The JSON Network Interface Programmers Guide

Ryan M. Hope
Cognitive Science Department
Rensselaer Polytechnic Institute

Draft of July 25, 2013

1 Communication flow

The JNI is based on a client-server model where the task environment acts as the server and the JNI moduleJNI module acts as the client. Since the environment plays the role of server, it should be running (and listening for incoming TCP connections) before the JNI JNI module module attempts to connect to it. A direct benefit of this design is that the environment can be run on a computer different than the one ACT-R runs on. The environment could even be a robot assuming it has a network connection.

Once the JNI moduleJNI module successfully connects to the environment it sends a **reset** command. The JNI moduleJNI module will wait for a **sync** reply from this command which prevents the model from running before the environment is ready. The next command sent by the JNI moduleJNI module to the environment, **model-run**, occurs when the model is started. Again, this command will wait for a **sync** reply from the environment before the model actually starts running. When an ACT-R model requestJNI modules information from the JNI moduleJNI module it returns information stored locally in the device. It is the responsibility of the environment to update the JNI moduleJNI module with new information when things change in the environment. Therefore, it is recommended that the environment send its first **update-display** command to the JNI moduleJNI module before the the **sync** reply for the **model-run** command. This ensures that the JNI modulemodels visual working memory (visicon) is seeded before the first conflict resolution cycle. Once the model is running, any combination of the remaining commands could be sent to or from the JNI moduleJNI module and environment. JNI moduleSee Figure 1 for a minimal example of the communications between the JNI module and environment.

2 JNI API

All communication between the JNI and the environment is encoded in JSON. JSON was picked over other data interchange formats (such as XML) because it has a much smaller grammar and maps more directly onto the data structures used in modern programming languages. JSON is also is also easy for humans to read and write, and easy for machines to parse and generate. JSON is built on two structures, *objects* and *arrays*. An object is an unordered set of name/value pairs. An object begins with { (*left brace*) and ends with } (*right brace*). Each name is followed by : (*colon*) and the name/value pairs are separated by , (*comma*). An *array* is an ordered collection of values. An array begins with [(*left bracket*) and ends with] (*right bracket*). Values are separated by , (*comma*). A *value* can be a *string* in double quotes, or a *number*, or *true* or *false* or *null*, or an *object* or an *array*. These structures can be nested.

The API for commands and responses sent between the JNI module and environment follow a simple pattern. All JSON messages are based on a root JSON object that contains three keys, *model*, *method* and *params*. The value for the *model* key is the name of the current model, the value for the *method* key is the name of the command (*string*) and the value of the *params* field is a JSON object. Complete JSON messages

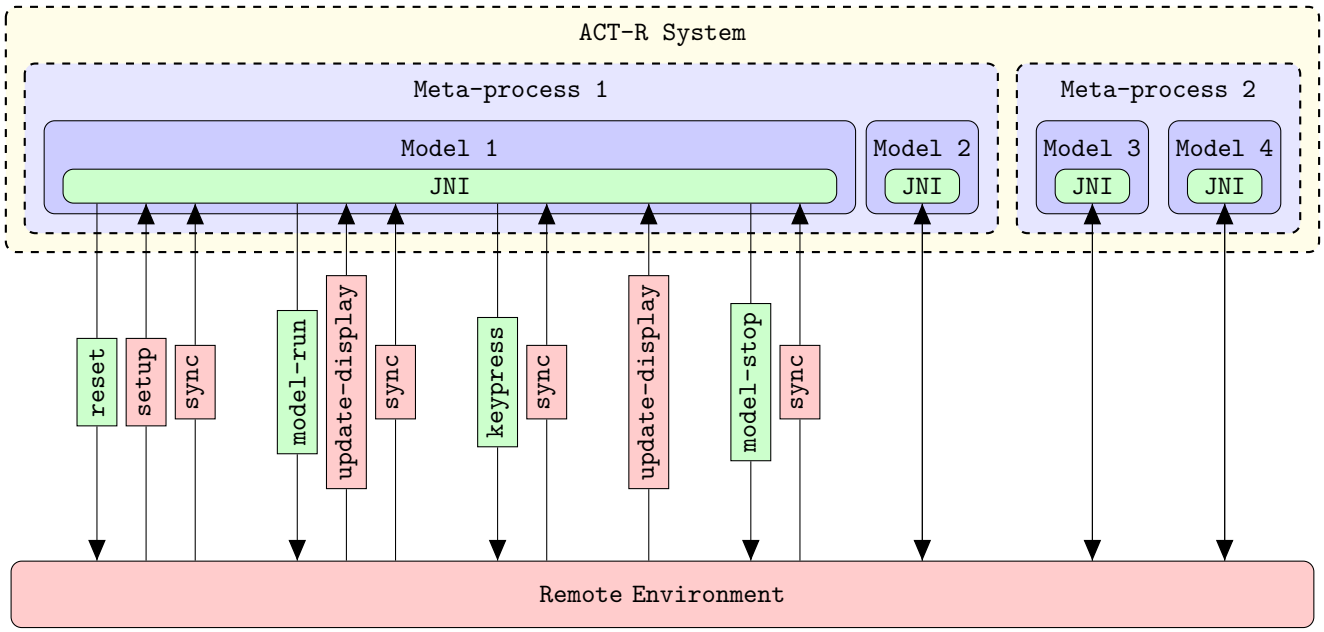

Figure 1: A schematic depicting how the JNI module fits into the ACT-R architecture as well as an example of the communication flow between the JNI module and remote environment. Nodes with rounded corners represent system (ACT-R or remote environment) components, square corner nodes represent API commands. Each model has its own instance of the JNI module which communicates with the environment, independent from and in parallel with other models, via TCP. These bi-directional connections are represented by the black double headed arrow lines for models 2-4. For model 1, an example sequence of API commands (ordered from left to right) is shown in place of the double headed arrow. The color of the API command nodes are color matched to the sending system component.

serialized to strings must be terminated with a carraige return and linefeed before being transmitted over TCP.

The complete set of API commands can be broken up into two groups: model commands and environment commands. Model commands are commands that could be sent from the JNI module to the environment. Most environments will need to implement handlers for all model commands. There is one JNI command that is completely optional. Environment commands are commands that could be sent from the environment to the JNI module during model execution. For a quick overview of the available API commands see Table 1. The following two subsections describe in detail the available model and environment commands.

2.1 Model commands

2.1.1 keypress

This command is used by the JNI module to inform the environment that a keyboard button has been pressed. The command takes two keyed parameters, the ASCII key code (*keycode*) and the unicode character corresponding to the button press (*unicode*). The environment should update itself as appropriate when it receives this command. See Example 1 for an example of this command.

```
{"model": "myModel", "method": "keypress",
 "params": {"keycode": 65, "unicode": "A"}}
```
(1)

Table 1: A summary of the available model and environment commands supported by the JNI module.

Model Commands		Environment Commands	
Command	Argument(s)	Command	Argument(s)
keypress	keycode, unicode	sync	
mousemotion	loc	update-display	loc-chunks, obj-chunks, clear
mouseclick	button	trigger-reward	reward
speak	message	set-cursor-loc	loc
reset	time-lock	new-digit-sound	digit
model-run	resume	new-tone-sound	frequency, duration
model-stop		new-word-sound	words
set-mp-time	time	new-other-sound	content, onset, delay, recode
		setup	width, height
		trigger-event	event, args

2.1.2 mousemotion

This command is used by the JNI module to inform the environment that the mouse has moved. The command takes one keyed parameter (*loc*), the x and y location in screen coordinates of the mouse's new location as an array. When the environment receives this command it should, at the very least, store the location because mouse click events do not contain the location of the click. See Example 2 for an example of this command.

```
{"model": "myModel", "method": "mousemotion",
 "params": {"loc": [342, 563]}}
```
(2)

2.1.3 mouseclick

This command is used by the JNI module to inform the environment that a mouse button has been pressed. The command takes one keyed parameter (*button*), the ID (button number) of the mouse button clicked. See Example 3 for an example of this command.

```
{"model": "myModel", "method": "mouseclick",
 "params": {"button": 1}}
```
(3)

2.1.4 speak

This command is optional and is typically not used by many models. If the environment has a text to speech (TTS) engine available, this command can be used by the JNI module to convert text to an audible voice. The command takes one parameter (*message*), a string containing the word(s) to be spoken. See Example 4 for an example of this command.

```
{"model": "myModel", "method": "speak",
 "params": {"message": "Quick brown fox."}}
```
(4)

2.1.5 reset

This command is used by the JNI module to inform the environment that the model has been reset. When the environment receives this command it should reset all its state as well. This command takes one keyed parameter (*time-lock*), a boolean, which when true indicates that the time-lock timing mode is being used.

When this parameter is false, the synchronous or asynchronous timing mode is being used. See Example 5 for an example of this command.

```
{"model": "myModel", "method": "reset",
 "params": {"time-lock": false}}
```
(5)

2.1.6 model-run

This command is used by the JNI module to inform the environment that the model is about to start running. This command takes one keyed parameter (*resume*), a boolean, which when true indicates that the model is resuming a previous run. See Example 6 for an example of this command.

```
{"model": "myModel", "method": "model-run",
 "params": {"resume": false}}
```
(6)

2.1.7 model-stop

This command is used by the JNI module to inform the environment that the model has stopped running. This command takes no parameters. See Example 7 for an example of this command.

```
{"model": "myModel", "method": "model-stop",
 "params": {}}
```
(7)

2.1.8 set-mp-time

This command is used by the JNI module to inform the environment of the current meta-process time. This command is only used in the time-locked timing mode and takes one keyed parameter (*time*), the current meta-process time. See Example 8 for an example of this command.

```
{"model": "myModel", "method": "set-mp-time",
 "params": {"time": 5.7}}
```
(8)

2.2 Environment commands

2.2.1 sync

This command must be sent in response to any model command, regardless of the current timing mode; it takes no parameters. See Example 9 for an example of this command.

```
{"model": "myModel", "method": "sync",
 "params": {}}
```
(9)

2.2.2 update-display

This command use by the environment to update the JNI module's knowledge of the location of all visual objects. The command takes three keyed parameters. The first parameter (*visual-location-chunks*) takes an array of JSON encoded visual-location chunks. The second command (*visual-object-chunks*) takes an array of JSON encoded visual-object chunks equal in length to the visual-location-chunks array. The structure of a chunk in JSON is an object with two key-value pairs. The first key is *isa* and its corresponding value should be a string describing its type. The second key is *slots* and its corresponding value is another JSON object whos key/value pairs are the slot and slot values of the chunk. For slot values only, strings prefixed with a colon will be treated as a chunk. The third parameter (*clear*) is a boolean which if set to true instructs the JNI to clear the visicon before processing the new visual chunks. See Example 10 for an example of this command. This command should be called whenever visual information changes in the environment.

```
{"model": "myModel", "method": "update-display",
 "params": {
  "visual-location-chunks": [
   {"isa": "visual-location", "slots": {"screen-x": 100, "screen-y": 200}},
   {"isa": "visual-location", "slots": {"screen-x": 300, "screen-y": 400}}],
  "visual-object-chunks": [
    {"isa": "visual-object", "slots": {"value": "hello"}},
    {"isa": "visual-object", "slots": {"value": ":world"}}],
  "clear": true}}
```
(10)

2.2.3 trigger-reward

This command is used by the environment to reward the model. The command takes one keyed parameter (*reward*), the value of the reward. See Example 11 for an example of this command.

```
{"model": "myModel", "method": "trigger-reward",
 "params": {"reward": 10}}
```
(11)

2.2.4 set-cursor-loc

This command is used by the environment to inform the JNI module of the environment's cursor location. The command takes one keyed parameter (*loc*), the x and y location of the cursor in screen coordinates as an array. See Example 12 for an example of this command.

```
{"model": "myModel", "method": "set-cursor-loc",
 "params": {"loc": [254, 355]}}
```
(12)

2.2.5 new-digit-sound

This command is used by the environment to create auditory chunks corresponding to digits. The command takes one keyed parameter (*digit*), a number representing the digit to be heard. See Example 13 for an example of this command.

```
{"model": "myModel", "method": "new-digit-sound",
 "params": {"digit": 8}}
```
(13)

2.2.6 new-tone-sound

This command is used by the environment to create auditory chunks corresponding to a particular frequency. The command takes two keyed parameters. The first parameter (*frequency*) is a number representing the frequency of the tone to be heard. The second parameter (*duration*) is a number representing the duration of the tone to be heard. See Example 14 for an example of this command.

```
{"model": "myModel", "method": "new-tone-sound",
 "params": {"frequency": 440, "duration": 1}}
```
(14)

2.2.7 new-word-sound

This command is used by the environment to create auditory chunks corresponding to a particular frequency. The command takes one keyed parameter (*words*), a string which is the word (or words) that will be heard. See Example 15 for an example of this command.

```
{"model": "myModel", "method": "new-word-sound",
 "params": {"words": "hello world"}}
```
(15)

2.2.8 new-other-sound

This command is used by the environment to create auditory chunks corresponding to a particular frequency. The command takes four keyed parameters. The first parameter (*content*) is is the content of the sound and can be any valid JSON value. The second parameter (*onset*) is a number which represents the amount of time between the onset and when the sound stops. The third parameter (*delay*) is a number which represents the content delay for the sound. The forth parameter (*recode*) is a number which represents the recode time for the sound. See Example 16 for an example of this command.

```
{"model": "myModel", "method": "new-other-sound",
 "params": {"content": "Boom!", "onset": 4, "delay": 0.1, "recode": 0.5}}
```
(16)