Mobile Multi-Agent Systems for the Internet-of-Things and Clouds using the JavaScript Agent Machine Platform and Machine Learning as a Service

Stefan Bosse

University of Bremen, Department of Mathematics & Computer Science,
ISIS Sensorial Materials Scientific Centre, Germany

Abstract The Internet-of-Things (IoT) gets real in today's life and is becoming part of pervasive and ubiquitous computing networks offering distributed and transparent services. A unified and common data processing and communication methodology is required to merge the IoT, sensor networks, and Cloud-based environments seamless, which can be fulfilled by the mobile agent-based computing paradigm, discussed in this work. Currently, portability, resource constraints, security, and scalability of Agent Processing Platforms (APP) are essential issues for the deployment of Multi-agent Systems (MAS) in strong heterogeneous networks including the Internet, addressed in this work. To simplify the development and deployment of MAS it would be desirable to implement agents directly in JavaScript, which is a well known and public widespread used programming language, and JS VMs are available on all host platforms including WEB browsers. The novel proposed JS Agent Machine (JAM) is capable to execute AgentJS agents in a sandbox environment with full run-time protection and Machine learning as a service. Agents can migrate between different JAM nodes seamless preserving their data and control state by using a on-the-fly code-to-text transformation in an extended JSON+ format. A Distributed Organization System (DOS) layer provides JAM node connectivity and security in the Internet, completed by a Directory-Name Service offering an organizational graph structure. Agent authorization and platform security is ensured with capability-based access and different agent privilege levels.

Keywords: Agents, IoT, Cloud Computing, Agent Platforms, Learning

I. Introduction

The Internet-of-Things gets more and more real in today's life and is becoming part of pervasive and ubiquitous computing networks offering distributed and transparent services. Agents are already deployed successfully in production and manufacturing processes **[1]**, and newer trends poses the suitability of distributed agent-based systems for the control of manufacturing processes **[2]**, facing manufacturing, maintenance, evolvable assembly systems, quality control, and energy management aspects, finally introducing the paradigm of industrial agents meeting the requirements of modern industrial applications by integrating sensor networks. Multi-agent systems can be already successfully deployed in sensing applications, e.g., structural monitoring **[3]**.

In **[4]** the agent-based architecture considers sensors as devices used by an upper layer of controller agents. Agents are organized according to roles related to the different aspects to integrate, mainly sensor management, communication and data processing. This organization isolates largely and decouples the data management from changing networks, while encouraging reuse of solutions.

Distributed data mining and Map-Reduce algorithms are well suited for self-organizing MAS. Cloud-based computing with MAS, e.g., as a base for cloud-based design and manufacturing, means the virtualization of resources, i.e., storage, processing platforms, sensing data or generic information **[5]**. Mobile Agents reflect a mobile service architecture. Commonly, distributed perceptive systems are composed of sensing, aggregation, and application layers, shown in Fig. **1**. But IoT and Cloud environments differ significantly in terms of resources: The IoT consists of a large number of low-resource devices interacting with the real world and having strictly limited storage capacities and computing power, and the Cloud consists of large-scale computers with arbitrary and extensible computing power and storage capacities in a basically virtual world. A unified and common data processing and communication methodology is required to merge the IoT with Cloud environments seamless, fulfilled by the mobile agent-based computing paradigm, discussed in this work.

The scalability of complex industrial applications using such large-scale cloud-based and wide area distributed networks deals with systems deploying thousands up to million agents. But the majority of current laboratory prototypes of MAS deal with less than 1000 agents **[2]**. Currently, many traditional processing platforms cannot yet handle a big number of agents with the robustness and efficiency required by the industry **[2]** and Cloud applications. In the past decade the capabilities and the scalability of agent-based systems have increased substantially, especially addressing efficient processing of mobile agents. The integration of perceptive and mobile devices in the Internet raises communication and operational barriers, which must be overcome by a unified agent processing architecture and framework, discussed in this work.

In this work the behaviour of mobile agents are modeled with dynamic Activity-Transition Graphs (ATG), which are directly implemented in JavaScript (*JS*) program code holding the entire control and data state of an agent. The agent model bases on the mobile processes model introduced by Milner **[6]** several decades ago. The code can be modified by the agent itself using code morphing techniques (directly supported by *JavaScript* Just-in-time Compiler VM platforms), and that is capable to migrate in the network between nodes. This approach requires only a minimal Agent Processing Platform Service (APPS) and a RPC-based Distributed Organization System (DOS) layer for the deployment in the Internet domain, entirely implemented in *JS*, too. The *AgentJS* code can be directly executed by the underlying *JS* VM, in contrast to earlier work where special *AgentFORTH* code was used and executed on a dedicated stack-based VM (implemented itself in *JS*, too) **[7]**.

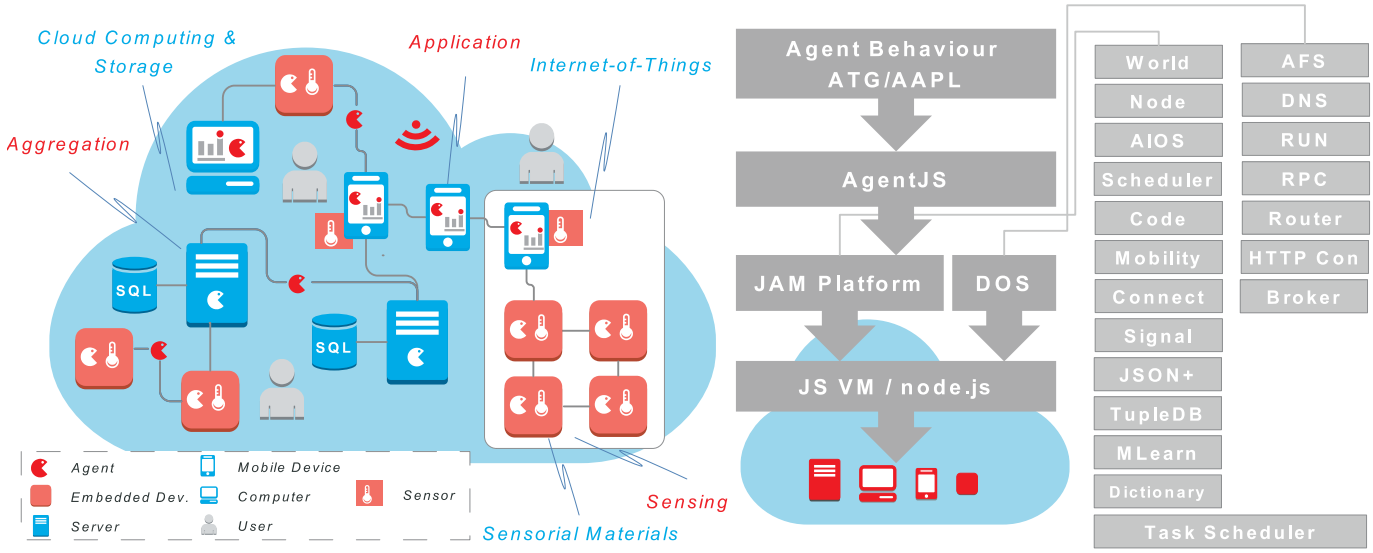

Fig. 1. Unified IoT - Cloud Distributed Perception and Information Processing with mobile agents and a JavaScript (JS) Agent Machine Platform (JAM) and Programming model AgentJS. An optional Distributed Organization System (DOS) layer **[7]** adds connectivity and security to JAM in the Internet domain.

The Bigraphical model proposed by Milner models the entire "computing" environment with place and link graphs, composing finally bigraphs **[8]**, reflected by the DOS layer and a Directory-Name mapping service (DNS) representing graphs.

Agents processed on one particular node can interact and synchronize by using a tuple-space, which were proposed in **[9]** and **[10]** as a suitable MAS interaction and coordination paradigm.

Privileged agents can store and look-up functions (e.g., learning algorithms) in a code dictionary. Remote interaction is provided by signals carrying data which can cross sensor node boundaries. The minimal APPS provides these interaction services among agent execution, mobility, and Machine Learning services accessed through the platform API. This approach provides a high degree of computational independency from the underlying platform and other agents, and enhanced robustness of the entire heterogeneous environment in the presence of node, sensor, link, data processing, and communication failures.

A sensor network as part of the IoT is composed of nodes capable of sensor processing and communication. Smart systems are composed of more complex networks (and networks of networks) differing significantly in computational power and available resources, rising inter-communication barriers.

They provide higher level information processing that maps the raw sensor data to condensed information. They can provide, e.g., Internet connectivity of perceptive systems (body area networks...). These smart systems unite sensing, aggregation, and application layers **[11]**, offering a more unified design approach and more generic and unified architectures. Smart systems glue software and hardware components to an extended operational unit, the basic cell of the IoT.

The central approach in this work focuses on mobile agents and the ability to support mobile reconfigurable code embedding the agent behaviour, the agent data, the agent configuration, and the current agent control state, finally encapsulated in portable *JavaScript* code. The mobility is granted by converting the agent program in a textual *JSON+* representation, and finally by parsing this text and executing the code again. This agent-specific mobile program code can be executed on a variety of different host platforms including mobile devices, embedded devices, sensor nodes, and servers, using *JAM* and a *JS* VM, closing the gap between the IoT and Cloud infrastructures.

Among the Internet-of-Things and Ubiquitous Sensorial Perception, one major field of application for the agent-based information processing is Structural and Structural Health Monitoring (SM/SHM) of technical structures.

One of the major challenges in sensing systems is the derivation of meaningful information from sensor input. Basically there are two different information extraction approaches: (I.) First those methods based on a system model of the technical structure, the device under test (DUT), and the sensor, and (II.) second those without any or with only a partial model. The latter class can profit from machine learning (ML), which usually bases on classification algorithms derived from supervised machine learning or pattern recognition using, e.g., self-organizing **[11]** and distributed multi-agent systems with less or no a-priori knowledge of the environment.

This work introduces some novelties compared to other data processing and agent platform approaches:

- Seamless integration of different host platforms (server, desktop computer, mobile devices, embedded devices, material-integrated sensing systems) with one unified agent model and portable platform architecture.
- The agent behaviour is modelled with an Activity-Transition Graph (ATG) and implemented entirely in *JavaScript* with a restricted and encapsulated access to the platform API (*AgentJS*).
- The novel proposed *JS* Agent Machine (*JAM*) is capable to execute *AgentJS* agents in a sandbox environment with full run-time protection. Agents can migrate between different *JAM* nodes seamless preserving their data and control state by using a on-the-fly code-to-text transformation in an extended *JSON+* format. A Distributed Organization System (DOS) layer provides *JAM* node

connectivity and security in the Internet. *JAM* provides machine learning as a service.

- Agent mobility crossing different execution platforms in networks and agent interaction by using tuple-space databases and global signal propagation aid solving data distribution and synchronization issues in the design of distributed sensor networks.
- Agent privilege levels based on capability rights and operational restrictions ensure agent authorization and platform security.
- Event-based information distribution and processing with agents reduces communication and overall network activity significantly, leading to reduced energy consumption. Regional-distributed on-line learning of pre-processed sensor data allows the prediction of the system response based on data from prior on-line training runs with selected system conditions. Machine learning is a platform service that can be accessed by the agent.

The next sections introduce the activity-based agent processing model, available mobility and interaction, and the proposed *JavaScript* agent platform architecture related to the programming model. Finally, the agent platform is evaluated with a case study and further applications are discussed.

II. The Activity-Transition-Graph-based AAPL Agent Behaviour Model

The implementation and deployment of mobile multi-agent systems with embedded and mobile systems is a complex design challenge. High-level agent programming and behaviour modelling languages can aid to solve this design issue. Reactive activity-based agent models can aid to carry out multi-agent systems on low-resource platforms.

In this work, agents are modelled based on a reactive behaviour model. The behaviour of a reactive activity-based agent is characterized by an agent state, which is changed by activities. Activities perform perception, plan actions, and execute actions modifying the control and data state of the agent. Activities and transitions between activities are represented by an activity-transition graph (ATG). The ATG-based agent-orientated programming language *AAPL* **[7]** was designed to offer the modelling of the agent behaviour on an easy-to-understand programming level, defining activities with procedural statements and transitions between activities with conditional expressions (predicates), which is suitable for beginners and non-experts. Though the imperative programming model is quite simple and closer to a traditional PL it can be used as a common source and intermediate representation for different agent processing platform implementations (hardware, software, WEB, simulation).

Definition: There is a multi-agent system (MAS) consisting of a set of individual mobile agents $\{A_1,A_2,..\}$. There is a set of different agent behaviours, called classes $\mathcal{C}=\{AC_1, AC_2,..\}$. A class AC consists of an Activity-Transition Graph that can be modified at run-time, and agent body variables. Activities perform actions like computation, migration, agent creation or destruction, and interaction with other agents. An agent is initially derived from one class. In a specific situation an agent A_i is processed on a network node N_l (e.g., mobile device) at a unique spatial location l. There is a set of different nodes $\mathcal{N}=\{N_1, N_2,..\}$ arranged in networks with physical or logical connectivity (e.g., a two-dimensional grid). Each node is capable to process a number n_i of agents in parallel or sequentially scheduled. An agent including its state can migrate to a neighbour node preserving its state (snapshot) where it continues working. There are agent sub-classes $\mathcal{SC} \subseteq \mathcal{S} \subseteq \mathcal{C}$ derived from super classes $\mathcal{S}$, enabling agent behaviour specialization and agent goal selection at run-time.

Therefore, the agent behaviour and the action on the environment is encapsulated in agent classes, with activities representing the control state of the agent reasoning engine, and conditional transitions connecting and enabling activities. Activities provide a procedural agent processing by a sequential execution of imperative data processing and control statements. Agents can be instantiated from a specific class at run-time. A multi-agent system composed of different agent classes enables the factorization of an overall global task in sub-tasks, with the objective of decomposing the resolution of a large problem into agents in that they communicate and cooperate with each other.

The activity-graph based agent model is attractive due to the proximity to the finite-state machine model, which simplifies the programmatically implementation.

An activity is activated by a transition depending on the evaluation of (private) agent data (conditional transition) related to a part of the agents belief in terms of *BDI* architectures, or using unconditional transitions (providing sequential composition), shown in Fig. **2**. Each agent belongs to a specific parameterizable agent class *AC*, specifying local agent data (only visible for the agent itself), types, signals, activities, signal handlers, and transitions.

A. AAPL Agent Classes, Reconfiguration, and Instantiation

The *AAPL programming model* (detailed description in **[7]**) relies on the ATG behaviour model by means of an agent class template containing activity and transition sections, shown in . Fig. **2**. *APPL* offers statements for parameterized agent instantiation, like the creation of new agents and the forking of child agents inheriting the parent state, using the `new(args)` and `fork(args)` statements, respectively.

There are statements for ATG transformations and composition. Agents can modify their behaviour at run-time by reconfigure the ATG using `transitionX(`a_i`,`a_j`,cond?)` and `activityX(`a_1`,`a_2`,..)` statements with `X`=`+` (add), `-` (remove), `*` (update), respectively.

B. AAPL Agent Interaction

Furthermore, unified agent interaction is provided by using synchronized Linda-like tuple database space access and signal propagation (messages carrying simple data delivered to asynchronous executed signal handlers).

Access of the tuple space is granted by using `in(TP)`, `rd(TP)`, `rm(TP)`, `exist?(TP)`, and `out(T)` primitives (`T`: n-dimensional value tuple, `TP`: n-dimensional tuple with value patterns), reading, removing, testing, or generating tuples, respectively. Reading bases on template pattern matching and can block agent execution if there is actually no match. Tuple-space communication is generative, i.e., a tuple is independent of the generating process after generation, and can have a lifetime beyond the generating process.

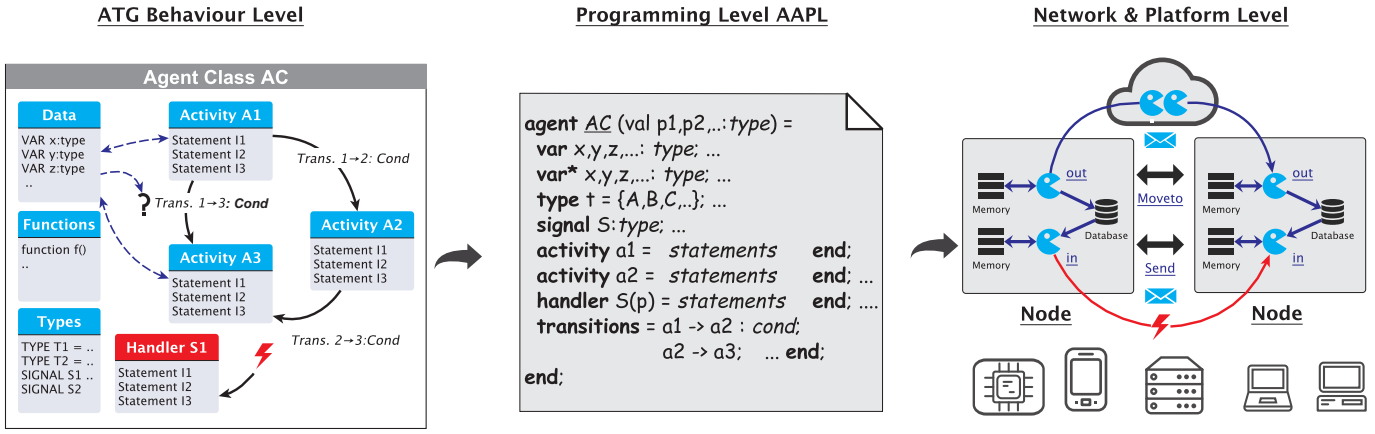

Fig. 2. Agent behaviour programming level with activities and transitions (AAPL, middle); agent class model and activity-transition graphs (left); agent instantiation, processing, and agent interaction on the network node level (right) **[12]**.

For some situations, tuple can remain in the tuple space never consumed. To avoid a flooding of tuple spaces with "orphan" tuples, the `mark(tmo,T)` operation can be used to store tuples with a limited lifetime `tmo`, which are destroyed by the TS manager automatically. These marking are extensively used in divide-and.-conquer systems discussed in the following sections. A signal `SIG` can be sent to an agent specified by its `ID` identifier using the `send(ID,SIG,ARG)` statement. A signal can be send to a group of agents of a specified agent class `AC` and within a given local range Δ by using the `broadcast(AC,Δ,SIG,ARG)` statement.

C. *AAPL Agent Mobility.*

Agent mobility (migration) is provided by the `moveto(DIR)` statement. The destination can be a geometric direction (e.g., `NORTH`), a delta-distance vector, a host port, or an URL/IP address.

D. *AAPL Machine Learning*

Learning agents can access basic machine learning operations provided by the platform as a service, using *`model`*`= learn(`*`datasets, classes, features, alg`*`?)` and `classify(`*`model, dataset`*`)` primitives. The agent stores only the learned model, and do not carry any learning algorithms or previous training sets.

Finally, agents can store and look-up functions in a code dictionary by using the `store(`*`name, fun`*`)` and *`fun`*`= lookup(`*`name`*`)` primitives.

III. JAM: The JavaScript Agent Processing Machine Platform

Commonly, agents are implemented with some high-level modelling and programming language having a specific, often abstract, agent behaviour and interaction model, that is finally compiled to machine code for native host platforms or virtual machines. The *AAPL* Programming Language is a common base for software and hardware implementations of reactive agents supporting a wide range of agent processing and host platforms including microchip hardware designs **[12]**. 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 **[7]**. 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 it would be desirable to implement *AAPL* agents directly in *JavaScript* (*JS*), which is a well known 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 recomposition 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.

A. *AgentJS: JavaScript Object and extended Code-to-Text JSON+ Representation*

Textual representations used as an 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 codings. 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.

B. 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. Though 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 not portable 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 the next sub-section. A creation of a new agent will always first stringify the agent object, and finally coding back a sandboxed 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.

C. AIOS: The Agent Execution and IO Environment for AgentJS

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 of 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.

D. Agent Scheduling and Checkpointing

Unfortunately, *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 of 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. Though 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, checkpointing 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 checkpointing 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 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.

```
1   agent explorer(dir,radius)
2    var x,y:int;
3    var mean,hop:int;
4    var goback:boolean;
5    activity init = ..
6    end;
7    acitivity move =
8      if (hop=radius) goback := true;
9      else begin hop++; moveto(dir); end;
10   end;
11   activity percept =
12     var s:int; rd(SENSOR,s?);
13     mean := (mean+s)/2;
14   end;
15   activity goback =
16     dir=opposite(dir);
17   end;
18   activity deliver =
19     out(MEAN,mean); signal($parent,DELIVER);
20   end;
21   handler S(v) = .. end;
22   transitions =
23     init->move;
24     move->percept:not goback;
25     move->move:goback and hop>0;
26     move->deliver:goback and hop=0;
27     move->goback:hop=radius;
28     percept->move;
29     goback->move;
30     deliver->end;
31   end;
32 end;
33
```

```
var explorer = function (dir,radius) {
  this.dir=dir; this.radius=radius;
  this.x=0; this.y=0; this.mean=0;
  this.hop=0; this.goback=false;
  this.act = {
   init:function() {..},
   move:function() {
    if (this.hop==radius) this.goback=true;
    else { this.hop++; moveto(this.dir);}
   },
   percept: function () { var s;
    B([function () {rd(['SENSOR',_],function(t) {s=t[1]}),
       function () {mean = (mean+s)/2}]);
   },
   goback: function () {
    this.dir=opposite(this.dir);
   },
   deliver: function () {
    out(['MEAN',this.mean]); signal(parent(),DELIVER);
    }};
  this.on = { S:function(v) {..} .. };
  this.trans = {
   init: function () {return 'move'},
   move: function () {
    if (!this.goback) return 'percept';
    else if (this.goback && this.hop>0)
      return 'move';
    else if (this.goback && this.hop==0)
      return 'deliver';
    else if(hop==radius) return 'goback'},
   goback: function () {return 'move'},
   deliver: function () {retunr 'end'}};
  this.next='init'};
```

Example 1.A simple neighbourhood explorer agent programmed in *AAPL* (left) and the corresponding *AgnetJS* code (right). Note: In *AgentJS* `this` always references the agent object, even in deeper context levels.

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 not capable 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, 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.

E. 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 Ex. **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 an an 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 inside *AgentJS* activity, transition, callback, and first-order function calls of *AIOS* provided functions is always bound to the agent object self reference! 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.

F. 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 untrusting networks not known 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 trusting and untrusting agents, different agent privilege levels were introduces, providing different *AIOS* API sets. Privilege level 0 is the lowest level and grants agents only computational and tuple-space IO statements. Level 0 agents are not allowed to replicate, to create or kill other agents, to send signals, and to modify their code. 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 - the most privileged - 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*. 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 (see Sec. IV.) with the appropriate rights. After migration, the privilege is lost and must be re-negotiated on a new platform using capabilities.

G. 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.

H. 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.

I. SEJAM: The JavaScript Agent Simulator Environment

Commonly, execution and simulation platforms are completely different environments, and simulators are significantly slower in the agent execution compared to real-world agent processing on optimized processing platforms. *SEJAM* is a MAS *AgentJS* simulator implemented on top of the *JAM* platform layer, executing agents with the same VM as a standalone agent platform would do. This capability leads to a high-speed simulator, only slowed down by visualization tasks and user interaction. Furthermore, multiple simulators can be connected via a stream link (sockets, IP links, etc.), improving the simulation performance by supporting parallel agent processing. Furthermore, the simulator can be directly connected to any other *JAM* node.

IV. 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 **[7]** for details) - are Object-orientated Remote Procedure Calls (ORPC). 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 **[13]**. 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 **[13]** for details) using one-way encryption with a private port. All parts are merged in a capability structure `[srvport] obj(rights)[protport]`. Capabilities are also used in this work for agent-agent platform negotiation, with a server port designating a platform or platform network. The rights field can only changed with the original secret protection port (otherwise *protport* is invalid).

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 in this work by a broker server that is visible in the Internet or Intranet domain. To provide compatibility with and among all existing browser, *node.js* server-side, and client-side applications, a RPC based inter-process communication encapsulated in *HTTP* messages exchanged with the broker server operating as a router was invented. 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 a 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 a *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.

V. Platform Evaluation

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

***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, One physical JAM with a JAM world consisting of four logical (virtual) nodes, connected in a grid (ring) with virtual circuit links (queues).*

***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*

The creation (instantiation) or forking of new agents involves always a code-to-text and text-to-code transformation using the sandbox environment. The performance of this operation is shown in Tab. **I** and **II**, for a small and a complex agent class. 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 Tab. **III**. 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. Tab. **IV** 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 Tab. **V**. Finally, Tab. **VI** 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 test agent classes used in this performance evaluation are explained in Sec. VI..

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

Table I. 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: -

Table II. Test Case 2: Agent creation on one logical (virtual) node, complex learner agent (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)

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

Table III. 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 (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:20M,B :9MB

Table IV. 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

Table V. 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

Table VI. 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.

VI. Use case: Event-based Sensing and Learning with Multi-Agent Systems

Large scale perceptive and ubiquitous systems with hundreds and thousands of devices require data processing concepts far beyond the traditional centralized approaches. Multi-Agent systems can be used to implement smart and optimized sensor data processing in these distributed sensor networks. In the following use case of a sensor network used for structural monitoring (e.g., of buildings, bridges, wind energy wings), different data processing and distribution approaches are implemented with agents, shown in Fig. **3**, leading to a significant decrease of network communication activity and a significant increase of reliability and Quality-of-Service.

An event-based sensing behaviour is used to collect sensor information from sensing devices (nodes). Adaptive path finding (routing) supports agent migration in unreliable networks with missing links or nodes by using a hybrid approach of random and attractive walk behaviour. Self-organizing agent systems with exploration, distribution, replication, and voting behaviours are used to identify a region of interest (ROI, a collection of stimulated sensors) and to distinguish sensor failures (noise) from correlated sensor activity within this ROI.

It is assumed in this Ex. that sensor nodes are arranged in a two-dimensional grid network (as shown in Fig. **3**) providing spatially resolved and distributed sensing information of the surrounding technical structure, e.g., a metal plate (based on previous work in **[3]**), or composite material.

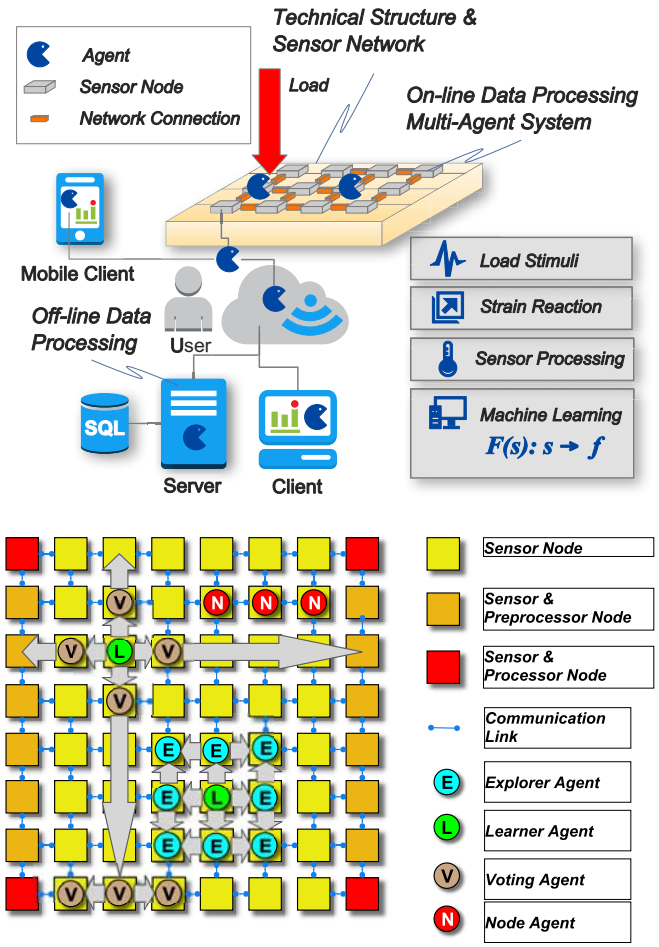

Fig. 3. The logical view of a sensor network with a two-dimensional mesh-grid topology integrated in a technical structure (Top) and examples of the population with different mobile and immobile agents (Bottom): node ,learner, explorer, and voting agents. The sensor network can contain missing or broken links between neighbour nodes.

Usually a single sensor cannot provide any meaningful information of the load of the mechanical structures. A connected area of sensors (complete sensor matrix or a part of it) is required to calculate the response of the material due to applied forces.

The complete sensor data processing system is partitioned into different agent classes. Some classes are super classes composed of sub-classes (e.g. the learner with the explorer and voter sub-classes). A sensor node is managed by a node agent, which creates and manages a learner agent, responsible for local sensor processing and prediction. If the node agent detects a change of its sensor(s), it will notify the learner agent by using a notification request tuple the learner is waiting for. This selects either the learning or classification modus of the learner. The learner agent will send out mobile explorer agents in a ROI that collect sensor data of the surrounding neigbourhood delivered back to the learner using tuple and signal I/O. After the data set is complete, either the learner uses the new data set for regional learning by accessing the platform ML API, or it uses the data set for a regional classification with the already learned and stored model. Each activated learner that performed a classification will send a classification vote to the outside of the network by creating voter agents (for redundancy four agents send out in orthogonal directions). The votes are further passed to the edge nodes and election agents, which collect all votes and compute the major global voted classification result., e.g., a specific detected load situation. A modified Decision tree learning model is used for load case classification, well suited due to the compact model representation (requiring less than 1kB storage). The event-based MAS behaviour and its temporal agent population is shown in Fig. **4**, retrieved from simulation. Each peak represents a run. Most load situations can be detected by this learning and major voting approach. Each learning/classification run requires about 0.5-1MB communication costs (using code compression) in the entire network only, and the agent population reaches up to 400 agents (peak value, but executed in the simulation by one physical *JAM* node), and a logical *JAM* node 10 agents.

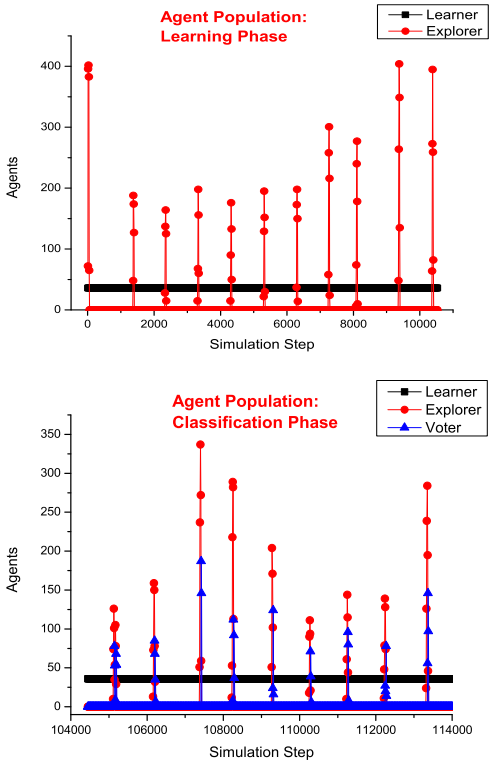

Fig. 4. Temporal agent population in the example network during learning (training) and classification phases (with 6 different load cases).

The structural monitoring learning approach as part of a sensor network can be extended to more global applications. E.g., ubiquitous perception using consumer devices equipped with sensors like smart phones, which can be used, e.g., for damage detection after earthquakes. Mobile learners can carry learned models for further processing on different nodes. Another case is cloud-based manufacturing with back propagation of sensor data gathered from products using *AgentJS* agents.

VII. Conclusions and Outlook

Large-scale distributed applications require a new processing and communication paradigm, which addresses scalability, adaptability, self-organization, robustness, and resource constraints. In this work, agents are represented by mobile *JavaScript* code (*AgentJS*) that can be modified at run-time by agents and that are processed by a modular and portable agent platform *JAM* in a protected sandbox environment encapsulating agent processes. *JAM* provides ML as a service, splitting algorithms (platform) from model data (agent), demonstrated with a structural monitoring use-case. *JAM* is implemented entirely in *JS*. The presented approach enables the development of perceptive clouds and smart systems of the future integrated in daily use computing environments and the Internet. Agents can migrate between different host platforms including WEB browsers by migrating the program code of the agent, embedding the state and the data of an agent, too, in an extended *JSON+* format. Migration of agents create snapshots and code-text-code transformations, which is executed with low latency as shown. The design and platform approach is suitable to cover the sensing, aggregation, and application layers of large-scale and massively distributed information processing systems efficiently. The Internet and WEB platform network is embedded in a distributed co-ordination and management shell providing an Object-Capability based RPC and global domain naming and file services. The RPC communication is encapsulated in generic IP/HTTP messages. A broker service is used to connect IP client-side only applications like WEB browsers or applications hidden in private networks, which are then fully capable of client- and server-side RPC communication.

The evaluation of the *JAM* shows the performance metrics and capability to process large-scale agent systems beyond the 1000 agent limit with low overhead, and confirm the suitability of *JAM* for low-resource embedded and mobile devices. Each *JAM* node is capable to execute up to 1000 agents with reasonable speed. The entire *JAM* (excluding *DOS*) platform requires about 600kB *JS* text code only, suitable for embedded systems.

References

[1] M. Caridi and A. Sianesi, *Multi-agent systems in production planning and control: An application to the scheduling of mixed-model assembly lines*, Int. J. Production Economics, vol. 68, pp. 29–42, 2000.

[2] M. Pechoucek, V. Marík, 2008. *Industrial deployment of multi-agent technologies: review and selected case studies*. Auton. Agent. Multi-Agent Syst. 17 (3), 397–431

[3] S. Bosse, A. Lechleiter, *Structural Health and Load Monitoring with Material-embedded Sensor Networks and Self-organizing Multi-agent Systems*, Procedia Technology, 2014, DOI: 10.1016/j.protcy.2014.09.039

[4] M. Guijarro, R. Fuentes-fernández, G. Pajares, *A Multi-Agent System Architecture for Sensor Networks*, Multi-Agent Systems - Modeling, Control, Prog., Simulations and Applications, 2008.

[5] D. Lehmhus, T. Wuest, S. Wellsandt, S. Bosse, T. Kaihara, K.-D. Thoben, and M. Busse, *Cloud-Based Automated Design and Additive Manufacturing: A Usage Data-Enabled Paradigm Shift*, Sensors MDPI, vol. 15, no. 12, pp. 32079–32122, 2015, DOI 10.3390/s151229905.

[6] R. Milner, *Communicating and mobile systems: the π-calculus*, Cambridge University Press, Cambridge (1999)

[7] S. Bosse, *Unified Distributed Computing and Co-ordination in Pervasive/Ubiquitous Networks with Mobile Multi-Agent Systems using a Modular and Portable Agent Code Processing Platform*, in The 6th International Conference on Emerging Ubiquitous Systems and Pervasive Networks (EUSPN 2015), Procedia Computer Science, 2015.

[8] R. Milner, *The space and motion of communicating agents*. Cambridge University Press, 2009.

[9] L. Chunlina, L. Zhengdinga, L. Layuanb, and Z. Shuzhia, *A mobile agent platform based on tuple space coordination*, *Advances in Engineering Software*, vol. 33, no. 4, pp. 215–225, 2002

[10] Z. Qin, J. Xing, and J. Zhang, *A Replication-Based Distribution Approach for Tuple Space-Based Collaboration of Heterogeneous Agents*, Research Journal of Information Technology, vol. 2, no. 4. pp. 201–214, 2010

[11] V. Di Lecce, M. Calabrese, and C. Martines, *From Sensors to Applications: A Proposal to Fill the Gap*, Sensors & Transducers, vol. 18, no. Special Isse, pp. 5–13, 2013.

[12] S. Bosse, *Design of Material-integrated Distributed Data Processing Platforms with Mobile Multi-Agent Systems in Heterogeneous Networks*, Proc. of the 6'th International Conference on Agents and Artificial Intelligence ICAART 2014. DOI:10.5220/0004817500690080

[13] S. J. Mullender and G. van Rossum, Amoeba: *A Distributed Operating System for the 1990s*, IEEE Computer, vol. 23, no. 5, pp. 44–53, 1990