A. Agent Input-Ouput System (AIOS)

The Agent Input-Output System (*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*. A *JAM* instance consists of multiple modules:

- Node
- World
- Code/Process (control and modification)
- Tuple space
- Signals
- Mobility
- Network and Communication (AMP)
- Scheduler
- Security
- Watchdog
- Artificial Intelligence (optional):
  - Machine Learning (ML)
  - Constraint Solving Programming (CSP)
  - Logic Programming and Satisfiability Solver (SAT)

The modules are accessible (directly or indirectly) by the agents via the AIOS, shown in Fig. 1.

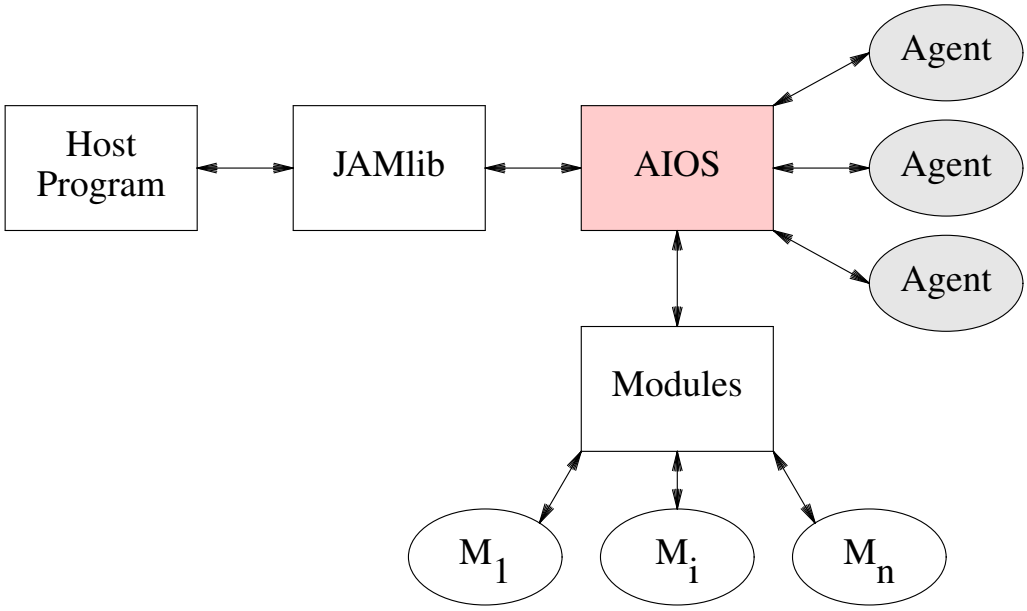

Figure 1. Interface between agents and JAM and between JAM and a host application by the Agent Input-Output System (AIOS).

A.1 Agent Roles

Security is provided by the agent platform by assigning execution roles and levels to agents. The roles are dynamic and can be changed at run-time. The execution of agents and the access of resources is controlled by those roles to limit Denial-of-Service attacks, agent masquerading, spying, or other abuse:

There are four levels:

1. Guest (not trustful, semi-mobile)

2. Normal (maybe trustful, mobile)

3. Privileged (trustful, mobile)

4. System (highly trustful, locally only, non-mobile)

The lowest level (0) does not allow tuple space access, agent replication, migration, or the creation of new agents. The JAM platform decides the security level for new received agents. An agent cannot create agents with a higher security level than its own. The highest level (3) has an extended AIOS 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 (defined by the services, e.g., agent role service or other resources like tuple space access). The system level can not be negotiated. The capability is node ans service 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 using a valid capability with the appropriate rights:

- Scheduling time (longest slice time for one activity execution, default value is between 20-200ms)
- Run time (accumulated agent execution time, default is 2s)
- Living time (overall time an agent can exist on a node before it is removed, default is 200s)
- Tuple space access limits (data size, number of tuples)
- Memory limits (fuzzy, usually the entire size of the agent code including private data, actually not limited)
- Network links and connectivity (supported by the AMP module)

A.2 Agent Scheduling

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 *JAM* platform. Agent 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 checkpoint functions `CP()` at the beginning of a body of each function contained in the agent code,

and by injecting the `CP` function calls in loop expressions. Although 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 agent process 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 processes 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 (or just applying an unconditional value). If the activity is blocked (agent is suspended, except signal handling), the next transition is computed after the resume of the agent process.

In contrast to the *AAPL* model that supports agent process blocking on statement level, eventually allowing multiple blocking statements (e.g., IO/tuple-space access) inside activities, JS is not capable of handling any kind of process blocking of user instructions (there is no process and blocking concept). For this reason, an activity may only contain one blocking statement, and the blocking is applied to the entire activity after the control flow of an activity function terminates.

Multiple blocking statements require scheduling blocks that can be used in *AgentJS* activity functions (at the end) handled by the *AIOS* scheduler. Blocking *AgentJS* functions with a pending result use common callback functions to pass function results to the agent, 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 references always 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 stops the agent process execution until an event occurred.

The basic agent scheduling algorithm is shown in the following algorithm 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.

```
∀ node ∈ world.nodes do
 ∀ process ∈ node.processes do
  Determine what to do with prioritized conditions:
    Order of operation selection:
    0. Process (internal) block scheduling [block]
    1. Resource exception handling
    2. Signal handling [signals]
       - Signals handled if process priority<HIGH
       - Signal handling increase process priority
         temporarily to allow low-latency activity
         and transition function scheduling!
    3. Transition execution
    4. Agent schedule block execution [schedule]
    5. Next activity execution
       - Lowers process priority
  if process.blocked or process.dead or
   process.suspended and process.block=[] and
   process.signals=[] or
```

```
 process.agent.next=none and process.signals=[] and
 process.schedule=[]
  then do nothing
elseif not process.blocked and process.block≠[]
  then execute next block function
elseif agent resources check failed
  then raise EOL exception
elseif process.priority < HIGH and process.signals≠[]
  then handle next signal, increase process.priority
elseif not process.suspended and process.transition
  then get next transition
  or execute next transition handler function
elseif not process.suspended and process.schedule≠[]
  then execute next agent schedule block function
elseif not process.suspended
  then execute next agent activity and
    compute next transition,
    decrease process.priority
```

Algorithm 1. JAM Agent Scheduler