provided with equal functionality that enables users to interact with a CE-centred knowledge base at the edge of the network. The library also comes equipped with a wide range of networking capabilities that enables it to interact with known peers, subject to customisable policies, over a network connection.

Providing CEStore-style functionality at the network edge gives a number of key benefits;

- Users have access to and can interact with a CENode agent directly on their device. Any CE provided to the agent can be parsed locally and any local knowledge stored can later be 'told' to other agents once a network connection is (re-)established.
- Features such as 'autocorrect' and CE 'spellchecking' can be provided at no bandwidth cost. The local agent can quickly check validity of any CE as it is being written in order to guide the user towards inputting correct CE and also giving insight into known concepts and instances.
- Instead of relying on a single CEStore server with a centralised knowledge base, CENode supports a network of peers with different "local" knowledge base variants.

*CENode is a joint project between School of Computer Science & Informatics, Cardiff University and IBM UK as part of the ITA Project (www.usukitacs.com/about_ita).

[1] https://www.ibm.com/developerworks/community/groups/service/html/communityview?communityUuid=558d55b6-78b6-43e6-9c14-0792481e4532

All communication with instances of CENode should be made through CE. Indeed, modifying the knowledge base requires users (or other agents) to submit information in CE, whilst extracted information can be returned programmatically in JSON format.

It should be noted that CENode does not aim to be a fully-fledged CE engine, in that it lacks certain capabilities that more complex systems support, such as rules. CENode is instead designed to be lightweight and easily-deployable and focuses on core CE functionality - supporting a conceptual model and instance management - and the blackboard architecture (see Section 3.2) through the CECard conversational protocol.

For further information on CE and the CECard protocol, please see:

- [2]: Conversational Sensing (*Preece, A. et al.*), in Next Generation Analyst II, SPIE DSS, 2014.
- [1]: Human-machine Conversations to Support Multi-agency Missions (*Preece, A. et al.*), in SIGMOBILE - Mobile Computing and Communications Review, 2014.
- [3]: Information Extraction Using Controlled English to Support Knowledge-Sharing and Decision-Making (*Ping, X. et al.*).

The rest of this document highlights the key features of CENode and describes ways in which it can be deployed and used for a wide range of applications.

# 1 ITA Controlled English & Dialects

The CE language is thoroughly documented[2] and CENode aims to be fully compatible with the subset of this specification defining model and instance creation, enabling multi-agent systems to include both CENode instances and the CEStore. This section describes the key types of sentences understandable by CENode and some additional supported dialects.

Please note that CENode does not (currently) support CE rules or the 'expressed by' clause used to declare synonyms.

## 1.1 Key Sentence Types

All modifications to the underlying CENode conceptual model are made through `conceptualise` statements.

For example, the sentence below creates a new concept, called 'teacher' as a subclass of the concept 'person' (assuming that 'person' has already been conceptualised):
`conceptualise a ∼ teacher ∼ T that is a person.`

Since, in this example, we have declared that a teacher is a type of person, then the CENode will allow instances of teacher to be made with the properties associated with

[2]Collaborative human-machine analysis using a Controlled Natural Language (*Mott, D.H. et al*) - https://www.usukita.org/sites/default/files/SPIE_AnalysisWithCNL_A4_7A_0.pdf

the person concept.

The following sentence modifies the 'teacher' concept to add some further properties:
`conceptualise the teacher T ∼ teaches ∼ the class C and has the subject S as ∼ subject ∼ and has the value A as ∼ age ∼.`

Submitting the above sentences to CENode will create and modify the 'teacher' concept. If 'class' and 'subject' are not already concepts in the model then the second sentence will fail to execute, since the node will be unable to correctly infer the types related to this concept. If the sentence is executed correctly, then the node will allow new instances of 'teacher' to have a `teaches` relationship and have `subject` and `age` values.

New instances of an existing concept can be declared with normal CE:
`there is a teacher named 'Mrs Smith'.`

As long as 'teacher' has been declared as a concept, then our 'Mrs Smith' instance will be created. We can then modify this instance:
`the teacher 'Mrs Smith' teaches the class 'B2' and has the subject 'Computing' as subject and has '45' as age.`

In this example, CENode will attempt to do some more work on behalf of the user or agent providing this information. If, for example, the subject 'Computing' had not yet been declared as an instance of `subject`, then a new instance of type `subject` named 'Computing' will be created. The same applies for the `class` 'B2'. Since `age` is simply a value of no particular type, there is no new instance to be created here, but the value will be embedded inside the 'Mrs Smith' object type. Supporting implicit instance creation is not as dangerous as the conceptualising equivalent, since it only involves creating an empty instance of a concept that already exists.

Note that if a property is encountered in the input CE that is not declared in the 'teacher' conceptual model (or in any of its ancestors), then this property will be ignored. The remainder of the sentence will still be executed. As with the CEStore, instance and concept deletion is not supported.

## 1.2 Additional Sentence Types

CENode is also able to understand some additional sentence structures to make interaction a little easier and to support information extraction. These sentences are not CE, and are instead a form of *gist* [1]. However, they can be safely sent to a CENode, which automatically processes them as if they are valid CE. In addition, whilst the CE specification defines valid sentences to be those ended in a full-stop (period), CENode will also accept sentences that do not.

### 1.2.1 Shorthand instance modification

One key addition to the grammar is a shorthand for modifying instances. The above 'teacher' example can be re-written as:

```
Mrs Smith teaches the class 'B2' and has the subject 'Computing' as subject
and has '45' as age.
```

CENode will attempt to resolve instance names to form a valid CE sentence, which is then parsed. Note that instance names in this type of sentence do not need to be case-sensitive. Replacing 'Mrs Smith' in the above example with 'mrs smith' will still work.

In addition, CENode will process one-word instance names that are not quoted, but multi-word names and all values still need to be quoted:

```
Mrs Smith teaches the class B2 and has the subject computing as subject and
has '45' as age.
```

### 1.2.2 Question-asking

Another addition to CENode is the ability to answer questions. This addresses the 'who/what/where' information useful to researchers and also allows information to easily be extracted from the node in an easy-to-understand gist format. Note that although, technically speaking, questions in this format are themselves gist, CENode treats them as if they are valid CE, so that they can safely be embedded in `ask card`s.

To this end, 'who' and 'what' questions are understood in the same way by the node. This means that the questions below are equal in meaning:

```
what is mrs smith?
who is mrs smith?
```

Both of these questions would result in a gist output looking something like the following:

```
Mrs Smith is a teacher.  Mrs Smith teaches the class 'B2' and has the subject
'Computing' as subject and has '45' as age.
```

'Who' and 'what' questions can also be used to find out about concepts and properties. For example, the sentence

```
what is a teacher?
```

would result output similar to:

```
A teacher is a type of person.  An instance of teacher teaches a type of class
and has a type of subject called subject and has a value called age.
```

Similarly, asking
`what is teaches?`
would give:
`'teaches' describes the relationship between a teacher and a subject (e.g. "the teacher 'TEACHER NAME' teaches the subject 'SUBJECT NAME'").`

'Where' questions work slightly differently and requires the CORE node model (see later) to be loaded to the store. 'Where' questions are only valid for instances, and will only provide a response if the instance in question has a property associated with some kind of location.

The CORE model includes a concept called `location` which can be used as a parent of other types of location (e.g. a building, a room, a road, etc.). As long as the instance in question has a property relating to any concept that has `location` as an ancestor, then a meaningful response can be obtained.

For example, let's assume that the `person` concept (that `teacher` inherits from) supports a relationship called 'lives in' that targets an instance of type `house`, which is a child of `location`:
`the teacher Mrs Smith lives in the house 'Number 23'.`

We can now ask a 'where' question:
`Where is Mrs Smith?`
and receive a response:
`Mrs Smith lives in the house 'Number 23'`

In general, CENode ignores stop words and punctuation, so the following are all valid questions:
`what is an apple?`
`where is the banana?`
`Who is Mrs Smith`

## 2 Node Models

A node's knowledge base (KB) represents the concepts and instances it knows about. Providing CE to the node updates the KB and asking questions allows the KB to be queried. Models allow a skeleton KB to be produced from which the knowledge can grow as new CE is added to the node.

A CE model is essentially a collection of CE sentences that can be delivered to a node in order to develop its conceptual model and populate its KB with initial instances. Since the CENode library is written in JavaScript, then a model is simply an array of CE sentences. For example, consider a simple model:

```
var my_model = [
    "conceptualise a ~ teacher ~ T.",
    "conceptualise a ~ class ~ C.",
    "conceptualise the teacher T ~ teaches ~ the class C.",
    "there is a teacher named 'Mrs Smith' that teaches the class 'B2'."
];
```

This model can then be loaded into an instance of CENode when the node is instantiated. See Section 6 for more details on this.

`cenode.js` comes bundled with models that can be used to initialise a CENode instance with some basic knowledge. As mentioned previously, loading such a model is sometimes mandatory (for example, when querying for an instance's location), since the models may include concepts and instances necessary for interacting with such information. As we progress through this document, the purpose of core models will become more clear.

It is usually recommended that *any* instance of CENode is at least supplied the CORE model, as this includes the `location` concept, as well as other concepts that are useful to subclass when further populating the model. These models are included in `cenode.js`'s `MODELS` object, so that the CORE model can be accessed by `MODELS.CORE`.

Currently, only the CORE model is recommended for general use. Instantiating a CENode with a particular model in different types of applications is described later on.

# 3 CENode Agents

Each CENode instance is accompanied by its own agent. A node's agent is spawned upon the node's instantiation and represents the recommended interface between the node's KB and its user. In a multi-node system, agents also handle any node-node interaction through the respect of 'policies' (see Section 5.2).

A CENode agent, although bundled with `cenode.js`, is actually entirely separate from the node's KB, and in fact has no more access to the conceptual model than another user programmatically using the library. Agents only work properly when the CORE model has been loaded, and each agent in a given CENode system should have a unique name, which by default is 'Moira' in the code. Information about an agent can be added to a node's KB (whether this refers to the local or another agent) using CE as follows (assuming that the CORE model has been loaded):

```
there is an agent named 'agent1'..
```

## 3.1 Cards

Agents are only useful when 'cards' are used as a delivery mechanism for CE, which forms the basis of the blackboard architecture implemented by the CEStore, and which

is also used as the recommended primary means for human-node and node-node communication in CENode. Different types of card extend from the `card` concept, and they are all included in the CORE model. Cards wrap CE in a value property and enable the information within to be shipped to different agents as required, and a particular agent will only 'open' a card to reveal the contents if the agent is an intended recipient.

It is rare that the `card` concept is used directly. Instead, one of its subclasses should be used, since the type of card determines what the information contained represents and what the response (if any) should be. Here is an example of a `tell card`:

```
there is a tell card named 'msg1' that is to the agent 'agent1' and is from
the agent 'agent2' and has the timestamp '123456' as timestamp and has 'there
is a teacher named \'Mrs Smith\'' as content
```

A `tell card` should be used to tell a particular agent some information, and an `ask card` should be used to query for some information. Using what we've covered so far, all of the `conceptualise` and instance-manipulation sentences would go into a `tell card` and the questions discussed in Section 1.2.2 would be wrapped in an `ask card`. Using the correct kind of card. The 'from' field of a card can be used by an agent to send back a response, if needed, and some agents may decide to ignore cards that have an old timestamp.

## 3.2 Blackboard Architecture

As mentioned, agents begin their life when the CENode they are associated with is instantiated. Agents continuously check their node's KB for any cards that are addressed to themselves. If a card is found that is addressed to and hasn't yet been seen by the agent, then the agent will act upon it.

If the card is a tell card, then the agent will open up the CE content contained within and feed it into its node with the aim of modifying its KB. If the card is an ask card, then the agent will attempt to answer the question and send a response back to the entity that initially sent the card.

If a card instance exists in a node's KB and the node's local agent is *not* a recipient, then no further action will occur for this card on this node. Of course, any programs using the `cestore.js` library may decide to do something with it, but generally it will be ignored by the local agent (unless its name is changed to that of the intended recipient).

Although this may seem useless, it actually forms the basis for the blackboard architecture, in which agents and users can read and write cards from and to a node. Later on in this document we'll cover *policies*, which allow agents to communicate automatically with each other in different ways. Submitting CE to agents wrapped in cards allows only the information that is actually needed by each node to be read by the agent of that node.

In general, any valid CE submitted to a node will be parsed immediately and the node's conceptual model appropriately updated. Sometimes, the node will return a response immediately (either programmatically or in a response to a HTTP request)

containing some relevant information. This usually only occurs when the CE represents a who/what/where question. However, when submitting CE within a card envelope, no response will be returned. This is because creating instances does not invoke a response fom the node and agents work separately and asynchronously from the rest of the CENode process. Agents will read cards from their node in their own time and will write responses back to it when necessary (e.g. in the case of an 'ask card' being submitted). When submitting cards, the contained CE is, essentially, parsed twice. Once when the card is initially submitted to the node (a process which involves adding an instance of 'card' along with its associated information). The second time is when the agent comes round to picking cards from the node and re-submitting the contained CE directly.

# 4 Using CENode

Generally, the installation and inclusion of CENode into your project is very simple, as all that is required is an import of the `cenode.js` library. This section describes how this can be done more clearly.

## 4.1 In a Web Application or Webpage

In a web application or webpage, the `cenode.js` library can be easily imported:

```
<script src="cenode.js"></script>
```

Once imported, a new CENode instance can be instantiated in a later `<script>` block and any required models can be passed as arguments. After instantiation, sentences can be added as direct CE (or embedded within cards):

```
<script>
var node = new CENode(MODELS.CORE, MY_CUSTOM_MODEL);
node.set_agent_name("agent1");

node.add_sentence("there is a teacher named 'Mrs Smith'");
node.add_sentence("there is a tell card named '{uid}' that is to the agent
        'agent1' and is from the individual 'user1' and has the timestamp
        '{now}' as timestamp and has 'there is a teacher named
        \'Mrs Smith\'' as content");
</script>
```

Since we have set the node's agent's name to 'agent1', both of the `add_sentence` lines would have equal functionality (although the node will prevent multiple instances being created with the same name and same type). In the former case, the CE will be parsed directly and the teacher will be added to the node's KB. In the latter, the card will be added to the KB, and the local agent will eventually find the card and update the KB further with the relevant information contained in the card.

Both {uid} and {now} are special character sequences that will be modified by the node once received. Please see Section 6 for more information on these and for other features available to applications using the library in such a way.

## 4.2 In a JavaScript Application

The library is also usable as part of a Node.js program. To get started with this, you will need to first install the Node.js environment. This can be done by visiting their website to download the necessary files (https://nodejs.org) or by using an existing package manager on your system.
For example, with Arch Linux:

```
# pacman -S nodejs
```

with Ubuntu:

```
# apt-get install nodejs
```

and with OS X (with Homebrew installed):

```
$ brew install node
```

Please note that the library is also mostly compatible with other JavaScript runtimes, such as `io.js`.

Once Node.js has been installed, you can create a simple Node.js app in a similar way to using the library in a web app:

```
var cenode = require("./cenode.js");

var node = new cenode.CENode(cenode.MODELS.CORE);
node.add_sentence(...)

... etc.
```

Beyond this point, functionality is precisely the same as that when the library is used in a web application. For more information on the programmatic API, please see Section 6.

## 4.3 As a RESTful Service

`cenode.js` also supports being run directly as a service using Node.js. To accomplish this, then Node.js needs to first be installed as described in the previous section. After installation, then the service can be started by running:

```
$ node cenode.js
Set local agent's name to 'Moira'.
CENode server instance running on port 5555...
```

By default this will start a web server on port 5555 with a local agent named 'Moira'.

The CENode instance run in this way provdes a webpage that you can use to administer the instance. To do so, visit `localhost:5555` in a web browser (or the hostname of the machine running the instance if not local). You will be presented with a display indicating some information about the node instance and will allow some simple controls (such as model-loading and sentence-inputs).

The CENode instance can be launched with different configurations by supplying command-line arguments. For example the below command will start the service on port 5432 and will set the name of the agent to 'agent1' (the output from the server is included below for your information):

```
$ node cenode.js agent1 5432
Set local agent's name to 'agent1'.
CENode server instance running on port 5432...
```

Once running, a RESTful interface is exposed to interact with the Node. For more information on this, please see Section 6.2.

# 5 Multi-Node Systems

As described earlier, CENode instances can either be run independently or as part of a multi-node system. This section outlines methods on how this might be accomplished. In a typical muli-node system, at least one of the nodes will need to be run as a service exposing the required HTTP endpoints.

## 5.1 General

All CENode instances in a multi-node system are, by default, equal in terms of functionality and behaviour. This is the case even if each node is deployed in a different way (e.g. some nodes may be running as a service, some as a web application, and some as a programmatic JavaScript application). Providing information to (and retrieving information from) a local node is simple, as shown briefly earlier and in more detail later on, and supporting inter-node communication is also relatively easy.

The `cenode.js` library comes equipped with the ability to allow agents to communicate over the network with other agents, and will adapt automatically to the environment it exists in. For example, if running in a web page it will use the browser's `XMLHttpRequest` object, and if running as a Node.js app it will use Node.js's `http` module. Either way, there is no intervention required by users when deploying a CENode as part of a multi-node system on a variety of platforms.

## 5.2 Policies

All inter-node communication should be described by *policies*. These are essentially instructions, written in CE, that instruct individual nodes to communicate with each other in different ways. All policy types understood by the agent are included in the CORE model (see Section 2).

Policies written to a particular CENode represent instructions that apply to its local agent. Agents periodically query the policies that are in their node's KB and act upon them accordingly. As such, policies can be created and modified using plain CE once the CENode instance is running with almost immediate effect.

All policies in the CORE model have an 'enabled' field, and any particular policy is active as long as this field is set to 'true'. For example, to disable a particular policy, named 'p1', you could issue the following CE:

```
the policy 'p1 has 'false' as enabled.
```

The local agent will now no longer act on this policy.

The rest of this Section describes the different types of policy in more detail.

### 5.2.1 `tell policy`

A `tell policy` inherits from `policy` and instructs the appropriate agent to tell the policy's target agent everything that the local agent is told.

For example, imagine our loocal agent is called 'agent1' and we tell it about the following agent:

```
there is an agent named 'agent2' that has 'agent2.address.com' as address.
```

We can now create a tell policy targeting this agent:

```
there is a tell policy named 'p1' that has 'true' as enabled and has the agent
'agent2' as target.
```

Once this policy has been created, then our local agent, 'agent1', will tell 'agent2' every piece of information that has been told to 'agent1' in tell cards by wrapping the content in a new tell card and HTTP POSTing this to the appropriate endpoint at 'agent2"s host address. As such, 'agent2' needs to be an agent running as a service instance. Please see Section 4.3 for instructions on setting this up.

Any cards which do not have 'agent1' as a recipient (or any other type of card) will not be included as part of the policy.

### 5.2.2 `ask policy`

An `ask policy` works in almost exactly the same way as a tell policy (with our local agent named 'agent1'):

```
there is an ask policy named 'p1' that has 'true' as enabled and has the agent
```

```
'agent2' as target.
```

In this scenario, every `ask card` sent to 'agent1' will also be sent to 'agent2' using a HTTP POST request. As with targets of a `tell policy`, target agents of an `ask policy` must be instances running as a service instance.

Ask policies are mostly useless unless the agent acting on the policy is able to receive a response from the policy's target. As discussed in Section 3.2, communication between agents and individuals using cards is *asynchronous*, and therefore an answer to a question cannot be included in the response of the POST reequest made as a result of the policy. In reality, when an 'ask card' is POSTed to the target, its agent will get round to reading the card in its own time and will write a card back to its *own* store if the card requires a reply.

Therefore, most multi-node setups using an 'ask policy' will also involve a 'listen policy' targeting the same target as the 'ask policy'. See Section 5.2.3 for more information.

### 5.2.3 `listen policy`

A `listen policy` instructs the local agent, 'agent1', to periodically poll the target agent for instances of 'tell card' sent to 'agent1'. Any cards found are opened and the content is added to the agent's node's KB as normal.

As with the previous two policy types, any target agent must be in a node running a as a service instance.

Listen policies are useful in conjunction with ask policies, since they enable a response to be retrieved from the target of the ask policy. For example, consider the following setup (assuming the local agent is named 'agent1'):

```
- there is an agent named 'agent2' that has 'agent2.com' as address
- there is an ask policy named 'p1' that has 'true' as enabled and
    has the agent 'agent2' as target
- there is a listen policy named 'p2' that has 'true' as enabled and
    has the agent 'agent2' as target
```

This setup will cause 'agent1' to forward all ask cards it receives to 'agent2' and will be able to receive a response from 'agent2', through the listen policy, once 'agent2' has read and replied to the ask card.

### 5.2.4 `forwardall policy`

A `forwardall policy` is slightly more complex because it has more options in its configuration. The general principle is that the agent the policy is active on will forward some tell cards that have been sent to this agent on to a set of other agents as required.

Unlike the other policy types, a `forwardall policy` does not trigger any network requests. Instead, any card-forwardings are made simply by adding targets as *recipients* of the cards. These can then be retrieved by other agents who have a `listen policy` targeting this node.

As with the previous examples, imagine the local agent which is acting on the `forwardall policy` is named 'agent1'.

The construction of a `forwardall policy` might look like this:
`there is a forwardall policy named 'p1' that has 'true' as enabled and has the timestamp '0' as start time and has 'true' as all agents`

In the above example, any tell cards that have previously been sent to 'agent1' and any arriving in future whilest the policy is enabled will have every agent known by agent1's node added as a recipient. Then, if any of these agents make a request to this node (as a result of a `listen policy` or otherwise), they can access these cards.

A node can discover other agents in two primary ways. One is explicit, in that the node has been given CE to describe a new agent:
`there is an agent named 'agent2'.`
The other is implicit, where the node will add to its KB any unknown instances that are mentioned. For example, assume the node does not yet have the agent 'agent2' in its KB and then receives the following card:
`there is a tell card that is to the agent 'agent1' and is from the agent 'agent2' and has 'there is a teacher named Mrs Smith' as content.`
In this case, the node will automatically create an instance of agent named 'agent2', thus discovering its existence.

The 'start time' field specifies that the policy should only affect cards with a timestamp greater than this, and so this can be set to '0' to activate the policy for all tell cards sent to 'agent1' during its lifetime. The 'all agents' field is a boolean which, if 'true', specifies that *all* known agents should be added as a recipient.

If 'all agents' is set to 'false' instead, then a set of agent recipients can be specified. Consider the more complex example below:
`there is a forwardall policy named 'p2' that has 'true' as enabled and has the timestamp '12345' as start time and has the agent 'agent2' as target and has the agent 'agent3' as target`

In the above example, the policy will cause the agent 'agent1' to add both 'agent2' and 'agent3' as recipients to all tell cards sent to 'agent1' with a timestamp greater than '12345' from now until the policy is disabled.

### 5.2.5 `feedback policy`

A `feedback policy` can be applied to an agent in order to make it give some kind of feedback to the agent or individual that has submitted a 'tell card' to it. This behaviour might be useful for providing information on input submitted to the node, and allows the local agent to report any misunderstandings in the input CE.

A `feedback policy` follows a similar setup to the other policy types, in that it can be enabled and can target a particular agent or individual, but, like the `forwardall policy` it will *not* invoke a network request. Instead, any feedback is included in a `tell card` addressed to the target, which is written to the agent's own node. Thus, if responses are required over the network, a `listen policy` must also be used.

Since no network activity is directly involved (unless there is a `listen policy` in place), this type of policy is mostly useful for JavaScript or web applications using the `cenode.js` library directly. Imagine that the local agent is named 'agent1' and there is a user, known as the individual 'individual1', that is submitting information to the node's agent through `tell card`s:

```
there is a feedback policy named 'p1' that has 'true' as enabled and has the
individual 'individual1' as target and has 'full' as acknowledgement
```

With this policy in place, 'agent1' will respond to all `tell card`s sent from 'individual1' with a full description of the action taken by 'agent1' on the node. If this is an error message, then the node will attempt to include information on which parts of the input sentence were not understood. If the message was understood fully, then the full understood CE will be returned in the response.

For security, it may sometimes be necessary for nodes to be restricted on the information returned. For example, in order to keep the inner knowledge of the node obfuscated for whatever reason, the `acknowledgement` property of the policy can be set to 'basic'. In this scenario, only an 'OK' will be sent back to the agent or individual that submitted the original tell card, with no indication of the inner knowledge of the node.

To keep agents from giving any feedback whatsoever, then simply disable the policy or don't set the policy in the first place.

## 5.3 Example Network Topologies Using Policies

Using policies allows for a wide variety of possible network topologies. Combining policies allow for useful configurations of multi-node setups. This section outlines a couple of examples for inspiration.

### 5.3.1 'Point-to-point topology'

In this example, two CENode instances communicate directly to each other by telling each other everything.

To implement this, two instances of CENode (each with a different names) running as services need to be launched. Each instance needs to know the address of the other instance's agent and a tell policy is needed on each node.

For example, consider 'agent1' runs on 'agent1.com' and 'agent2' runs on 'agent2.com'. The configuration CE can be added on each instance's webpage control panel (see Section 4.3 for more information).

On agent1's node, the following sentences are required:

```
- there is an agent named 'agent2' that has 'agent2.com' as address
- there is a tell policy named 'p1' that has 'true' as enabled and
    has the agent 'agent2' as target
```

Agent2's setup is symmetrical:

```
- there is an agent named 'agent1' that has 'agent1.com' as address
- there is a tell policy named 'p1' that has 'true' as enabled and
    has the agent 'agent1' as target
```

### 5.3.2 'Star topology'

In this example, one CENode instance, at the centre of the star, acts as a router of information between any number of 'client' nodes. The router node needs to be run as a service, but the clients can be run in any configuration. In this scenario, each node will tell the router everything it knows, and the router will forward this information on to every other client node.

Firstly, each client node needs to know about the router node and to tell it everything and listen for any cards the router node might have for it:

```
- there is an agent named 'router' that has 'router.com' as address
- there is a tell policy named 'p1' that has 'true' as enabled and
    has the agent 'router' as target
- there is a listen policy named 'p2'  that has 'true' as enabled and
    has the agent 'router' as target
```

Secondly, the router node needs to simply forward every message it receives on to every agent it knows about:

```
- there is a forwardall policy named 'p1' that has 'true' as enabled and
    has the timestamp '0' as start time and
    has 'true' as all agents
```

# 6 CENode API

As discussed throughout this document, `cenode.js` can be used programmatically in a JavaScript or web application as well as a web service for receiving RESTful requests. This section describes the methods available on both types of interface.

## 6.1 Programmatic Interface

When used as a library as part of a JavaScript application or within a webpage (see Sections 4.1 and 4.2), `CENode` instances expose a number of useful public methods. All the methods in this section are callable on instances of `CENode`.

`CENode CENode ([model1[, model2[, model3 ...]]])`

*Instantiates and returns a new* ***`CENode`*** *object with any number of models to initially develop the node's KB. Generally, it is recommended that the default CORE model at least be loaded. Instantiating a CENode starts the lifecycle of an agent within it, whose name is set to 'Moira' by default.*

**Example Usage**

```
var node = new CENode(MODELS.CORE);
```

The Node can instead be instantiated with custom models. These can be created as described in Section 2 and then loaded in the same way:

```
var node = new CENode(custom_model_1, custom_model_2);
```

`String guess_next (String input)`

*Returns a guess of the rest of the CE sentence in* ***`input`*** *for supporting auto-complete. Note that this feature is still under development*

**Example Usage**

```
var guess = node.guess_next("there is a p");
```

In this case, the node will look through its knowledge base to find concepts with name starting with 'p' (for example, 'person'). `guess` would then contain 'there is a person named'.

`Instance[] get_instances ([String concept_name[, Bool recurse]])`

*Return a list of instance objects. If* ***`concept_name`*** *is included, then only instances of this type will be included. If* ***`recurse`*** *is set to* ***`true`*** *then instances of the concept's children, grandchildren, etc., will also be included.*

**Example Usage**

```
var tell_cards = node.get_instances("tell card");
var all_cards = node.get_instances("card", true);
```

`Concept[] get_concepts ()`

*Return a list of concept objects known by the node.*

**Example Usage**

```
var concepts = node.get_concepts();
```

`String get_instance_type (Instance instance)`

*Return a string representing the type (i.e. the name of the concept) of the given instance.*

**Example Usage**

```
var concept_name = node.get_instance_type(instance);
```

`String get_instance_ce (Instance instance)`

*Return a string representing the CE required to construct the instance.*

**Example Usage**

```
var ce = node.get_instance_ce(instance);
```

'ce' will be of the form `"there is a teacher named 'Mrs Smith' that..."`.

`String get_instance_gist (Instance instance)`

*Return a string representing a gist description of the instance.*

**Example Usage**

```
var gist = node.get_instance_gist(instance);
```

'gist' will be of the form `"'Mrs Smith' is a teacher. Mrs Smith teaches the class B2 and..."`.

`String get_concept_ce (Concept concept)`

*Return a string representing the CE required to construct the concept.*

**Example Usage**
`var ce = node.get_concept_ce(concept);`
'ce' will be of the form `"conceptualise a ∼ teacher ∼ T that..."`.

`String get_concept_gist (Concept concept)`

*Return a string representing a gist description of the concept.*

**Example Usage**
`var gist = node.get_concept_gist(concept);`
'gist' will be of the form `"A teacher is a type of person and teaches a type of class and..."`.

`[bool, String] add_ce (String ce[, bool nowrite])`

*Immediately updates the node's KB, as long as 'ce' is valid CE. The returned* `bool` *indicates whether the CE-parsing was successful (*`true` *indicates valid CE) and the returned* `String` *will contain the input CE when it is valid, or an error message if it is not. 'nowrite' is an optional boolean argument, which, if set to* `true`*, will not update the KB during parsing.*
`add_ce` *can parse special character sequences to aid users in creating cards. The sequence* {`now`} *will be replaced by the current timestamp of the node's environment and* {`uid`} *will be replaced by an an appropriate identifier for the card.*

**Example Usage**
`var data = node.add_ce("there is a teacher named 'Mrs Smith'");`

`[bool, String] ask_question (String question)`

*Immediately return a response to a KB query, as long as 'question' is a valid question type. The returned* `bool` *indicates whether the question-parsing was successful (*`true` *indicates valid question) and the returned* `String` *will contain the response gist when it is valid, or an error message if it is not.*

**Example Usage**
`var data = node.ask_question("who is Mrs Smith?");`

`[bool, String] add_nl (String nl)`

*Attempts to build a valid CE sentence based on the NL 'nl' input. The returned* **`bool`** *indicates whether an attempt was made (*`true` *indicates returned CE) and the returned* **`String`** *will contain the attempt if it is valid, or an error message if it is not.*

**Example Usage**

```
var data = node.add_nl("there is a teacher called Mrs Smith");
```

`[bool, String] add_sentence (String sentence[, bool nowrite])`

*Adds a sentence to be processed by the node. In turn, this will first attempt to parse CE, then a question, and finally NL. The returned* **`bool`** *represents whether or not any of the above was successful and the returned* **`String`***'s content varies. For more informtaion, see the documentation for* **`add_ce`**, **`ask_question`**, *and* **`add_nl`** *respectively.*
**`add_sentence`** *can parse special character sequences to aid users in creating cards. The sequence* {**`now`**} *will be replaced by the current timestamp of the node's environment and* {**`uid`**} *will be replaced by an an appropriate identifier for the card.*

**Example Usage**

```
node.add_sentence("there is a teacher named 'Mrs Smith'");
var answer = node.add_sentence("Who is Mrs Smith?");
```

The example below illustrates the use of special character sequences.

```
node.add_sentence("there is a tell card named '{uid}' that has the timestamp
'{now}' as timestamp ...");
```

`String add_sentences (String[] sentences)`

*Adds an array of sentences to the model. Internally, this uses* **`add_sentence`**, *so the above information applies. Responses made by the node are also returned in an array, where the ordered elements in the responses array are associated with the appropriate elements in the sentences array.*

**Example Usage**

```
var sentences = [
"conceptualise a ~ teacher ~ T that is a person",
"there is a teacher named 'Mrs Smith'"
]
var responses = node.add_sentences(sentences);
```

`void set_agent_name (String name)`

> *Sets a new name for the node's local agent. Updating this value will mean the agent will open different cards and will ignore any further cards sent to its previous name. Note that agent names are case-insensitive when qualifying card recipients.*
>
> **Example Usage**
> `node.set_agent_name("Agent 1");`

`String get_agent_name ()`

> *Retrieves the name of the local agent.*
>
> **Example Usage**
> `var agent_name = node.get_agent_name();`

## 6.2 RESTful HTTP Interface

When run as a service (see Section 4.3), which runs on port 555 by default, users can visit `localhost:5555` to administer the node (use a different hostname if not running the node locally). The webpage served at this address allows administers to submit new sentences to the node, inspect instances and concepts, change the name of the local agent, load models, and reset the store.

In addition to these features, a node that is running as a service also exposes a RESTful interface for other nodes (or applications) to make HTTP requests to. There is no authentiation mechanism implemented on `cenode.js` and the service will accept requests from any domain (i.e. CORS is enabled) so that nodes can be accessed from within web applications run within a browser. In fact, these RESTful endpoints are the same ones that are used by the agents themselves when acting on policies applied to them.

`POST /sentences ()`

*Submit a newline-separated set of CE sentences to the node. The sentences should be in the body of the request and no key is necessary. Nodes will return any content produced as a result of parsing the input sentences within the body of the response separted by new lines. If there is no response for an input sentence, then this will be represented by an empty string. This means that, for example, line 2 of the response body corresponds to the sentence on line 2 in the body of the request.*
*Remember that agents are asynchronous, so responses from 'ask cards' will not be included in the response to `/sentences` - instead, use `GET /cards` to check for replies to 'ask cards'.*

**Example Usage**
REQUEST:

```
POST /sentences
Content-Type:  text/ce

what is Mrs Smith?
there is a teacher named 'Mrs Smith'
what is Mrs Smith?
Mrs Smith teaches the class 'B2'
```

RESPONSE:

```
200 OK
Content-Type:  text/ce

I don't know what Mrs Smith is.

Mrs Smith is a teacher.
```

`GET /cards ([?agent=NAME])`

*Retrieve cards from the node. If 'agent' is specified, then only cards addressed to 'NAME' are returned. Cards are returned in pure CE in the body of the response, separated by newlines.*

**Example Usage**
REQUEST:

```
GET /cards?agent=agent1
```

RESPONSE:

```
200 OK
Content-Type:  text/ce

there is a tell card named 'msg41' that is to the agent 'agent1'
there is an ask card named 'msg56' that is to the agent 'agent1'
```

(Note that the cards have been truncated for clarity).

## References


[1] Alun Preece, Dave Braines, Diego Pizzocaro, and Christos Parizas. Human-machine Conversations to Support Multi-agency Missions. *SIGMOBILE Mob. Comput. Commun. Rev.*, 18(1):75–84, February 2014.

[2] Alun Preece, Chris Gwilliams, Christos Parizas, Diego Pizzocaro, Jonathan Z. Bakdash, and Dave Braines. Conversational sensing, 2014.

[3] Ping Xue, Stephen Poteet, Anne Kao, David Mott, Dave Braines, Cheryl Giammanco, and Tien Pham. Information Extraction Using Controlled English to Support Knowledge-Sharing and Decision-Making. *Cognitive Science*, 2012.