A. JAMSH

A.1 Overview

The *JAM* Shell *jamsh* is a command line interpreter that provides direct access to the *JAM* library *jamlib* implementing the core JavaScript Agent Machine. Commands can be executed either from command line (of the shell) or by a script. The *JAM* shell is available for native execution as CLI program[2] using *nodejs* or similar CLI JS VMs and as a WEB browser implementation[1] (*webui*) as an IDE framework.

A.2 Shell Architecture

The *jamsh* atchitecture consists of:

- *JAM* library *jamlib* embedding the *AIOS*
- *JAM* shell interpeter
- A world provided by *jamlib* consisting of at least one virtual node (more can be added at run-time)

The following architecture diagram Dia. 1 shows the relevant blocks and a world consiting of different virtual nodes with agents bound to a virtual node. All virtual nodes are handled by one agent scheduler provide by the *AIOS*.

The *JAM* library provides Internet connectivity to link multiple *JAM* platforms via the Agent Monitor Protocol (AMP).

The *JAM* shell can operate in two modes:

1. Interactive loop mode executing user commands via an extended command line with history
2. Script mode executing commands from a script file

Both modes can be combined.

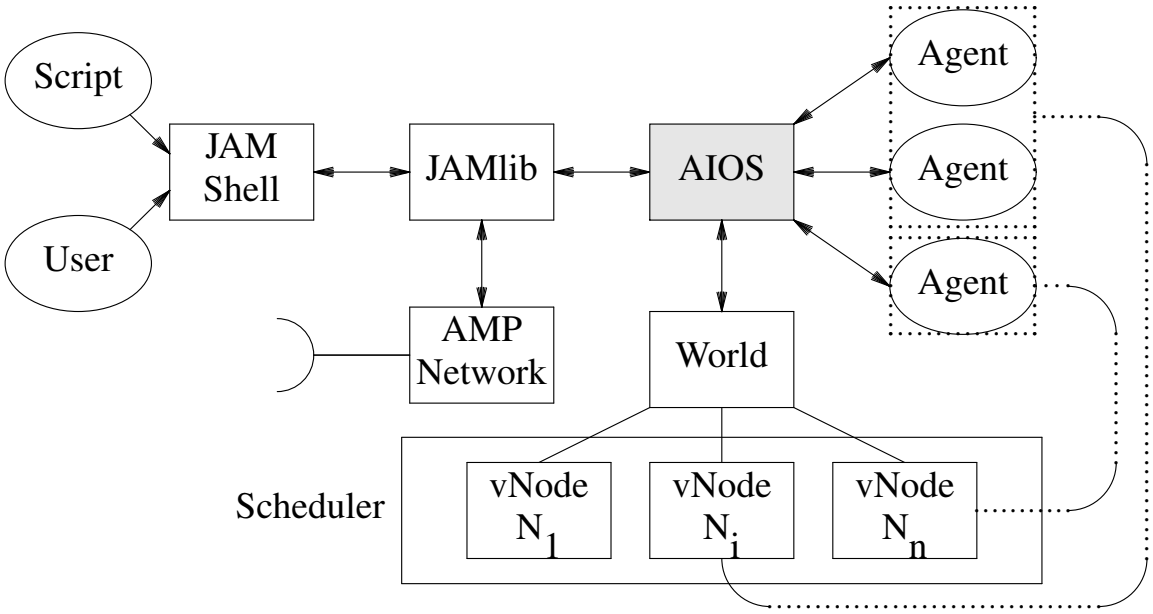

Figure 1. JAM shell architecture diagram

A.3 Shell Commands

The following shell commands are available:

add

```
function ({x:number, y:number})
```

Adds a new logical (virtual) node to the current world.

ask[2]

```
function (question:string,
          choices: string [])
          → string
```

Asks a question and read answer. Available only in script mode in CLI version.

broker[2]

```
function (ip:string)
```

Starts a SIP UDP broker server

Capability

```
function (port, private)
         → capability
```

Creates a security capability object consisting of a public server port and a private field containing an optional object number, a rights field, and a protection port validating the private part. The following sub-functions are available:

```
type capability = { cap_port: port, cap_priv:privat }
function Capability.toString (capability) → string
function Capability.ofString (string) → capability
```

The capability string format is:

```
[PP:PP:PP:PP:PP:PP](O(RR)[RR:RR:RR:RR:RR:RR])
```

clock

```
function (ms)
         → number
```

Returns system time (ms or `hh:mm:ss` format).

cluster

```
constructor cluster(desc:{}) → Cluster
```

Creates a *JAM* shell worker cluster.

config

```
function(options)
```

Configures *JAM* (AIOS). Available options:

- *print*
- *printAgent*
- *TSTMO* sets global tuple timeout (even if with *out* inserted)

connect

```
function ({x:number, y:number},
          {x:number, y:number})
```

Connects two logical nodes.

connect

```
function (to:dir)
```

Connects two physical nodes. Requires the setup of an appropriate port (using the `port` creation operation). E.g., all IP connections requires an IP port. A connection in South direction (`DIR.NORTH`) requires a `DIR.NORTH` port.

```
port(DIR.IP(10001),{proto:'udp'})
connect(DIR.IP('134.102.219.4:10001'))
// using HTTP proto in client mode
port(DIR.IP(),{proto:'http'})
connect(DIR.IP('134.102.219.4:10002'))
// using HTTP proto in server mpde
port(DIR.IP(10002),{proto:'http'})
connect(DIR.IP('134.102.219.4:10002'))
```

Examples for connecting to remote endpoints

connected

```
function (to:dir)
  → boolean|null|string []
```

Checks connection between two nodes

compile

```
function (function|string, name?:string,
options?:{})
```

Compiles an agent class constructor function (code or text). Compilation includes analysing the source code and creating of a sand-boxed constructor function with a private environment. The name of the class is optionally. The options argument can specify the verbosity of the analyser using the attribute `verbose:number`.

concat

```
function (@a, @b)
         → @c
```

Concatenate two values (strings, arrays or objects).

contains

```
function (@a, @v)
         → boolean
```

Checks if an array or object contains a value or an array of values.

create

```
function (ac:string,
          args:*[]|{},
          level?:number,
          node?)
```

Creates an agent from class `@ac` with given arguments `@args` and `@level`

csp?

Constraint Solving Programming API (optional)

csv?

The CSV module provides oeprations for reading and writing data in CSV format. The supported data matrix type is defiend below.

```
typeof csv = {
  read:  function (file:string, convert:boolean)
         → error|data,
  write: function (file, header: string [],
                   data, sep?:string)
         → length:number
}
type data = {$x:number|string|boolean}[]
```

disconnect

```
function (to:dir)
```

Disconnects a logical node or a remote endpoint of a physical node

env[2]

Shell environment including command line arguments `attr:value`

empty

```
function (@a)
  → boolean
```

Tests empty string, array, or object.

exec

```
function (cmd:string)
```

Executes a JAM shell command.

exit[2]

Exits the shell.

extend

```
function (level:number|number[],
          name:string,
          function,
```

```
          argn?:number|number[])
```

Extends the AIOS with user defined functions. The function can be registered for a specific AIOS execution level (0-3) or a range [0-3].

filter

```
function (@a,
          f:function)
         → b
```

Filters array or object using a user function return Boolean values or *none* and a value (combined mapping and filtering).

http.get[2]

```
function (ip:string,
          path:string,
          callback?:function)
```

Serves HTTP get request

http.put[2]

```
function (ip:string,
          path:string,
          data,
          callback?:function)
```

Serves HTTP put request

http.server[2]

```
function (ip,
          dir,
          index?)
```

Creates and starts a HTTP file server

inp

```
function (pattern:[],
          all:boolean)
```

Reads and remove (a) tuple(s) from the tuple space of the current node.

kill

```
function (id:string)
```

Kills an agent (*id*="*" kill all agents) immediately.

info

```
function (kind:"node" |
          "version" |
          "host" |
          "agent" |
          "agent-data",
          id?:string)
         → info {}
```

Returns information about node, versions, host, agent, or agent process.

lookup[2]

```
function (pattern:string,
          callback:function (string [])
```

Asks broker for registered nodes (requires broker connection)

load

```
function (path:string)
         → {}
```

Loads a JSON file and returns object. (Browser: Only files located in the HTML root directory or below can be loaded via the browser).

log

```
function (msg:string)
```

Agent logging function

mark

```
function (pattern:[],
          tmo:number)
```

Stores a tuple with timeout in the tuple space of the current node.

ml?

Machine Learning API (optional)

name

```
function ("node" | "world")
         → string
```

Returns name of current node or world

nn?

Artificial Neural network API (optional)

neg

```
function (v) → v
```

Negates number, array or all attributes of an object.

ofJSON

```
function (string) → object
```

Converts JSON to object including functions

on

```
function (event:string,
          handler:function)
```

Installs an event handler. Supported events: "agent+", "agent-", "signal+", "signal", "link+", "link-"

open

```
function (file:string,
         verbose?:number)
```

Opens an agent class file using the native file system interface. The WEB browser version uses XHTTP requests to load files.

out

```
function (tuple:[])
```

Stores a tuple in the tuple space of the current node.

port

```
function (dir,
          options?,
          node?)
```

Creates a new physical communication port.

```
port(DIR.IP(10001),{proto:'udp'})
port(DIR.IP(),{proto:'http'})
port(DIR.IP(10002),{proto:'http'})
```

Examples of communication ports

Port

```
function (number [])
```

```
          → port
```

Creates a capability port from a number array (Six elements). The following sub-functions are available:

```
type port = string[6]
function Port.toString(port) → string
function Port.ofString(string) → port
function Port.prv2pub(port) → port
function Port.unique() → port
function Port.equal(port,port) → boolean
```

Private

```
function (obj:number[0..65535],
          rights:number[0..255],
          port)
          → privat
```

Creates a security private object consisting of containing an optional object number, a rights field, and a protection port validating the private part. The following sub-functions are available:

```
type privat = {
  prv_obj:number,
  prv_rights:number,
  prv_rand:port }
function Private.decode (privat,
                         rand:port)
          → boolean
function Private.encode (obj:number,
                         rights:number,
                         rand:port)
          → privat
function Private.equal (privat,
                        privat)
          → boolean
function Private.number (privat) → number
function Private.ofString (privat) → string
```

```
function Private.restrict (privat,
                           mask:number,
                           rand:port)
         → privat
function Private.rights (privat) → number
function Private.rights_check (privat,rand:port,
                               required:number)
         → boolean
function Private.toString (privat) → string
```

A server keeps a private random port that is used to encode, restrict (rights) and decode (check for validity) a private field of a capability.

The private field string format is: `O(RR)[RR:RR:RR:RR:RR:RR]`

provider

```
function (function(pattern)
          → null | tuple)
```

Registers a new tuple provider function.

rd

```
function (pattern:[],
          all:boolean)
  → null | tuple | tuple []
```

Reads (a) tuple(s) from the tuple space of the current node.

reverse

```
function('a) → 'b
```

Reverses array or string

rm

```
function (pattern:[],
          all:boolean)
```

Removes (a) tuple(s) from the tuple space of the current node.

sat?

Logic (SAT) solver API (optional)

script

```
function (file:string)
```

Loads and execute a jam shell script using the native file system API or XHTTP requests in the WEB browser version (Browser: only files in the HTML root directory or below can be loaded).

setlog

```
function (flag:string,
          on:boolean)
```

Enables or disables logging attributes

signal

```
function (to:aid,
          sig:string|number,
          arg?:*)
```

Sends a signal to specified agent

start

```
function ()
```

Starts *JAM* agent processing (enables scheduler and various managers).

stats

```
function (kind:"process" |
               "node" |
               "vm")
```

Returns statistics

stop

```
function ()
```

Stops JAM agent processing (disables scheduler)

sql?

```
function (path,options)
```

Opens or creates an SQL database. A memory DB can be created with filename=":memory:". Requires either native sqlite3 plug-in or built-in sqlite3 JS module.

test

```
function (pattern)
         → boolean
```

Tests a tuple pattern for matching tuples.

ts

```
function (pattern:[],
          function(tuple) → tuple)
```

Updates a tuple in the tuple space (atomic action) using the supplied update function - non-blocking.

time

```
function () → string
```

Returns AIOS platform time in milli seconds.

toJSON

```
function (object)
         → string
```

Converts object including functions to JSON formatted string.

verbose

```
function (level:number)
```

Sets verbosity level

versions

```
function() → {}
```

Returns JAM shell and library version

A.4 Networking

Networking consists of the creation of ports and links between ports (and *JAM* nodes). Commonly multi-cast IP ports are used in the Internet domain. A multi-cast IP port can connect with an arbitrary number of IP ports of remote *JAM* nodes. All ports provide an Agent Management Port (AMP) interface used to transfer agent code, signals, tuples, and control messages between JAM nodes. In the Internet or Intranet domain there are three different communication protocols that can be used to transport AMP messages: UDP, TCP, and HTTP. The HTTP protocol distinguishes pure clients (that can connect to other remote ports only) and server (supporting remote connecting). The HTTP client mode is used in WEB Browser only. Different IP ports using different protocols can coexist on a *JAM* node. All IP ports are handled by an internal router.

Note: An IP port can be defined by using the `DIR.IP(&quot;ip:ipport&quot;)` directional type. Other port directions like `DIR.NORTH(&quot;ip:port&quot;)` can be used, too. But these port directions support uni-cast communication only.

A.4.1 UDP Unicast ports and links

```
port(DIR.IP(ip:number|string="<ip>:<port>"),
     {proto:'udp',multicast:boolean,verbose:number});
connect(DIR.IP(ip:number|string="<ip>:<port>"));
```

A.4.2 UDP Multicast ports and links

```
port(DIR.IP(ip:number|string="ip:ipport"),
     {proto:'udp',verbose:number});
conenct(DIR.IP(ip:number|string="ip:ipport"));
```

A.4.3 UDP Multicast ports using broker service

Client Side

```
port(DIR.IP("*"),{proto:'udp',broker:"ip:ipport",
                  name:'/domainX/'+name("node"),
                  multicast:boolean,verbose:number});
connect(DIR.IP("/domainX/B"));
lookup(DIR.PATH('/domainX/*'),function (result) {
  log('lookup: '+result)
});
```

Broker Server

```
broker("<ip>:<port>");
```

A.5 Port Scanning

```
// Node 1
port(DIR.IP(10001),{proto:'http'})
// Node 2
p=port(DIR.IP(11001),{proto:'http'})
p.amp.scan({address:'localhost',port:10001,null,function (reply
  print(reply) // {type:12, port:'      '}
})
```

A.6 Clusters

A *JAM* shell can create a worker process cluster.

```
type desc =
  {
    id:string,
    ports:{port:number,
           proto:'http'|
                 'udp'|
                 'tcp'} []
    todo:string,
    poll?: function (process)
  } []
constructor cluster(desc) → Cluster
Cluster.methods = {
  check: function (index?:number) -> stats,
  poll: function (index?:number),
  restart: function (index?:number),
  start: function (index?:number),
  stop : function (index?:number),
}
```

A.7 Examples

```
// Agent Class Construtor
function fib(args) {
  this.todo = args.val;
  this.output = [];
  this.f = function(n) {
        return n < 2 ? n : this.f(n-2) + this.f(n-1)
  }
  this.act = {
    calculate: function() {
      var n = head(this.todo)
      this.todo = filter(this.todo,
                         function(elem)
```

```
                          { return elem != n })
      var result = this.f(n)
      this.output.push(result)
    },
    print: function() {
      var next = head(this.output)
      this.output = filter(this.output,
                           function(elem)
                           { return elem != next })
      log(next)
    }
  }
  this.trans = {
     calculate: function() {
       return empty(this.todo) ? print : calculate
     },
    print: function() {
      if(empty(this.output)) {
        log('Killing agent')
        kill()
      }
      return print
    }
  }
  this.next = calculate
}
// Compile agent class and add it to the world library
compile(fib)
// Start JAM scheduler loop
start()
// Create an agent from already compiled class
create('fib', {val: [10, 5]})
```

Example. Agent class compilation from function and creation of agents

```
// Start explorer agent
port(DIR.IP())
connect(DIR.IP(10002))
open('explorer.js',0)
var dialog = {
```

```
  dialog:[
    {question:'Where are you?',
     choices:['Street 1','Street 2','Street 3',
              'Anywhere','Other place']},
    {question:'How do you rate ambient light?',
     choices:['Dark','Good','Bright']}
  ],
  action: function (dialog) {
    var place;
    switch (dialog[0].answer) {
      case 'Street 1':
      case 'Street 2':
      case 'Street 3':
        place=dialog[0].answer; break;
    }
    if (place) switch (dialog[1].answer) {
      case 'Dark': return {light:150,place:place};
      case 'Bright': return {light:50,place:place};
    }
  }
}
var id=create('explorer',dialog)
print('Started agent '+id)
start()
print("Done.")
```

Example. Agent class compilation from file and creation of agents