Chapter 8

JAM: The JavaScript Agent Machine

Agent Processing Platform based on JavaScript

JAM: The JavaScript Agent Machine 264
AgentJS: The Agent JavaScript Programming Language 265
AIOS: The Agent Execution and IO Environment 268
JAM Implementations 275
Performance Evaluation 283
SEJAM: The JavaScript Agent Simulator 289
Heterogeneous Environments 291
Further Reading 292

epubli, ISBN 9783746752228 (2018)

This Chapter introduces a programmable agent processing platform that is capable of processing *AAPL* based agents entirely programmed in *JavaScript*. The *JavaScript* Agent Machine (*JAM*) platform is programmed entirely in JavaScript, too, supporting the execution and migration of agents in heterogeneous environments including the Internet and the Internet-of-Things.

8.1 JAM: The JavaScript Agent Machine

The latest extension of the agent platform family was the Agent Forth Virtual Machine (*AFVM*) fully implemented in *JavaScript* including Browser implementations, which is capable of executing agent code using a stack-based *FORTH* machine language with *AAPL* agent-specific extensions [BOS15D]. The *AFVM* was optimized originally for low-resource environments, including single microchip implementations. One major feature of *AAPL* agents is the capability of reconfiguration and agent behaviour (re-)composition at run-time. The code-based agent platforms therefore support this by enabling and using code-morphing.

To simplify the development and deployment of multi-agent systems on the Internet *AAPL* agents should be directly implemented in *JavaScript* (*JS*), which is a well-regarded and public widespread used programming language. *JS* execution platforms are available for a very broad range of devices and operating systems, e.g., Intel *x86/x64*, *Arm32*, *Linux*, *Windows*, *Solaris*, *MacOS*, *FreeBSD*, *Android*, *IOS*, and many more. Furthermore, the implementations of mobile agents directly in *JS* would benefit from actually existing high-performance *JS* VMs, e.g., Googles *Chrome V8* or Mozillas *Spidermonkey* engines with Just-in-Time native code compilation (JIT). At a glance, *JS* is a very simple but highly dynamic language covering procedural, object-orientated, and functional concepts. Even if a JIT-based VM is used, full code-to-text and text-to-code transformation is preserved at any execution time, including functions and data. This enables the capability of code morphing at run-time, a prerequisite for *AAPL*-based agents, used to store the current state of an agent process (e.g., prior to migration) and to modify the behaviour of an agent by applying a re-composition to the ATG by the agent itself. In contrast to *JAVA* and common *JAVA*-based agent frameworks (e.g., *JADE*), *JS* has a loose coupling to and low dependencies of the underlying execution platform. This is a significant advantage over *JAVA* or *C* programming languages, which must be always compiled before the code can be executed, and being very sensitive for API and library mismatches. *JS* considers functions as first-order values, enabling code reconfiguration on-the-fly like any other data modification using the built-in `eval` function.

An overview of the *JAM* architecture and the composition of JavaScript modules is shown in Figure *8.1*. The core JAM platform implements the agent execution (left side of Figure *8.1*), and an optional Distributed Operation System (DOS) layer adds Internet connectivity (right side of Figure *8.1*).

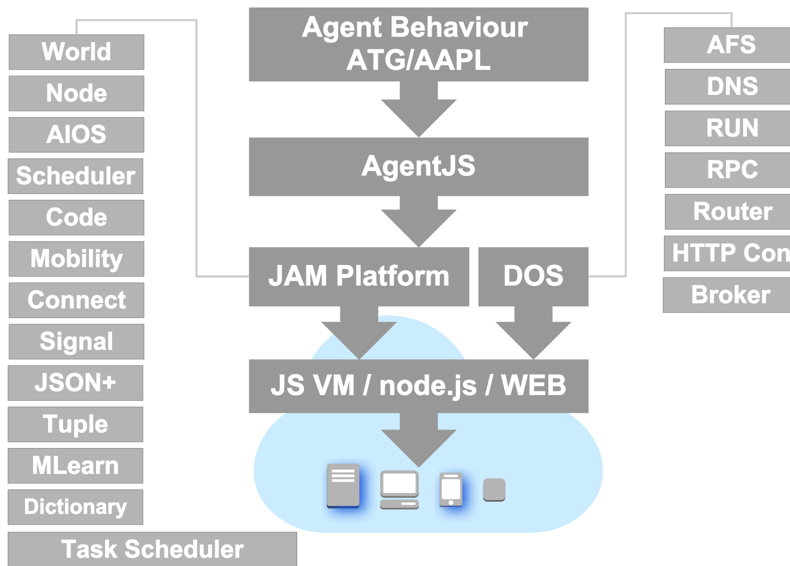

Fig. 8.1 *The JAM architecture and modules*

8.2 AgentJS: The Agent JavaScript Programming Language

AgentJS is common JavaScript aligning the JS object model with *AAPL* agent template classes. A comparison of *AgentJS* with the meta language *AAPL* is shown in Example 8.1. The programmatical and semantic structure of *AgentJS* is very close to *AAPL*. *AgentJS* consists of an agent class constructor function with some required attributes defining the agent activities, the transitions, auxiliary functions, and agent data.

8.2.1 AgentJS: The JavaScript Object and extended Code-to-Text JSON+ Representation

Textual representations used as a data and code interchange format is a prerequisite for data and code processing in strong heterogeneous platform and network environments, mixing big- and little endian machines, different data word sizes, and data coding. Though byte-code based interchange formats are widely used, they require a strict compliance of the coding between a sender and a receiver. At any time, a *JS* object can be converted to text in *JSON* format at run-time. Originally, *JSON* was introduced for portable exchange of *JS* data objects in a textual representation only, being much more compact and easier to interpret than *XML*. A *JAM/AgentJS* agent is basically a *JS* object containing data (values, data objects, arrays) and functions, representing the agent activities and transitions of the ATG, requiring an extended *JSON*

text formatter and parser supporting functions, which was introduced in *JAM*. An entire agent process can be converted at any time to the textual representation (*JSON+*) preserving its current control and data state, which can be exchanged by different network and agent platform nodes, and that is finally back converted to *JS* code. The only existing limitation are circular (self) references inside of an object, which still cannot be handled, but not being a real restriction. Transferring text instead of binary code results in a significantly increased communication cost on agent migration, but the text can be compressed reducing the size significantly (experiments showed that *LZ* compressing reduces the *JSON+* text size and hence the communication costs about 5-6 times). Embedded devices can utilize hardware compressor modules, e.g., using FPGA-based co-processors, maximizing communication efficiency without additional CPU costs.

8.2.2 The AgentJS Sandbox Environment

Stability and robustness of the agent processing platform is one major challenge in the design of those platforms. Agents can be considered as autonomous or semi-autonomous processes and execution units. But this autonomy requires strictly bounded and safe platform environments for the execution of agent processes, and the strict isolation of agent processes from each other. An agent platform must be capable to execute hundreds and thousands of different agent processes. Although there are extension modules for some *JS* VMs (e.g., *webworker*) allowing the execution of a *JS* program in a separate host process (or thread), this method is importable and is creates significant overhead in time and memory space. Unfortunately, *JS* has only a very limited scoping mechanism, basically limited to function closures and the *this* object, and with one global space shared by all imported modules and evaluated code. This limitation initially prohibits the safe and interference-free execution of multiple agent processes within one *JS* VM. But fortunately, *JS* provides the `with (mask) {code}` statement, executing the code with an additional new overlaid name space given by the *mask* object argument. This cannot limit the name space scope (scopes are chained, and higher scopes like the global one are still visible), but it can be used to override higher scope level and global name qualifiers, and to invalidate references to free variables and functions without compromising other agent processes or the *JAM* modules.

So basically the agent process execution is an execution of a function with a strictly limited visible name space without any bindings to external and free variables and functions. To ensure this, the *JSON* parsing and evaluation is always performed inside the `with` statement with a mask environment only providing a selected *AIOS* set of objects and functions, discussed in Section 8.2. A creation of a new agent will always first stringify the agent object, and finally coding back a sand-boxed agent object free of any free and global

object references, which can be executed the *AIOS* agent scheduler without any interference with the platform and other agents. This approach protects the agent execution and *JAM* at least against failures by accident using common *JS* coding styles. The capability of full intrusion protection depends on the *JS* VM environment itself.

8.2.3 AgentJS-AAPL Relationship

AgentJS is a modified and restricted *JS* programming and object model, which can be directly executed by any *JS* VM using the *AIOS* execution layer. The *AAPL* model can be directly mapped on this *AgentJS* model without further transformation steps, shown in Example 8.1. The only exception is the decomposition of activities in scheduling blocks if they contain blocking statements (e.g., line 12), except one blocking tail-statement at the end of an activity, that do not require encapsulation in a scheduling block. Transition functions may not block, otherwise an exception is raised.

There is a significant difference between *AgentJS* and common *JS* programs: The `this` object used inside *AgentJS* activity, transition, callback, and first-order function calls of *AIOS* provided functions is always bound to the agent object itself! Due to the *JSON+* code-text transformation, there cannot be any free variables inside an agent object (the references would be lost on transformation and migration), including the commonly used `self` variable. Finally, there may no cyclic object links and objects posing method prototypes (i.e., only data structures can be created and used by an agent).

Ex. 8.1 *A simple neighbourhood explorer agent programmed in AAPL (left) and the corresponding AgentJS code (right). Note: In AgentJS* `this` *always references the agent object, even in deeper context levels.*

```
1   agent explorer(dir,radius)          function explorer(dir,radius) {
2     var x,y:int;                        this.dir=dir; this.radius=radius;
3     var mean,hop:int;                   this.x=0; this.y=0; this.mean=0;
4     var goback:boolean;                 this.hop=0; this.goback=false;
5     activity init = ..                  this.act = {
6     end;                                 init:function() {..},
7     acitivity move =                     move:function() {
8       if (hop=radius) goback := true;     if (this.hop==radius)
9       else                                  this.goback=true;
10      begin hop++; moveto(dir);           else {this.hop++;
11      end;                                        moveto(this.dir);}
12    end;                                 },
13    activity percept =                   percept: function () { var s;
14      var s:int; rd(SENSOR,s?);           B([function () {
15                                              rd(['SENSOR',_],
16                                                  function(t) {s=t[1]}),
17      mean := (mean+s)/2;                    function () {mean = (mean+s)/2}]);
18    end;                                 },
```

```
19    activity goback =                    goback: function () {
20      dir=opposite(dir);                  this.dir=opposite(this.dir);
21    end;                                 },
22    activity deliver =                   deliver: function () {
23      out(MEAN,mean);                      out(['MEAN',this.mean]);
24      signal($parent,DELIVER);             signal(parent(),DELIVER);
25    end;                                  }};
26    handler S(v) = .. end;              this.on = { S:function(v) {..} .. };
27    transitions =                       this.trans = {
28      init->move;                        init: function () {return move},
29      move->percept:not goback;          move: function () {
30      move->move:goback and hop>0;        if (!this.goback) return percept;
31      move->deliver:goback and hop=0;     else if (this.goback && this.hop>0)
32      move->goback:hop=radius;              return move;
33      percept->move;                      else if (this.goback && this.hop==0)
34      goback->move;                         return deliver;
35      deliver->end;                       else if(hop==radius)
36                                            return goback},
37    end;                                 goback: move,
38  end                                    deliver: end
39                                        };
40                                        this.next=init};
```

8.3 AIOS: The Agent Execution and IO Environment

The *AIOS* is the main execution layer of *JAM*. It consists of the sandbox execution environment encapsulating an agent process, with different privileged sub-sets depending on the agent role level (0,1,2). Furthermore, the *AIOS* module implements the agent process scheduler and provides the API for the logical (virtual) world and node composition. The sandbox environment provides restricted access to a code dictionary based on the privilege level, enabling code exchange between agents. The *AIOS* is the interface and abstraction layer between agents programmed in *AgentJS* and the agent processing platform (*JAM*). Furthermore, it provides an interface between host applications and *JAM*, shown in Figure 8.2.

The AIOS consists of various modules, with the most common modules:

- Agent Manager and Scheduler (AS)
- Tuple Space Module and data base (TS)
- Agent signal management module (SI)
- Agent processing module (PR)
- Code Morphing and reconfiguration module (CD)
- Node and world management module (ND)
- Node communication module (CM)

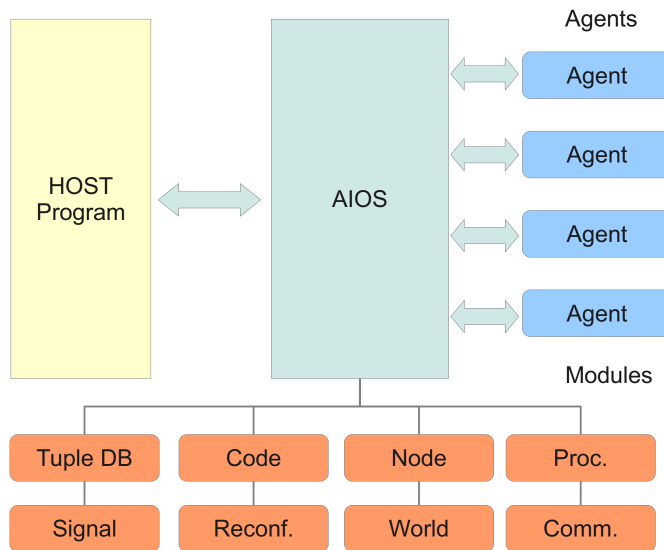

Fig. 8.2 *The Agent Input-Output System interface between agents and the platform*

8.3.1 Agent Scheduling and Check-pointing

JS has a strictly single-threaded execution model with one main thread, and even by using asynchronous callbacks, these callbacks are executed only if the main thread (or loop) terminates. This is the second hard limitation for the execution of multiple agent processes within one *JS JAM* platform. Agents processes are scheduled on activity level, and a non-terminating agent process activity would block the entire platform. Current *JS* execution platform including VMs in WEB browser programs provide no reliable watchdog mechanism to handle non-terminating *JS* functions or loops. Although some browsers can detect time-outs, they are only capable to terminate the entire *JS* program. To ensure the execution stability of the *JAM* and the *JAM* scheduler, and to enable time-slicing, check-pointing must be injected in the agent code prior to execution. This step is performed in the code parsing phase by injecting a call to a checkpoint function `CP()` at the beginning of a body of each function contained in the agent code, and by injecting the `CP` call in loop conditional expressions. Though this code injection can reduce the execution performance of the agent code significantly, it is necessary until *JS* platforms are capable of fine-grained check-pointing and thread scheduling with time slicing. On code-to-text transformation (e.g., prior to a migration request), all `CP` calls are removed.

AIOS provides a main scheduling loop. This loop iterates over all logical nodes of the logical world, and executes one activity of all ready agent pro-

cesses sequentially. If an activity execution reaches the hard time-slice limit, a `SCHEDULE` exception is raised, which can be handled by an optional agent exception handler (but without extending the time-slice). This agent exception handling has only an informational purpose for the agent, but offers the agent to modify its behaviour. All consumed activity and transition execution times are accumulated, and if the agent process reaches a soft run-time limit, an `EOL` exception is raised. This can be handled by an optional agent exception handler, which can try to negotiate a higher CPU limit based on privilege level and available capabilities (only level-2 agents). Any ready scheduling block of an agent and signal handlers are scheduled before activity execution.

After an activity was executed, the next activity is computed by calling the transition function in the transition section.

In contrast to the *AAPL* model that supports multiple blocking statements (e.g., IO/tuple-space access) inside activities, *JS* is incapable of handling any kind of process blocking (there is no process and blocking concept). For this reason, scheduling blocks can be used in *AgentJS* activity functions handled by the *AIOS* scheduler. Blocking *AgentJS* functions returning a value use common callback functions to handle function results, e.g., `inp(`*pat*`, function(`*tup*`){..})`.

A scheduling block consists of an array of functions (micro activities), i.e., `B(`*block*`) = B([function(){..}, function(){..},...])`., executed one-by-one by the *AIOS* scheduler. Each function may contain a blocking statement at the end of the body. The `this` object inside each function always reference the agent object. To simplify iteration, there is a scheduling loop constructor `L(`*init*`, `*cond*`, `*next*`, `*block*`, `*finalize*`)` and an object iterator constructor `I(`*obj*`, `*next*`, `*block*`, `*finalize*`)`, used, e.g., for array iteration.

Agent execution is encapsulated in a process container handled by the AIOS. An agent process container can be *blocked* waiting for an internal system-related IO event or *suspended* waiting for an agent-related AIOS event (caused by the agent, e.g., the availability of a tuple). Both cases stop the agent process execution until an event occurred.

The basic agent scheduling algorithm is shown in Algorithm 8.1 and consists of an ordered scheduling processing type selection, i.e., partitioning agent processing in agent activities, transitions, signals, and scheduling blocks. In one scheduler pass, only one kind of processing is selected to guarantee scheduling fairness between different agents. There is only one scheduler used for all virtual (logical) nodes of a world (a *JAM* instance). A process priority is used to alternate activity and signal handling of one agent, preventing long activity and transition processing delays due to chained signal processing if there are a large number of signals pending.

Alg. 8.1 *AIOS Agent Scheduler Algorithm [process: agent execution container, block: external system scheduling block(s), schedule: agent scheduling block(s), signals: list of pending agent signals, blocked: process blocked and waiting for external system IO event, suspended: agent blocked and waiting for AIOS event]*

```
1   ∀ node ∈ world.nodes do
2    ∀ process ∈ node.processes do
3     • Determine what to do with prioritized conditions:
4       Order of operation selection:
5       0. Process (internal) block scheduling [block]
6       1. Resource exception handling
7       2. Signal handling [signals]
8          - Signals only handled if process priority < HIGH
9          - Signal handling increase process priority temporarily to
10           allow low-latency act/trans scheduling!
11      3. Transition execution
12      4. Agent schedule block execution [schedule]
13      5. Next activity execution
14         - Lowers process priority
15
16    if process.blocked or process.dead or
17     process.suspended and process.block=[] and process.signals=[] or
18     process.agent.next=none and process.signals=[] and process.schedule=[]
19    then do nothing
20    else if not process.blocked and process.block≠[]
21    then execute next block function
22    else if agent resources check failed
23    then raise EOL exception
24    else if process.priority < HIGH and process.signals≠[]
25    then handle next signal, increase process.priority
26    else if not process.suspended and process.transition
27    then get next transition or execute next transition handler function
28    else if not process.suspended and process.schedule≠[]
29    then execute next agent schedule block function
30    else if not process.suspended
31    then execute next agent activity and compute next transition,
32         decrease process.priority
33
```

8.3.2 Agent Roles

Security is another major challenge in distributed systems, especially concerning mobile agent processes. Each agent platform node (i.e, one physical VM, with multiple *JAM* VMs operating on the same network node) can receive agents originating either from inside trusting node networks or coming from untrustful networks unknown by the VM. Generally, the VMs have no information about other network nodes except a sub-set of network connectivity used to receive and propagate agent code. To distinguish at least trustful and not

trustful agents, different agent privilege levels were introduces, providing different *AIOS* API sets.

For security reasons and to limit Denial-of-Service attacks, agent masquerading, spying, or other misuse, agents have different access levels (roles). There are four levels:

0. Guest (not trusting, semi-mobile)
1. Normal (maybe trusting, mobile)
2. Privileged (trusting, mobile)
3. System (trusting, locally, immobile)

Privilege level 0 is the lowest level and grants agents only computational and tuple-space IO statements. The lowest level does not allow agent replication, migration, or the creation of new agents. Level 1 agents can access the common *AIOS* API operations and capabilities, including agent replication, creation, killing, sending of signals, and code morphing. Level 2 agents are additionally capable to negotiate (set) their desired resources on the current platform, i.e., CPU time and memory limits. An agent of level n may only create agents up to level n. The highest level (3) has an extended AIOS operation set with host platform device access capabilities. Agents can negotiate resources (e.g., CPU time) and a level raise secured with a capability-key that defines the allowed upgrades. The system level can not be negotiated. Level-2 agents can initially only be created inside the *JAM*. They can fork level-2 agents, but after a migration the destination node decides about the privilege level and can lower it, e.g., considering the agent source being not trustful. A migrated agent can get a higher privilege level by negotiation, requiring a valid platform capability with the appropriate rights. After migration, the privilege is lost and must be re-negotiated on a new platform using capabilities. The *JAM* platform decides the security level. The capability is node specific. A group of nodes can share a common key (identified by a server port). A capability consists of a server port, a rights field, and an encrypted protection field generated with a random port known by the server (node) only and the rights field.

Among the AIOS level, other constrain parameters can be negotiated:

- Scheduling time (maximal slice time for one activity execution, default is 20ms)
- Run time (accumulated agent execution time, default is 2s)
- Living time (overall time an agent can exist on a node before it is killed, default is 200s)

- Tuple space access limits
- Memory limits (practically fuzzy, usually the entire size of the agent code including private data, actually unlimited)

8.3.3 The Execution Platform and Networking

The *JAM* execution platform consists of different virtualization layers. Each physical *JAM* node (a program executed on a host platform, e.g., a smart phone or server) has a logical world consisting of logical nodes (at least one). Agent processes are bound to and executed on one logical node at any time. Logical nodes can be connected by using virtual circuit links (queues), and physical nodes can be connected by using peer-to-peer network connections (sockets, IP links, UART serial links, and so on) or the *DOS* layer introduced in the next section. Agents can migrate between logical and physical nodes. The entire *JAM* (excluding *DOS*) platform requires about 600kB *JS* text code only.

8.3.4 Agent Process Mobility and Migration

The control state of an agent is stored in a reserved agent body variable `next`, pointing to the next activity to be executed. The data state of an *AgentJS* agent consists only of the body variables. There are no references to variables outside the agent process context. Migration requires a snapshot of the agent process, in this case the agent itself, a code-to-text transformation, transportation of the text code to another logical or physical node, and a back text-to-code conversion with a new sandbox environment. The agent object is finally passed to the new node scheduler and can continue execution. Text code sizes of medium complex agents (with respect to data and control space) are reasonable low about 10kB, simpler agents tend to 1kB, that can be significantly reduced by using LZ compression. One drawback of this method raises with pending scheduling blocks existing still in the snapshot. They must be entirely saved in the migrated snapshot, too, and back converted to code on the new node. Pending scheduling blocks contain function code and hence can increase the snapshot size significantly. Therefore, migration (using the `moveto` operation) requests should not be embedded in a scheduling block.

8.3.5 Security by Capability-based Authorization and a lightweight Distributed Organization System Layer

In the simplest case *JAM* nodes are connected by peer-to-peer network links. But large-scale network environments like the Internet are organized in hierarchical graph-like structures with changing and transparent connectivity. To organize *JAM* nodes in such large-scale and heterogeneous networks, an additional Distributed Organization System (DOS) layer is required. Furthermore, large-scale networks introduce new issues in privacy, security, and trust, which must be addressed by the *DOS*.

The fundamental communication concept of the *DOS* - that is entirely implemented in *JS* (see [BOS16A] for details) - are Object-orientated Remote Procedure Calls (ORPC), already introduced in Section *7.8* in the *JAVM* platform context. They are initiated by a client process with a transaction operation, and serviced by a server process by a pair of get-request and put-reply operations, based on the *Amoeba* DOS [MUL90]. Transactions are encapsulated in messages and can be transferred between a network nodes. The server is specified by a unique port, and the object to be accessed by a private structure containing the object number (managed by the server), a right permission field specifying authorized operations on the object, and a second port protecting the rights field against manipulation (see [MUL90] for details) using one-way encryption with a private port. All parts are merged in a capability structure `[srvport]obj(rights)[protport]`. The rights field can only changed with the original secret protection port (otherwise *protport* is invalid).

Capabilities are also used in this work for agent-platform negotiation, with a server port designating a platform or platform group. Agent mobility (shown in Figure *8.3*) is performed by text-code transformations and by using a run server. A capability rights field can be used to determine the maximal privilege level of a migrated agent process.

The integration and network connectivity of client-side application programs like Web browsers as an active agent processing platform requires client-to-client communication capabilities, which is offered by a broker server that is visible on the Internet or Intranet domains, shown in Figure *8.3* (left). To provide compatibility with and among all existing browser, *node.js* server-side, and client-side applications, an RPC based inter-process communication encapsulated in *HTTP* messages exchanged with the broker server operating as a router is used. Client applications communicate with the broker by using the generic *HTTP* client protocol and the `GET` and `PUT` operations. RPC messages are encapsulated in *HTTP* requests. If there is an RPC server request passed to the broker, the broker will cache the request until another client-side host performs a matching transaction to this server port. The transaction is passed to the original RPC server host in the reply of an *HTTP* `GET` operation.

There is a Directory and Name Server (*DNS*) providing a mapping of names (strings) on capability sets, organized in directories. A directory is a capability-related object, too, and hence can be organized in graph structures. *DNS* server can be distributed and chained in graphs, too. A capability set binds multiple capabilities associated with the same semantic object, e.g., a file that is replicated on multiple file servers.

Each directory contains rows and a set of columns for each row with different restricted row capabilities enabling rights restriction and selection of objects and authorization key, e.g., used for agent role negotiation, privilege granting, code dictionary access. Column selection can base on the agent privilege level.

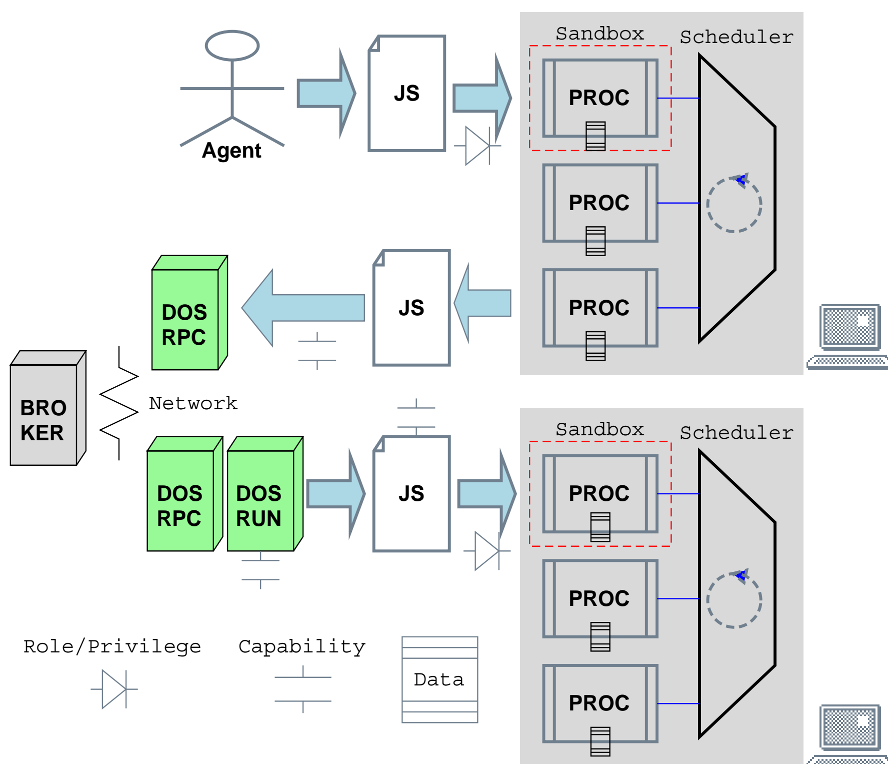

Fig. 8.3 *JAM agent mobility using the DOS with text-code and code-text transformations. Capabilities determine the agent process role and privilege level granted by the sandbox environment.*

8.4 JAM Implementations

The *JAM* platform can be deployed on a wide variety of host platforms and operating environments.

8.4.1 JAMLIB

JamLib is an embeddable *JAM* module that can be integrated in any host application. The *JamLib* code does not depend on special IO modules. It depends only on the core `fs` and `util` modules (command line version for *node.js/jxcore/JVM*) or a common WEB browser environment. *JamLib* is intended for integration in mobile application software. It provides a unified interface to *JAM*.

A *JAM* instance can be created to execute *AgentJS* agents. A JAM object instance provides a standard interaction interface that enables access of the

JAM by the host application. Agent execution can be controlled by using the `create`, `execute`, `migrate`, and `kill` methods. The tuple apace, which offers basic agent communication, can be accessed by the host application by the `inp`, `out`, `rd`, and `rm` methods. Callback and IO functions for tuple space access of *JAM*, agent creation, and agent migration, can be defined, shown in Example 8.2.

The *AIOS* can be extended with additional functions that can be accessed by *AgentJS* agents. The extend method adds a new host application function to a specific privilege level set. Functions added to *AIOS* provide agents the possibility to access variables and functions outside of the sandbox and the *AIOS*! An *AIOS* function may not return references to functions or external objects (variables)! This would violate the sandbox approach. Especially external function references would be unusable after an agent has migrated. Extension functions cannot access the agent object itself.

Agent mobility requires the transfer of agent code in textual *JSON+* format snapshots from one *JAM* node to another. A host application must provide a way to send a text message to a specified destination, and to pass text messages containing *JSON+* agent snapshots to the *JAM*. Received snapshots can be executed by simply calling the *JAM* method `migrate`. Received signals can be passed by the signal method to the *JAM* execution engine. In the other direction, the host application must provide a connection to the outside world. This is done by the `connections` option passed during the *JAM* instantiation.

Ex. 8.2 *Simple JamLib usage*

```
1  var Jam = require('pathto/jamlib');
2
3  Send agent data in JSON+ format to node specified by dest (string path)
4  function send (data,dest) {
5    Send data ..
6    return data.length;
7  }
8
9  Test if a destination is reachable.
10 function link (dest) {
11   return true;
12 }
13
14 Define a tuple provider function
15 function provider(pat) {
16   switch (pat.length) {
17     case 2:
18       switch (pat[0]) {
19         case 'SENSOR2':
```

```
20            return [pat[0],(256*rnd())|0];
21        }
22        break;
23    }
24  }
25
26  Define a tuple consumer function
27  function consumer(tuple) {
28    switch (tuple.length) {
29      case 3:
30        switch (tuple[0]) {
31          case 'ADC':
32            console.log('Host application got '+tuple);
33            return true;
34        }
35        break;
36    }
37    return false;
38  }
39
40  Create JAM instance
41  var myJam = Jam.Jam({
42    connections : {
43      path: {send:send,link:link}
44    },
45    consumer:consumer,
46    print:console.log,
47    provider:provider,
48    verbose:0,
49  });
50  myJam.init();
51  myJam.start();
52
53  Define a simple agent class template
54  function ac(arg1,arg2) {
55    this.x=arg1;
56    this.y=arg2;
57    this.act = {
58      init: function () {log('init '+this.x)},
59      comp: function () {this.x++;this.y--;log(this.x+','+this.y)},
60      wait: function () {sleep(1000);}
61    };
62    this.trans = {
63      init: function () {return comp},
64      comp: function () {return wait},
65      wait: function () {return comp}
66    };
```

```
67    this.next='init';
68  }
69
70  Create agent on this JAM instance (root node) directly using the
71    constructor function
72  var a1 = myJam.createAgent(ac,[1,2],1);
73
74  Or compile the agent class constructor and add it to the world library
75  myJam.compileClass(ac);
76  Create agent using the loaded agent class identified by the class name
77  var a2 = myJam.createAgent('ac',[1,2],1);
```

8.4.2 JAMSH: JAM Shell

The JAM shell provides an interpreter API for the JAM library. The supported shell commands that can be executed via the shell command line or from a script file are summarized in Definition 8.1.

Def. 8.1 *JAM shell commands*

```
Shell Commands

add({x,y})
  Add a new logical (virtual) node
connect({x,y},{x,y})
  Connect two logical nodes
connected(to:dir)
  Check connection between two nodes
compile(function)
  Compile an agent class constructor function
create(ac:string,args:*[]|{},level?:number,node?)
  Create an agent from class @ac with given arguments @args and @level
exit
  Exit shell
extend(level:number|number[],name:string,function,argn?:number|num-
ber[])
  Extend AIOS
http(ip,dir,index?)
  Create and start a HTTP file server
inp(pattern:[],all:boolean)
  Read and remove (a) tuple(s) from the tuple space
kill(id)
  Kill an agent
link(to:dir)
  Connect two phyiscal nodes
log(msg)
  Agent logger function
```

```
open(file:string)
  Open an agent class file
out(tuple:[])
  Store a tuple in the tuple space
port(dir,options,node)
  Create a new physical communication port
rd(pattern:[],all:boolean)
  Read (a) tuple(s) from the tuple space
rm(pattern:[],all:boolean)
  Remove (a) tuple(s) from the tuple space
script(file:string)
  Load and execute a jam shell script
setlog(<flag>,<on>)
  Enable/disable logging attributes
signal(to:aid,sig:string|number,arg?:*)
  Send a signal to specifid agent
start()
  start JAM
stats(kind:"process"|"node"|"vm")
  Return statistics
stop()
  stop JAM
ts(pattern:[],callback:function(tuple)->tuple)
  Update a tuple in the space (atomic action) - non-blocking
time()
  print AIOS time
unlink(to:dir)
  Disconnect remote endpoint
verbose(level:number)
  Set verbosity level
```

An example script is shown below in Example 8.3. The script defines an agent class constructor function *fib* that is immediately compiled and analysed. Finally, the *JAM* scheduler loop is started and an agent is instantiated from the previously installed agent class.

Ex. 8.3 *JAM shell script example*

```
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(fib)
start()

on('link+',function () {
  create('fib', {val: [10, 5]})
});
port(DIR.IP('134.102.219.1:10001'));
link(DIR.IP('10.102.1.2:10001'));
```

8.4.3 JAMAPP: JAM Application Program

The *JAM* App provides a GUI for the *JAM* library and is available for node.js (jxcore) and WEB browser. The *JAM* App has a multi-page view and provides configuration (setup), network control, *JAM* control and diagnostics, agent control and information, and message logging.

The WEB browser version is a mixed HTML/CS/JS application and includes JAM (*JAM* library) accessed by only one HTML page. The entire WEB package can be loaded from a http server running inside a JAM shell. The size of the entire WEB package is only about 1MB ensuring fast loading and includes the entire GUI framework.

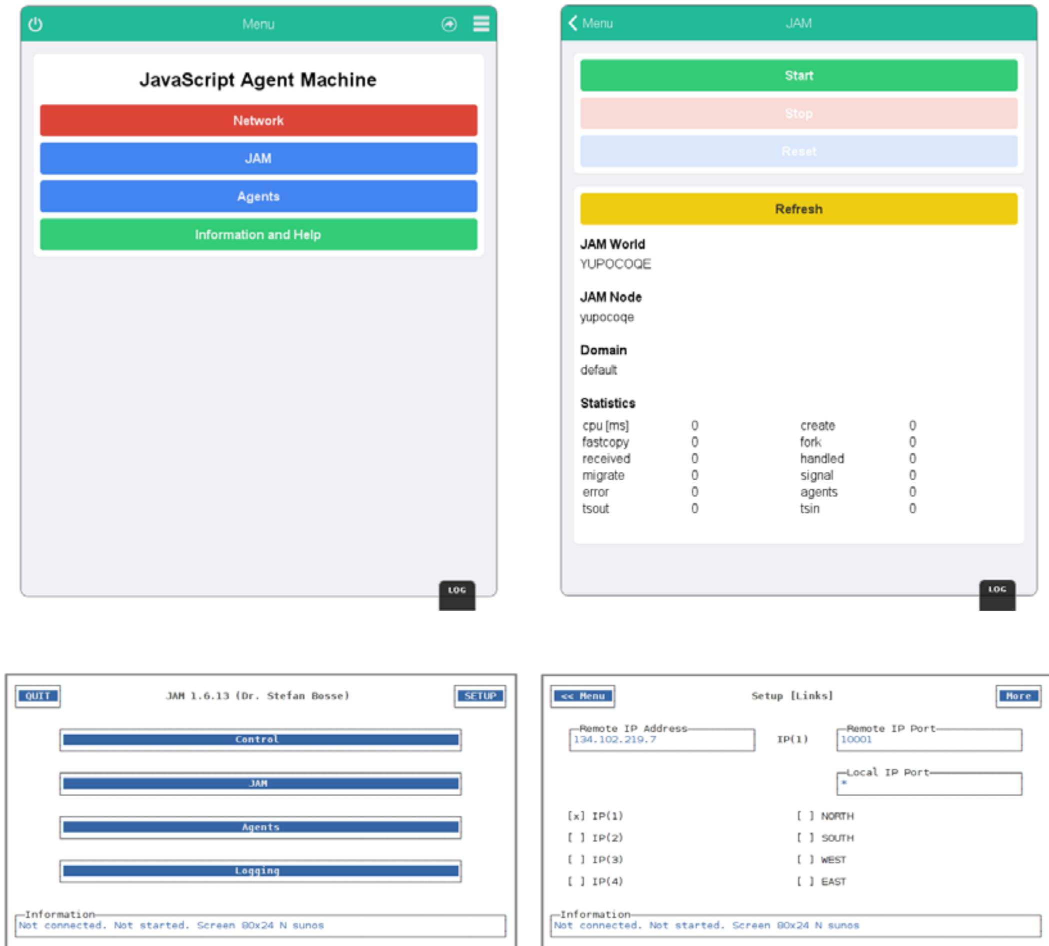

Fig. 8.4 *(Top) Browser version of the JAMapp with multi-page navigation (Bottom) Terminal version of the JAMapp*

8.4.4 JS Execution Platforms

Originally, *JAM* was executed on *node.js* or by a WEB browser *JS* engine. On some host platforms the more portable *jxcore* VM was used. Basically, *node.js* and *jxcore* add an event-processing and IO layer on the top of a core *JavaScript* engine providing asynchronous (callback-based) IO and event processing. They use Googles *V8 JavaScript* engine offering just-in-time (JIT) native code compilation. But both *V8*-based execution platforms require a substantial amount of memory, even on start-up (about 30-50M), see the following performance evaluation section. On low-resource platforms, used, e.g., in the IoT with less than 128MB RAM, the *V8* engine with its JIT approach cannot be

used. To deploy *JAM* on low-resource platforms, the *JVM* execution engine was developed. *JVM* is based on Samsungs *jerryscript* engine and the *IoT.js* project. *JVM* is a byte code engine that compiles JS directly to byte code from a parsed AST. This byte code can be stored in a file and loaded at run-time. *JVM* is well suited for embedded and mobile systems, e.g., the Raspberry PI Zero equipped with an ARM processor. *JVM* has approximately 10 times lower memory requirement and start-up time compared with *nodes.js*.

8.4.5 JAM Connectivity

JAM agents are mobile, i.e., a snapshot of an agent process containing the entire data and control state including the program specifying the agent behaviour, can migrate to another *JAM* platform. *JAM* provides a broad range of connectivity, shown in Figure 8.5, available on a broad range of host platforms. There is Peer-to-Peer (P2P) connectivity between neighbour nodes by using, e.g., serial links used in mesh-grid networks, and wide-area connectivity, i.e., Internet, by using the Distributed Organization System layer (DOS) and a broker server.

In P2P connectivity, *JAM* nodes communicate via the Agent Management Port (AMP). *AMP* provides messaging between *JAM* nodes for agent migration, signal and tuple migration, and agent control. *AMP* messages can be transferred via any stream-like link. Additionally, an external monitor program (debugger) can connect to a *JAM* node via *AMP*.

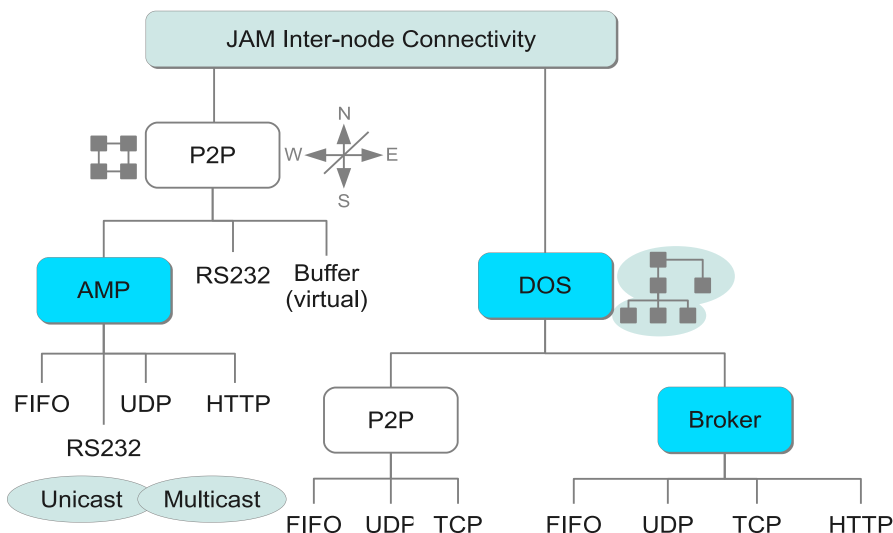

Fig. 8.5 *JAM Connectivity (P2P: Peer-to-Peer, AMP: Agent Management Port, DOS: Distributed Organization Layer)*

On the Internet IP-based protocols are commonly used to provide *AMP* message passing between *JAM* nodes using *UDP*, *TCP*, or *HTTP* protocols. One common issue are private or virtual networks with Network Address Traversal (NAT). To establish *UDP* communication between NAT networks, an external public rendezvous broker providing *UDP* hole punching techniques can be used. In this case, *JAM* nodes register on the broker with their node name, and other *JAM* nodes can connect to the (IP hidden) *JAM* nodes by their node names. The broker supports domain services (partitioning of nodes in domains/groups, e.g., based on GPS data).

8.5 Performance Evaluation

In [BOS16A] and [BOS17A], the performance of *JAM* was evaluated on different host platforms and using different JS VMs.

The *JAM* platform was evaluated with different benchmark tests executed on different host platforms (A & B), in terms of Clouds low-resource and in terms of IoT mid-resource systems. Please note that the measurement results depend on the *JS* VM garbage collector algorithm and activity at run-time.

The following host platforms were used in the following performance evaluation, and *one physical JAM is usually composed of a JAM world consisting of four logical (virtual) nodes, connected in a grid (ring) with virtual circuit links (queues)*:

Test host platform A
Embedded System, Intel(R) Celeron(R) CPU 743 @ 1.30GHz, 2GB DRAM, node.js v0.10.36, Sun Solaris-11 OS,

Test host platform B
Smartphone, Toughshield R500+, 1GB DRAM, Android 4.1.2, quad-core Arm Cortex A5, ARMv7-A, 1.2GHz, jxcore v.0.10.40

Test host platform C
Hewlett-Packard HP xw9400 Workstation, AMD Opteron 2216 2.4GHz x64, node.js and JVM

Test host platform D
Lenovo, ideapad, A-10, quad-core Rockchip A9 processor running at 1.6GHz, node.js and JVM

Test host platform E
Raspberry Pi Zero, Broadcom 1GHz ARM11, node.js

The creation (instantiation) or forking of new agents always involves a code-to-text and text-to-code transformation using the sandbox environment. The performance of this operation is shown in Table 8.1 and 8.2, for a small and a complex agent class performed on a PC (A) and mobile computer (B). Below

1000 agents/physical *JAM* an agent creation requires about 1-5ms, and the memory overhead is reasonable small. Migration requires the same code transformation, resulting in similar results, shown in Table 8.3. The ARMv7 host platform under-performs significantly compared with the x86 platform. This has two reasons: The ARMv7 processor has smaller code/data caches (L1:32 vs. 64kB, L2:512kB vs. 1MB), and the *node.js/jxcore* VM is optimized for x86 architectures. Table 8.4 shows the agent context switch performance of the *JAM* scheduler, which is very fast. Again. the ARMv7 platform under-performs, but is still fast enough for mobile devices. Tuple space I/O adds only a small overhead, as shown in Table 8.5. Finally, Table 8.6 poses the minimal memory requirement for a *JAM* node with a typical agent population. Commonly, less than 32MB is required, confirming the suitability of *JAM* for low-resource embedded systems.

The processing performance of *JAM* depends on the host platform (computer, server, smart phone, embedded system) and the used JS engine (node.js/V8, *jxcore*/V8/*Spidermonkey*, *JVM*).

Creation of Agents	*Time/Agent*	*+Memory/Agent*
100	A:0.7ms, B: 1.7ms	A:2.0kB, B: 17.9kB
1000	A:0.7ms, B: 2.4ms	A:2.9kB, B: 21.7kB
10000	A:23ms, B: 10.6ms	A:18.2kB, B: 17.5kB

Tab. 8.1 *Test Case 1: Agent creation on a logical (virtual) node, simple agent (text code size 0.9kB, five activities each with two statements, two variables and two parameters), memory: VM overhead/agent (heap+stack)*

Creation of Agents	*Time/Agent*	*+Memory/Agent*
100	A:1.6ms, B:4.5ms	A:11.5kB, B:131kB
1000	A:1.6ms, B: 5.1ms	A:91kB, B: 83.8kB
10000	A:3.1ms, B:-	A:80.8kB, B: -

Tab. 8.2 *Test Case 2: Agent creation on one logical (virtual) node, complex learner agent (with agent text code size 10.4kB, two sub-classes: explorer, voter, total 20 activities, each with about 10 statements, 33 variables, and two parameters), memory: VM overhead/agent (heap+stack)*

epubli, ISBN 9783746752228 (2018)

Initial Agents / logical node (total)	*Migrations/ Agent (total)*	*Migration+ Execution Time/Agent*	*+Memory/Physical node*
1 (4)	1000 (4000)	A:1.3ms, B: 4ms	A:28MB, B:16MB
10 (40)	1000 (40000)	A:1.0ms, B: 3.7ms	A:26MB, B:17MB
100 (400)	1000 (400000)	A:1.1ms, B: 3.3ms	A:70MB, B:28MB

Tab. 8.3 *Test Case 3: Agent migration from one logical (virtual) node to a neighbour node, physical node with four logical nodes connected in a ring, explorer agent (with agent text code size 4.3kB), n agents on each logical node, N circular migrations in the ring network two activity executions/ agent/migration, memory: VM overhead/agent (heap+stack)*

Agents / logical node (total)	*Scheduled Activities/Agent (total)*	*Scheduling + Execution Time/Agent*	*+Memory/Physical node*
1 (4)	20000 (80000)	A:16μs, B:67μs	A:5MB, B: 7 MB
10 (40)	20000 (800000)	A;8μs, B:33μs	A:6MB, B: 9MB
100 (400)	20000 (8000000)	A:8μs, B:29μs	A:20MB, B: 9MB

Tab. 8.4 *Test case 4: Agent scheduling on four logical (virtual) nodes, simple agent (text code size 1kB, five activities each with one statement, two variables, and two parameters), memory: VM overhead/physical node (heap+stack)*

Agents / logical node (total)	*Scheduled Activities/ Agent (total)*	*Scheduling + IO Execution Time/ Agent*	*+Memory/Physical node*
1 (4)	2000 (8000)	A:31μs, B:151μs	A:4MB, B: 7MB
10 (40)	2000 (80000)	A28μs, B:119 μs	A:6MB, B: 7MB
100 (400)	2000 (800000)	A:64μs, B: 217 μs	A:26MB, B: 16MB

Tab. 8.5 *Test case 5: Agent scheduling on four logical (virtual) nodes, simple agent with Tuple Space I/O (pairwise "out/in" in different activities) (text code size 1kB, five activities each with one statement, two variables, and two parameters), memory: VM overhead/physical node (heap+stack)*

Agents / physical node	*VM Memory / physical node*
1	A:23.2MB, B: 25 MB
10	A:23.7MB, B:25 MB
100	A:31.1MB, B: 38 MB
1000	A:104MB, B: 107 MB

Tab. 8.6 *Test case 6: Lowest memory requirements in minimal JAM configuration (1 world, 1 node), agent creation on one logical (virtual) node, complex machine learner agent (text code size 10.4kB, LZ compressed size 2.1kB, two sub-classes: explorer, voter, total 20 activities, each with about 10 statements, 33 variables, and two parameters), with VM parameter --max-new-space-size=1024, total VM memory=heap+stack, after agent creation.*

Commonly a *JS* VM compiles *JS* source text to an intermediate Abstract-Syntax-Tree (AST) representation. Simple VMs interpret this AST, enhanced VMs compile the AST to native or virtual machine code for improved execution performance.

JVM is a Byte-code interpreter compiling *JS* text code to Byte-code at run-time directly from parsed AST, and V8-based machines are hybrid interpreters combining Byte-code execution and just-in-time (JIT) native code generation. Byte-code engines have the advantage over native code engines to have a high degree of portability, but the disadvantage of slower execution speed (~100 times). Native code engines have a much higher memory consumption, shown in the experimental evaluation and comparison in Table 8.7.

In all benchmark experiments in Table 8.7 the *JAMLIB* code library was used. The creation and migration of agents require text-to-code and code-to-text transformations that are computational expensive (see A1 benchmark).

Though the computation speed of *JVM* is about 100 times slower compared with V8 engines, the agent activity computation performance is only 3-4 times slower (see FFT benchmark F1).

This is a result of a watchdog timer built in the *JVM* byte code interpreter required by *JAM* for agent scheduling and time slicing.

The watchdog raises an exception if an agent activity exceeds the (negotiated) time slice. In V8 engines this watchdog approach cannot be implemented (due to the native code compilation), and must be emulated by, e.g., check pointing, which injects checkpoint function calls in the agent code (in all functions and loops), slowing down the agent activity execution significantly. The watchdog extension is discussed in the next section.

JS VM/Benchmark	*Host C*	*Host D*	*Host B*	*Host E*
JVM, A1, N4, n=100 Creation/Migration	1.6/9.7ms 3/4MB	4.8/21.6ms 2/3MB	8.4/49ms 2/3MB	-
ND1, A1, N4, n=100 Creation/Migration	0.2/1.1ms 27/36MB	-	-	-
ND2, A1, N4, n=100 Creation/Migration	0.13/1ms 21/32MB	-	-	2.2/15ms 15/27MB
JVM, F1, D2000, n=10 Computation,	1300ms 6MB	3000ms 5MB	4200ms 5MB	
ND1, F1, D2000, n=10 Computation	400ms 45MB	-	-	-
ND2, F1, D2000, n=10 Computation	550ms 35MB	-	-	300ms 40MB

Tab. 8.7 *Benchmarks comparing different JS VMs. All times per agent and action, all memory values are total JAM resident memory after the benchmark operation, A1: Simple Explorer Agent, F1: FFT Computation Agent, ND1:node.js v5.11, ND2: node.js v0.10, n: Number of agents, N: Number of logical nodes, D: Size of data vector*

8.5.1 Watchdog Control and Time Slicing

There are two bottlenecks in the *JS* VM related to agent process control (including creation, forking, migration of agents) and agent process execution: (1) Efficient text-code and code-text transformation (2) Time slice control of agent activity execution using a watchdog. The fall-back solution of time slice control working on all platforms is the injection of checkpoint function in the agent code prior execution (used in the previously shown evaluation). The checkpoint function reduces execution performance and slows down text-code transformation. An advanced method is the modification of the *JS* VM with built-in watchdog control. Such a watchdog and more *AIOS* related enhancements were added to the *jxcore* and *jerryscript* engines leading to *AIOS*-optimized *jxcore+* and *jvm* engines (based on V8.3.28 with node.js 0.10.40 and *jerryscript* 1.0 with iot.js, respectively)

There are two different approaches compared in Figure 8.6.

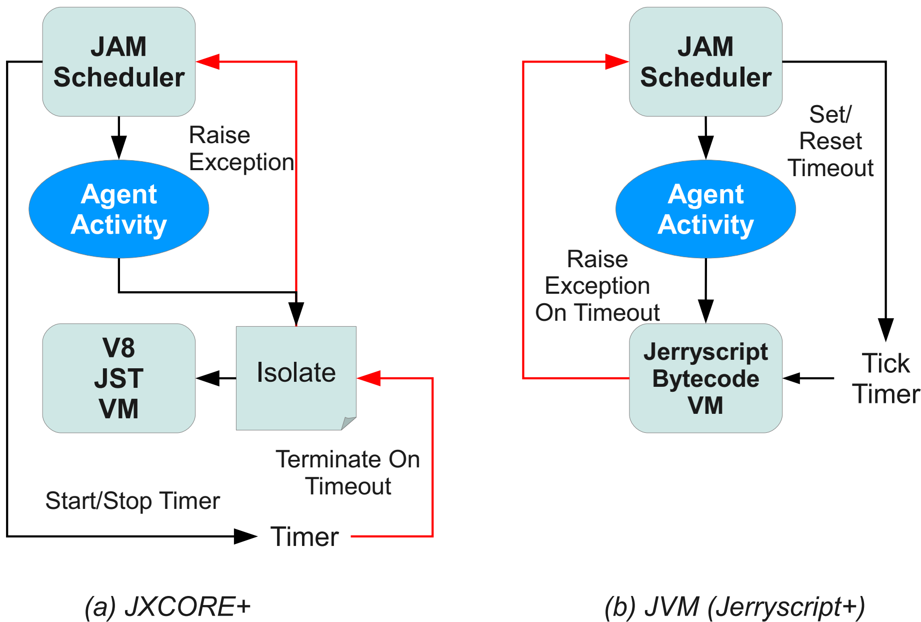

Fig. 8.6 *Two different approaches extended common JS VMs with watchdog control: (a) V8 Isolate used in jxcore+ (b) Watchdog control embedded in Bytecode VM used in jvm*

V8 Isolate

This watchdog implementation utilizes the V8 isolated context instance for the execution of the AIOS. An isolated instance enables the protected execution of agent code with pre-emptive termination of the code, finally throwing JS exceptions directed to the AIOS. The termination of the agent processing is performed by a different timer thread or a native timer. An agent activity is executed by the JAM scheduler in an isolated container. The watchdog is implemented with a timer started on each agent activity processing and that terminates the agent process if a time-out occurs (time-slice, typically 20ms). The agent activity termination is transformed in a JS exception that is handled by the JAM scheduler.

Embedded Bytecode Interpreter

The time-out control is embedded in the Bytecode interpreter loop. On each Bytecode command execution the process time-out is checked. If a time-out event occurs, the Bytecode interpreter raises a JS exception directly that is handled by the JAM scheduler.

Different benchmarks were performed with *jamlib* version 1.19.3 comparing different JS VMs with and without watchdog control, shown in Table 8.8.

Benchmark	jxcore+ 1.3.3X Watchdog	node.js 5.9.0 Code CP	jvm Watchdog	node.js 0.10.40 Code CP
creation N=1000	- µs - MB	- µs - MB	- µs - MB	-
scheduling N=100, M=1000	4.6 µs 37 MB	2.8 µs 38 MB	- µs - MB	2.0 µs 26 MB
migration N=100, M=100	560 µs 150 MB	750 µs 63 MB	-	650 µs 62 MB
computation (fft) N=100, M=10	26 ms 110 MB	153 ms 73 MB	-	120 ms 58 MB

Tab. 8.8 *Benchmarks comparing the impact of a watchdog versa code check pointing on performance. All times per agent and action, all memory values are total JAM resident memory after the benchmark operation; Host: i5-4310, 2GHZ, 6GB RAM, Solaris 11.3; N: Number of agents, M: Number of iterations*

8.6 SEJAM: The JavaScript Agent Simulator

There is a simulation environment build on the top of *JAM*, *SEJAM*, discussed in Section 11.7. The simulation environment adds a GUI with visualization capabilities to *JAM*. One of the key features of *SEJAM* is the capability to integrate the simulator in real world *JAM* networks (similar to hardware-in-the-loop simulation).

Additionally, *SEJAM* is a multi-domain simulator that can be used to combine MAS and physics simulation in one monolithic program.

The coupled physical and computational simulation consists of two engines:

- Physics: Multi-body physics solver using a mesh-grid network of nodes connected by damped springs that can be parametrized (*Canon*); and
- Computation: Mobile Multi-Agent Systems and Networks of JavaScript Agent Machines processing agents.

The simulator features a fully JavaScript based modelling and programming environment, and *SEJAM2* is programmed entirely in JavaScript, too! Agents are deployed in ICT networks:

- Each node of the network is coupled to sensors and actuators
- Agents can access sensors and control actuators
- Sensors and actuators are modelled with multi-body physical systems
- Direct interaction between agents and physical system and vice versa

The physical model can be accessed by all agents, shown in Fig *8.7*. A MAS simulation consists of node and worker agents. There is one artificial agent (world agent) representing the world and manages the simulation, i.e., generating and updating sensor data accessed by the node and worker agents by using the unified tuple-space interface. The world agent can read and modify physical simulation variables, i.e., reading strain, force parameters, and setting material/structure properties (stiffness).

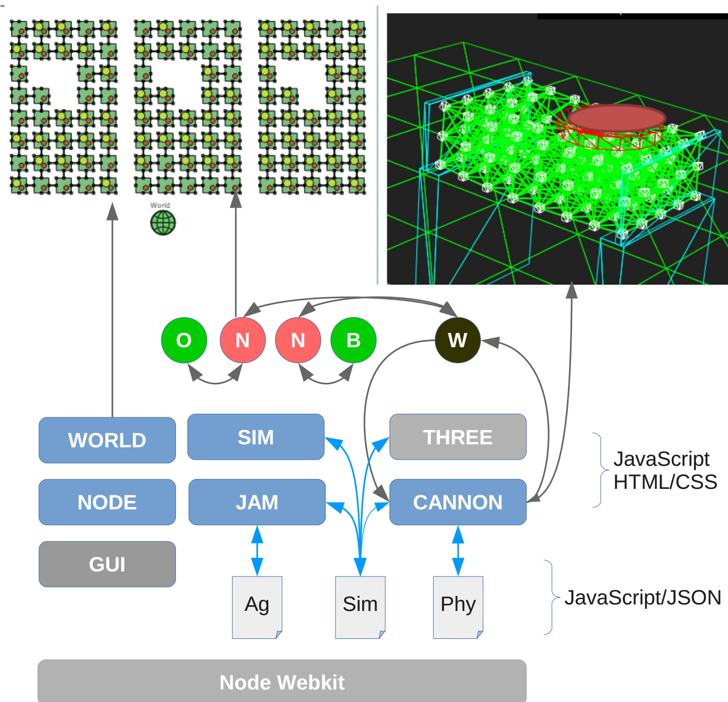

Fig. 8.7 *SEJAM2 Simulation Environment (Left) MAS (Right) Physics (Top) Agents, N: Node, R: Mobile Worker, W: World (Middle) Model, Ag: AgentJS, Sim: Simulation, Phy: Cannon Model*

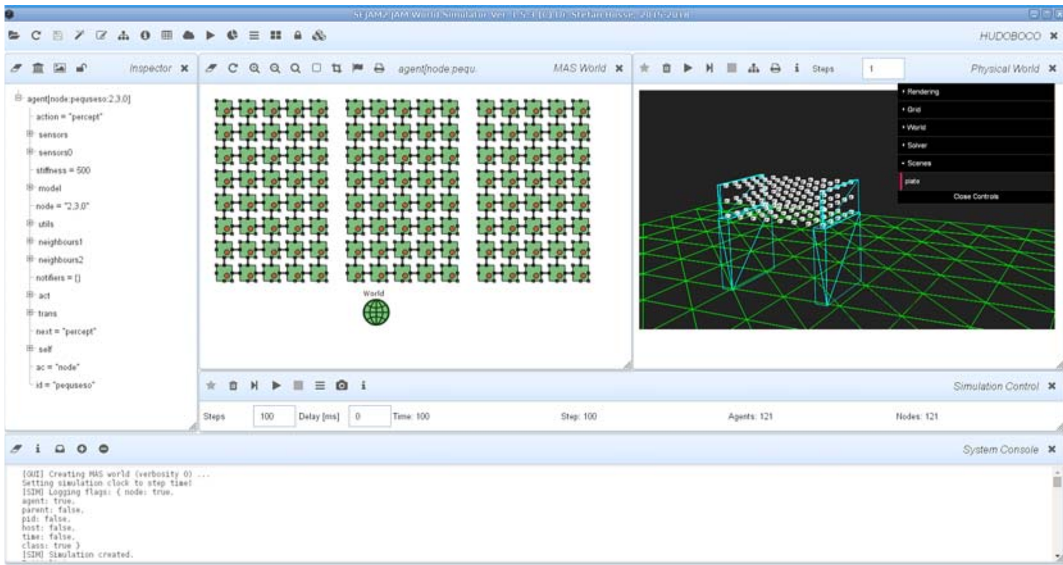

Fig. 8.8 SEJAM2 Simulator with a combined MAS (center) and MBP simulation (right) based on JST/JS/JSON modelling.

A snapshot of a simulation run with a plate consisting of a 8x5x3 network of computer and mass nodes connected by springs is shown in Figure 8.8. The objective of the complex simulation is to investigate the interaction of computational systems (agents) with sensors, actuators, and physical systems.

8.7 Heterogeneous Environments

JAM can be already deployed in heterogeneous environments composed of devices ranging from embedded computers up to servers. But *JAM* requires a JS VM that requires a substantial amount of memory and computational power. Very low-resource host platforms like single microchip computers with a size about 1mm^2 are unsuitable. The *PAVM* platform meets the requirements of very low-resource hosts. Although both platforms support agents with nearly the same behaviour and operational model they are not compatible on code level.

To enable the composition of future large-scale applications ranging from very-low-resource nodes (1mm^2 computers integrated in materials, smart materials and structures) to high-resource devices (generic computers, servers, mobile and embedded devices), both platform architectures must co-exist supporting agent migration seamlessly. This co-existence demands a compatibility layer by introducing an on-the-fly cross-compiler enabling agent mobility between different platform technologies seamlessly, shown in Figure

8.9. This just-in-time cross-compiler translates agents to *AgentFORTH* object code to *AgentJS* text code agents and vice versa by preserving the entire agent state. This compiler is optimally contained in *JAM* as a service. AgentJS is a dynamic language with dynamic storage management, whereas *AgentFORTH* is a static language with storage allocated on agent creation.

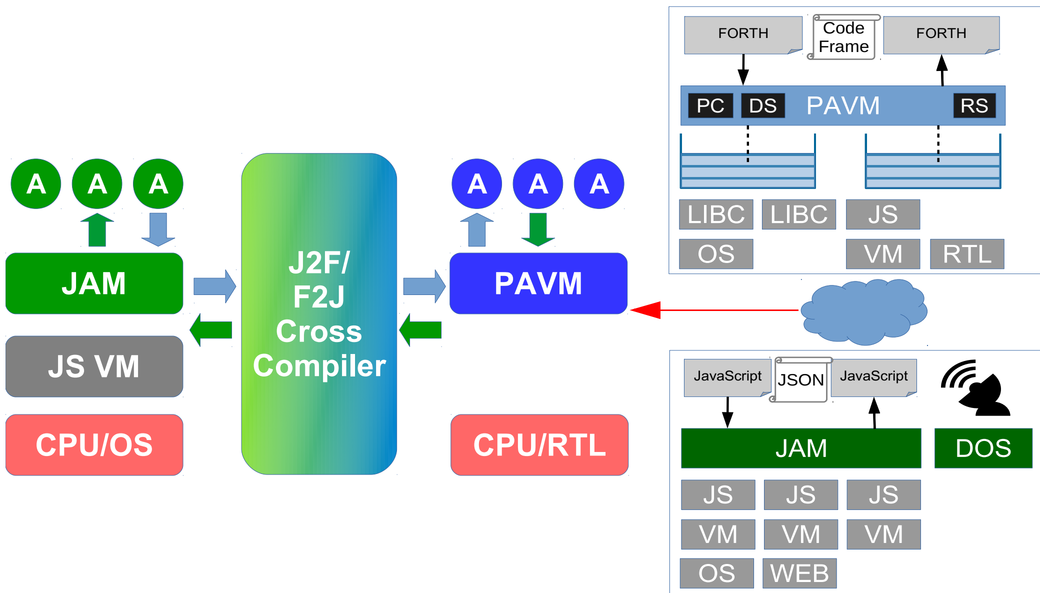

Fig. 8.9 *Dual-machine approach coupling JavaScript JAM and FORTH code PAVM by using a Just-in-Time agent code cross-compiler (J2F: JavaScript-to-FORTH)*

The F2J direction is always possible, but the opposite J2F direction is currently not fully defined and requires constrained JS (with pseudo-static memory allocation).

8.8 Further Reading

1. A. Aravinth, *Beginning Functional JavaScript - Functional Programming with JavaScript Using EcmaScript 6*. Apress, ISBN 97814842-26551, 2017. (Note: JAM only supports EcmaScript 5)
2. H. Mehnert, J. Ohlig, and S. Schirmer, *Funktional programmieren lernen mit JavaScript*. O'REILLY, 2013.
3. H. Lin, *Architectural Design of Multi-Agent Systems*. IGI Global, 2007, ISBN 9781599041087

4. U. Hölzle, *Adaptive Optimization for Self: Reconciling High Performance with Exploratory Programming*, Stanford University, 1994.

epubli, ISBN 9783746752228 (2018)