Distributed Machine Learning with Self-organizing Mobile Agents for Earthquake Monitoring

Stefan Bosse
University of Bremen, Department of Mathematics & Computer Science, Germany

Abstract - **Ubiquitous computing and The Internet-of-Things (IoT) raises rapidly in today's life and is becoming part of self-organizing systems (SoS). A unified and scalable information processing and communication methodology using mobile agents is presented to merge the IoT with Mobile and Cloud environments seamless. A portable and scalable Agent Processing Platform (APP) is an enabling technology that is central for the deployment of Multi-agent Systems (MAS) in strong heterogeneous networks including the Internet. A large-scale distributed heterogeneous seismic sensor and geodetic network used for earthquake analysis is one example, which can be extended by ubiquitous sensing devices like smart phones. To simplify the development and deployment of MAS in the Internet domain agents are directly implemented in JavaScript (JS). The proposed JS Agent Machine (JAM) is an enabling technology. It is capable to execute AgentJS agents in a sandbox environment with full run-time protection, low-resource requirements, and Machine Learning as a service. A simulation of a seismic network and real earthquake data demonstrates the deployment of the JAM platform. Different (mobile) agents perform sensor sensing, aggregation, local learning and prediction, global voting, and the application.**

Keywords - Agent Platforms, Self-organizing Systems, Distributed Learning, Earthquake Monitoring

I. Introduction and Overview

The IoT and mobile networks get real in today's life and is becoming part of pervasive and ubiquitous computing networks with distributed and transparent services. Robustness and scalability can be achieved by self-organizing and self-adaptive systems. Agents are already deployed successfully in sensing, 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. Mobile Multi-agent systems can fulfill the self-organizing and adaptive (self-*) paradigm **[3]**. Distributed data mining and Map-Reduce algorithms are well suited for self-organizing MAS. Cloud-based computing with MAS means the virtualization of resources, i.e., storage, processing platforms, sensing data or generic information **[4]**. Mobile Agents reflect a mobile service architecture. Commonly, distributed perceptive systems are composed of sensing, aggregation, and application layers, shown in Fig. **1**, merging mobile and embedded devices with the Cloud paradigm **[5]**. But generic Internet, IoT, and Cloud environments differ significantly in terms of resources: The IoT consists of a large number of low-resource or mobile devices interacting with the real world 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, which can be fulfilled by the mobile agent-based computing paradigm, discussed in this work.

The scalability of complex ubiquitous 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]**, sensing, 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. A sensor network as part of the IoT is composed of low.-resource nodes **[5]**. Smart systems are composed of more complex networks (and networks of networks) differing significantly in computational power and available resources, raising inter-communication barriers. These smart systems unite sensing, aggregation, and application layers **[5]**, shown in Fig. **1**, requiring a unified design and architecture approach. 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, superior compared with common Java frameworks, e.g. Jason **[6][7]** or the *JADE* platform **[14]**, and object-orientated systems like SWARM **[15]**, not supporting mobile agents. In this work the reactive behaviour of mobile agents are modeled with dynamic Activity-Transition Graphs (ATG). The agent behaviour is implemented entirely in *JavaScript* with a restricted and encapsulated access to the platform API. This agent-specific mobile program code *AgentJS* can be executed on a variety of different host platforms using *JAM* and a generic *JS* VM, closing the gap between the IoT and Cloud infrastructures. The *JS* Agent Machine (*JAM*) is an enabling technology that is capable to execute thousands of *AgentJS* agents in a sandbox environment with full run-time protection.

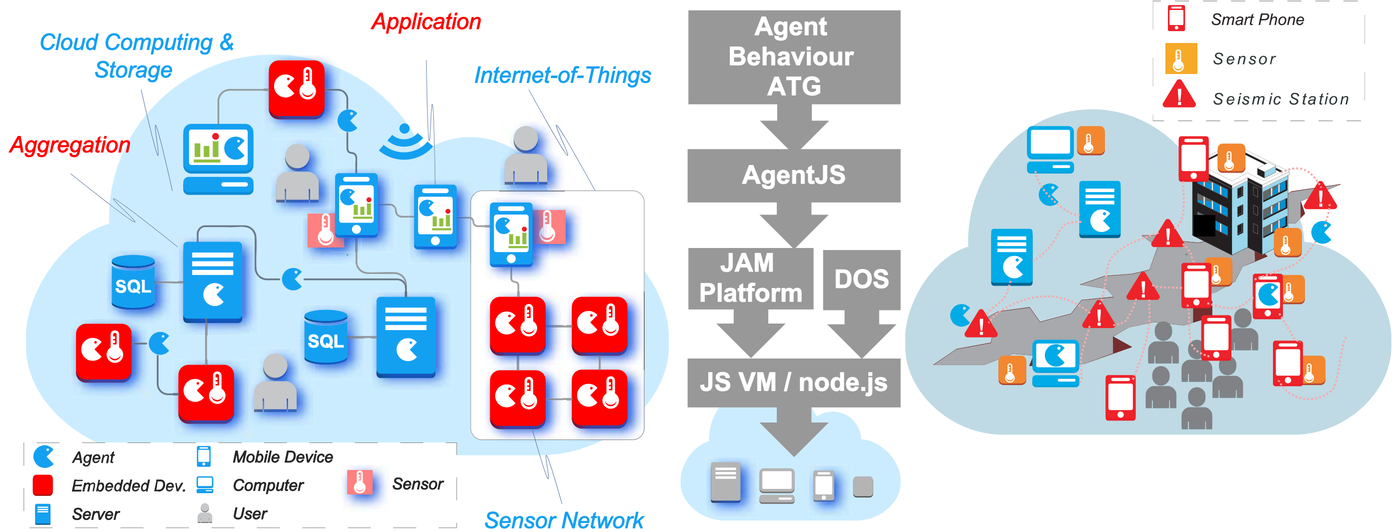

Fig. 1. (Left) Unified IoT - Cloud Distributed Perception and Information Processing with mobile agents (Middle) Portable JavaScript (JS) Agent Machine Platform (JAM) and an optional Distributed Organization System (DOS) layer **[8]** adds connectivity and security to JAM in the Internet domain. (Right) Use-case example: Deployment in a seismic network extended with smart phones, building networks, and mobile devices for distributed earthquake analysis.

Agents can migrate between different *JAM* nodes seamless preserving their data and control state by using on-the-fly code-to-text transformation. Agent privilege levels based on capability rights and operational restrictions ensure agent authorization and platform security. Agent self-re-configuration of the ATG at run-time enabling self-adaptive and feedback learning agents. A Distributed Organization System (DOS) layer provides *JAM* node connectivity and security in the Internet. This approach requires only a minimal Agent Processing Platform Service (APPS), extended with an object-orientated RPC communication, implemented in *JS*, too.

Current Just-in-Time (JIT) compiling VMs pose reasonable performance comparable to native code. The minimal APPS provides tuple-based agent interaction services (see **[7]**) 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.

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

- Seamless integration of different host platforms (server, desktop computer, mobile devices, smart phones, embedded devices, material-integrated sensing systems) with one unified agent model and the deployment of the portable *JAM* platform architecture.
- Scalable and event-based sensor data processing
- Machine Learning as a platform service: Agents can carry learned models (mobile learner) without carrying the learner code. Incremental learning avoids an accumulated database.
- Distributed on-line learning and classification with regional data combined with distributed global voting of regionally classified prediction situations with majority election.

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 deployment of the agent platform is demonstrated and evaluated by a simulation using a JAM network representing the north american seismic network (CI), and earthquake learning based on reduced seismic data and MAS self-organization.

II. AgenJS: The JS Agent Behaviour Model

In this work, a novel agent process platform *JAM* is used that provides Machine Learning (ML) as a service. *JAM* is implemented entirely in *JavaScript* (*JS*) including the ML service. *JAM* is capable of executing agents programmed in *JS*, called *AgentJS*, in a protected sandbox environment.

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 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 which they communicate and cooperate with one other.

Agent interaction is required in MAS, providing synchronization and data exchange. The tuple-space communication paradigm with a set of simple but synchronizing access operations (input, output, read, remove) is well accepted and an understood approach. Signals can be used instead for simple distributed one-way notifications carrying simple data.

The activity-graph based agent model is attractive for fine-grained agent scheduling. An activity is always executed atomically, but after an activity terminates, it is a well defined break point for agent process scheduling.

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 the Belief-Desire-Intention (*BDI*) architecture, or using unconditional transitions (providing sequential composition). 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. The principle *AgentJS* structure of an agent class is shown in Def. **1**. In contrast to common *JS* objects, an *AgentJS* class definition may not use any references to free variables or functions. The `this` variable references always the agent object, and can be used, e.g., in transition functions, handlers, activities, and first order functions directly.

Def. 1. Principle structure of an AgentJS Class Definition with a set of activities $\{a_1,a_2,..\}$ encapsulated in an activity section, followed by the transition section implementing the agent ATG.

```
1   var ac = function(p1,p2,..) {
2     this.p1=p1; .. Parameter
3     this.v1=0;  .. Varaiables
4     Activities
5     this.act = {
6       init: function () {..},
7       a1: function () {..},
8       a2: function () {..},
9       ..
10      end: function () {..}
11    };
12    Error and Signal Handler
13    this.on = {
14      error: function (e) {..},
15      exit: function (e) {..},
16      SIG1: function (v) {..},
17      ..
18    };
19    Transitions
20    this.trans = {
21      init: function () {return a1},
22      a1: function () {..},
23      a2: function () {..},
24      ..
25    };
26    this.next=init;
27  }
```

III. JAM: The JavaScript Agent Machine Platform

JAM consists of different modules entirely implemented in *JS* that can be executed by any standalone *JS* VM or within WEB browsers. The deployment in Internet and client-side applications like browsers and the Internet require a Distributed Co-ordination and Operation System layer (DOS) with a broker service (details can be found in **[8]**).

A. Agent Execution Environment

The *AIOS* is the main execution layer of *JAM*. It consists of the sandbox execution environment encapsulating an agent process, with different privileged API sub-sets depending on current agent role. Furthermore, the *AIOS* module implements the agent process scheduler and IO. The sandbox environment provides restricted access to a code dictionary based on the privilege level, enabling code exchange between agents. Level 0 agents are not privileged to replicate, create, or kill other agents and to modify their code.

B. Agent Creation and the Sandbox Environment

Agents are either instantiated from an agent class template or forked from already existing agents. The template is genuine *JS* with some behavioural modifications, that can be transformed in the textual *JSON*+ representation, derived from the *JS* Object Notification format (*JSON*), using a modified parser and text converter. *JSON*+ includes additional function code. Agents are executed always in a sandbox environment, which requires always a code-text-code transformation that is performed on agent creation or migration, discussed below.

C. Agent Roles

To distinguish at least trusting and untrusting agents, different agent privilege levels were introduces, providing different *AIOS* API sets, with level 0 as the lowest level that grants agents only computational and tuple-space IO statements. Level 1 agents can access the common *AIOS* API operations, including agent replication, creation, killing, sending of signals, and code morphing. Level 2 agents are capable to negotiate 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*. 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.

D. Agent Mobility

Agent mobility, provided by the *AIOS* `moveto(to)` statement, requires a process snapshot and the transfer of the data and control state of the agent process. 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. Thus, the migration starts with a code-to-text transformation to the extended *JSON*+ representation of the agent object, 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.

E. *Agent Interaction*

Agents can interact with each other by using a tuple-space database part of *JAM*. *AIOS* provides the common tuple-space access operations (e.g., `out(tup)`, storing a tuple *tup*, `inp(pat,function(tup){..})`, matching, returning, and removing a tuple based on pattern *pat*,). Tuple space communication is generative, i.e., a tuple can survive the producer process/agent. The `mark(tmo,tup)` 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. A signal `SIG` can be sent to an agent `ID` using the `send(ID,SIG,arg)` statement, or sent 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.

F. *Machine Learning as a Service*

Learning agents can access basic machine learning operations provided as a platform service, offered by `model=learn(datasets, classes, features, alg?)` and `feature=classify(model, dataset)` primitives. The agent stores only the learned model, and do not carry any learning algorithms, leading to a separation of the learning algorithm (platform) from the data (agent).

G. *Security and the Distributed Organization System Layer*

In the simplest case *JAM* nodes are connected by peer-to-peer links. But large-scale network environments like the Internet are organized in hierarchical graph-like structures with changing and basically non-visible interconnections. To organize *JAM* nodes in such large-scale and heterogeneous networks, an additional Distributed Organization System (DOS) layer was added. Furthermore, privacy, security, and trust are addressed by the *DOS*. The fundamental communication concept of the *DOS* - that is entirely implemented in *JS* (see **[8]** for details) - are client-server Object-orientated Remote Procedure Calls (ORPC) **[9]**.

IV. Event-based and Self-* Distributed Learning

Many sensing applications operating stream-based collecting sensor data centralized and periodically, resulting in high communication and processing costs. Frequently, most of the sensor data do not contribute to new information about the sensing system. Only a few sensors will change their data beyond a noise margin. In previous work **[3]** it was shown that different data processing and distribution behaviours can be used and implemented with agents, leading to a significant decrease in network communication activity and a significant increase of the reliability and Quality-of-Service. Global self-organization based on a local event-based sensor processing and global distribution behaviour; Extended with an adaptive path finding (self-routing) supporting agent migration in unreliable networks with partially missing connectivity or nodes by using a hybrid approach of random and attractive walk behaviour; And local self-organizing agent systems with divide-and-conquer exploration, distribution, replication, and interval voting behaviours based on feature marking (details in **[3]**).

In this work, decentralized and self-organizing learning performed by mobile agents is added. Distributed learning divides a spatial distributed (sensor) data set in local regions and applies learning to limited local regions, based on a divide and conquer approach. Decision trees are simple and compact models derived from learning with training data, and well suited for agent-based learning. For the sake of simplicity, generic graph-based networks of nodes, e.g. nodes connected in the Internet, are mapped on a two-dimensional mesh-grid with a spatial neighbourhood placing, providing virtual paths for mobile agents based on physical location, shown in Fig. **2**. Each sensor or computational node represents an agent processing entity, which can be populated with mobile and immobile agents.

It is possible to perform incremental learning at run-time using trees **[10]**, very attractive for agent and SoS approaches. A learned model (carried by the learner agent) is used to map data set vectors on class values. The tree consists of nodes testing a specific attribute variable, i.e., a particular sensor value, creating a path to the leaves of the tree containing the classification result, e.g., the load situation class. Among the distribution of the entire learning problem, event-based activation of learning entities can improve the system efficiency significantly. Commonly the locally sampled sensor values are used for an event prediction, waking up the learner agent, which collect neighbourhood data by using a divide-and-conquer system with explorer child agents. Traditional Decision Tree Learner (DTL) (e.g., using the ID3 and *C4.5* algorithms) select data set attributes (feature variables) for decision making only based on information-theoretic entropy calculation to determine the impurity of training set columns (i.e., the gain). This is well suited for non-metric symbolic attribute values, like color names, shapes, and so on. The distinction probability of two different symbols is usually 1. Numerical sensor data is noisy and underlies variations due to the measuring process and the physical world. Two numeric (sensor) values a and b have only a high distinction (separation) probability if the uncertainty intervals $[a-\sigma,a+\sigma]$ and $[b-\sigma,b+\sigma]$ do not overlap. Among the entropy of a data set column, the standard deviation σ giving the value spreading of a specific column must be considered, too. To improve attribute selection for optimized data set splitting, a column ε-interval entropy computation was applied, that extends each value of a column vector with an uncertainty interval $[v_i-\varepsilon,v_i+\varepsilon]$. Values with overlapping intervals are considered to be non distinguishable, lowering the entropy $entropy(col,\varepsilon)$ considering the lower/upper bounds of variable values/intervals. The modified learning algorithm tries best variable separation and partition based first on the entropy of a column of the training data sets, and if this is not possible it performs separation based on the best column value deviation to find a good feature variable separation. The prediction (analysis and classification) algorithm is a hybrid approach consisting of the tree iteration, but uses a simple nearest-neighbourhood estimation for selecting the best matching feature values (or intervals) (details in **[13]**).

The event-based regional learning leads to a set of local classification results from different learners, which can differ significantly, i.e., the classification set can contain wrong predictions.

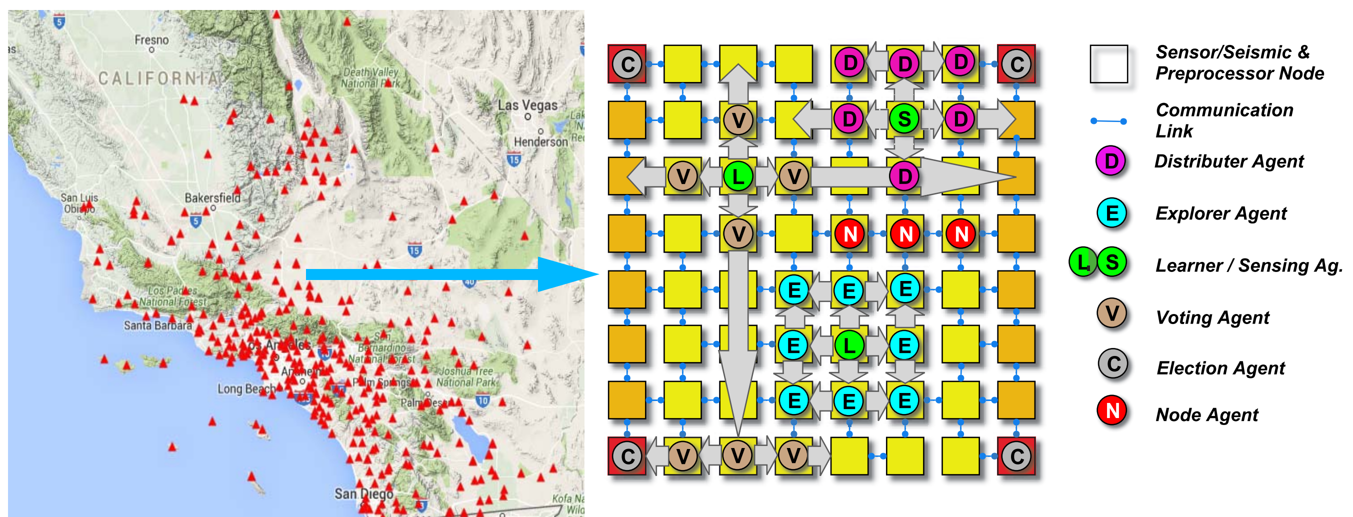

Fig. 2. (Left): The South California Seismic Sensor Network CI [Google Maps], explained in Sec. ; (Right) The logical view of the Sensor Network mapped on a logical two-dimensional mesh-grid topology with spatial neighbourhood placing, and examples of the population with different mobile and immobile agents: Node, learner, explorer, voting, and election agents. Non-mobile node agents are present on each node. Sensor nodes create learner agents performing regional learning and classification. Each sensor node has a set of sensors attached to the node, e.g., vibration/acceleration sensors.

To suppress wrong local predictions, a global vote election with majority decision is applied. All learner agents send their results to election nodes using voter agents. This election result is finally used for the system prediction. The variance of different votes can be an indicator for the trust of the election giving the right prediction. Learner agents can migrate carrying an already learned local model.

V. Use-case: Distributed Earthquake Analysis

To demonstrate the capability of the *JAM* APPS and the *AgentJS* programming model, a complex use case was selected: On-line Distributed Seismic data evaluation by a hierarchical and self-organizing MAS. It is assumed the seismic stations are connected to the Internet, or any other communication link suitable for agent migration, and that they are equipped with the *JAM* APPS. In this use-case, the north american network CI (Southern California area) was chosen with about 180 stations. Different earthquake events and big test data sets taken from the South California Earthquake data center **[11]** were used and processed by the learning MAS.

One major challenge is data reduction. The original test data contains temporal resolved seismic data of at least three sensors (horizontal East, horizontal North, and vertical acceleration sensors) with a time resolution about 10ms, resulting in a very high-dimensional data vector.

Usually a seismic sensor samples only noise below a threshold level, mainly resulting from urban vibrations and sensor noise itself. For machine learning, only specific vibration activity inside a temporal Region of Interest (ROI) is relevant. To reduce the high-dimensional seismic data, (I) The data is down sampled using absolute peak value detection, (II) searching for a potential temporal ROI, and (III) down sample the ROI data again with a final magnitude normalization and a 55-value string coding. The process is shown in Fig. **3**. The compacted 55-string coding assign normalized magnitude values to the character range `0`, `a-z`, and `A-Z` (`!`), with `0` indicating silence, and `!` overflow. If there were multiple relevant nearby vibration events separated by "silence", a `*` character separator is inserted in the string pattern to indicate the temporal space between single patterns.

The vibration (acceleration) is measured in two perpendicular horizontal and one vertical directions.

This gives a significant information for an earthquake recognition and localization. The data reduction is performed by a node agent present on each seismic measuring station platform. Only the compact string patterns are used as an input for the distributed learning approach. Based on this data, the learning system should give a prediction of an earthquake event and a correlation with past events. To deploy regional learning for a spatial ROI, seismic stations should be arranged in a virtual network topology with connectivity reflecting spatial neighbourhood, e.g., by arranging all station nodes in a two-dimensional network. The virtual links between nodes are used by mobile agents for exploration and distribution paths. They do not necessarily reflect the physical connectivity of station nodes.

To perform and evaluate the event-based and distributed learning approach introduced in the previous section, the *SEJAM* simulator is used (implemented on the top of *JAM*). It consists of multiple full operational virtual *JAM* nodes connected by virtual links enhanced with a GUI **[13]**. The seismic stations of the CI network are mapped on a two-dimensional grid with spatial proximity. Each (virtual) node in the network starts a resident node agent responsible for data sampling, reduction, and for the creation and notification of a learner agent. If the node agents detect vibration activity beyond a threshold, they will notify the learner agents via tuple-space interaction. The learner will sent out exploration agents that collect neighbourhood data, finally back delivered to the learner agent.

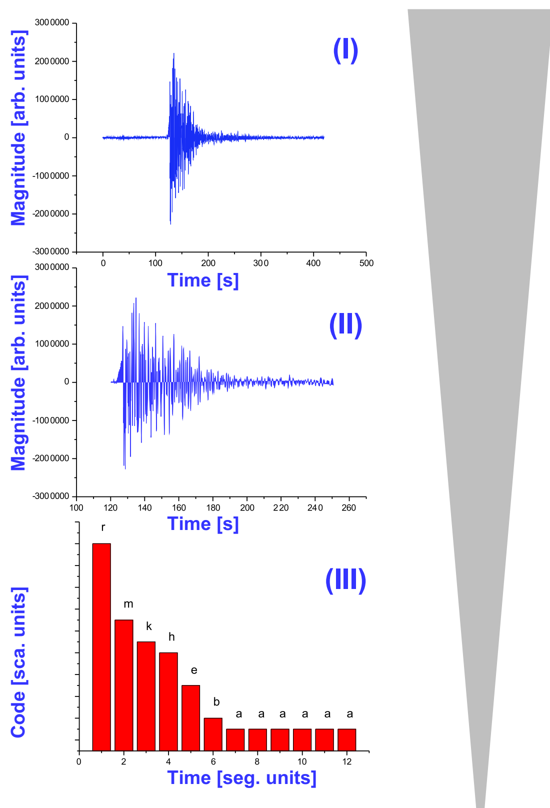

Fig. 3. Seismic data reduction: (I) Downsampling (1:16) with absolute peak value detection, (II) ROI analysis and ROI clipping, (III) Downsampling (1:64) and scaling/normalization with 55/57-string coding (0,a-z,A-Z,!,*)

This agent either learns a classification model based on the new data and an externally injected and available earthquake event marking, or applies the collected data to predict an already learned event. If an event was predicted, voter agents are sent out to notify election nodes, making the global decision about an earthquake event. The event-stimulated temporal agent population for the learning and prediction phases is shown in Fig. **4**.

The experimental setup uses Monte-Carlo simulation methods to add noise and uncertainty to the seismic input data (10%).

Classification probabilities (mean values from multiple classification runs, table) and some selected run-time examples of classifications are shown in Fig. **5**, all based on global majority election. Multiple learning runs were performed to train the network using a random sequence of different earthquake events with noisy data.. During the classification phase, a random sequence was used, too.

Most earthquake events can be recognized with a high prediction accuracy. The mean prediction probability for the correct classification was computed from the vote distribution of multiple experiments using Monte-Carlo simulation techniques (creating noisy sensor data).

The transition from learning to prediction is seamless and bases on the node/learner experience (learned events).

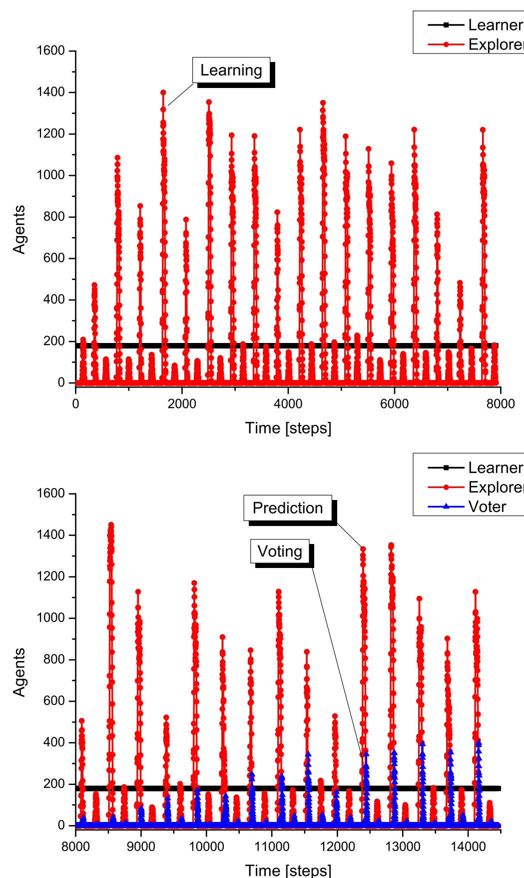

Fig. 4. Temporal agent population with 180 seismic nodes during learning (training, top) and classification (prediction and voting, bottom) phases.

Event Case	Mean Prediction Prob. [%]
CI.SILENCE	80.0
CI.2004.045	74.0
CI.2004.167	97.0
CI.2005.006	94.0
CI.2005.106	91.0
CI.2005.163	97.8

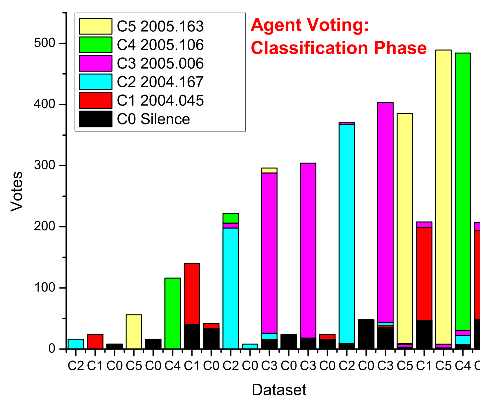

Fig. 5. (Top) Overall classification probability to hit the right event (Bottom) Selected classification distributions

Furthermore, after an event is elected by majority, this result can be back propagated to the learner adding the new data set as a new training set and performing incremental learning to improve further prediction accuracy. A typical learning and ROI exploration run in the entire network requires about 3-5MB total communication cost if code compression is enabled, which is a reasonable low overhead (with a peak value about 500-1000 mobile explorer agents operating in the network). Vote distribution produces only a low additional communication overhead (less than 1MB in the entire network).

VI. Outlook: Ubiquitous Devices as an Extension

In **[12]**, smart phones were successfully used to enhance the earthquake prediction by extending the seismic database with sensor data from mobile devices. In **[16]**, an open participatory platform for privacy-preserving social mining (Planetary Nervous System) was introduced, i.e., basically a virtualization of sensors that can profit from the proposed agent framework. The previously introduced learning system deployed in the seismic station network using the local station data can be extended by devices from such ubiquitous networks, which can execute the learner agents collecting sensor data (vibration, air pressure, temperature) from such devices. In contrast to seismic stations located at fixed and well known positions, mobile devices change their position dynamically. The mobile learner carrying an already learned spatially local model in a specific region, can migrate to mobile devices in this region and performs further learning or prediction. The extension of earthquake analysis with a large number of ubiquitous mobile devices can aid to improve disaster management significantly by providing spatially fine resolved sensor and event data covered by a high node density. Furthermore, building sensor networks can be included providing additional information about the buildings (health) state (illustrated in Fig. **1**, right side).

VII. Conclusions

Distributed learning with agents basing on local region perception and global voting was successfully deployed for seismic data analysis and earthquake recognition with a good prediction accuracy. It offers a self-organizing and robust learning approach. To suppress wrong local predictions, a global majority vote election is applied.

Agents are implemented with mobile *JavaScript* code (*AgentJS*) that can be modified at run-time by agents, processed by a modular and portable agent platform *JAM*. ML is provided as a service, splitting algorithms (platform) from model data (agent). *JAM* is implemented entirely in *JS* satisfying low-resource requirements. The presented approach enables the development of perceptive clouds and self-organizing 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. The entire *JAM* and DOS application requires about 600kB of compacted text code. Due to the autonomy and loosely coupling of *AgentJS* agents, a high degree of adaptivity and robustness is supplied, servicing as a pre-requisite for self-organizing systems in strong heterogeneous environments.

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. Maŕık, 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, *A hybrid approach for Structural Monitoring with self-organizing multi-agent systems and inverse numerical methods in material-embedded sensor networks*, Mechatronics, 2015, doi:10.1016/j.mechatronics.2015.08.005

[4] 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.

[5] 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.

[6] R. H. Bordini and J. F. Hübner, *BDI agent programming in AgentSpeak using Jason*, Computational Logic in Multi-Agent Systems, Volume 3900 of the series Lecture Notes in Computer Science, Springer, 2006, pp. 143-164.

[7] 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

[8] 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 Proc. of the 6th EUSPN 2015, Procedia Computer Science.

[9] 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

[10] F. Jiang, Y. Sui, and C. Cao, *An incremental decision tree algorithm based on rough sets and its application in intrusion detection*, Artif Intell Rev, vol. 40, pp. 517–530, 2013.

[11] http://scedc.caltech.edu/research-tools/eewtesting.html

[12] Q. Kong, R. M. Allen, L. Schreier, and Y.-W. Kwon, *MyShake: A smartphone seismic network for earthquake early warning and beyond*, Sci. Adv., vol. 2, 2016.

[13] S. Bosse, S*tructural Monitoring with Distributed-Regional and Event-based NN-Decision Tree Learning using Mobile Multi-Agent Systems and common JavaScript platforms*, Procedia Technol., SysInt Conference 2016

[14] F. Bellifemine and G. Caire, *Developing Multi-Agent Systems with JADE*. John Wiley & Sons, Ltd, 2007.

[15] N. Minar, R. Burkhart, C. Langton, and M. Askenazi, *The Swarm Simulation System : A Toolkit for Building Multi-agent Simulations*, Working Paper 96-06-042, Santa Fe Institute, Santa Fe., 1996.

[16] E. Pournaras, I. Moise, D. Helbing, *Privacy-preserving ubiquitous social mining via modular and compositional virtual sensors.* In 2015 IEEE 29th International Conference on Advanced Information Networking and Applications (pp. 332-338).