Incremental Distributed Learning with JavaScript Agents for Earthquake and Disaster Monitoring

Stefan Bosse

University of Bremen, Department of Mathematics & Computer Science, Scientific Centre Sensorial Materials, 28359 Bremen, Germany

ABSTRACT

Ubiquitous computing and The Internet-of-Things (IoT) emerge rapidly in today's life and evolve to Self-organizing systems (SoS). A unified and scalable information processing and communication methodology is required. In this work mobile agents are used to merge the IoT with Mobile and Cloud environments seamlessly. A portable and scalable Agent Processing Platform (APP) provides an enabling technology that is central for the deployment of Multi-agent Systems (MAS) in strongly heterogeneous networks including the Internet. A large-scale use-case deploying Multi-agent systems in a distributed heterogeneous seismic sensor and geodetic network is used to demonstrate the suitability of the MAS and platform approach. The MAS is used for earthquake monitoring based on a new incremental distributed learning algorithm applied to regions of sensor data from stations of a seismic network with global ensemble voting. This network environment can be extended by ubiquitous sensing devices like smart phones. Different (mobile) agents perform sensor sensing, aggregation, local learning and prediction, global voting and decision making, and the application. The incremental distributed learning algorithm outperforms a prior developed non-incremental algorithm (Distributed Interval Decision Tree learner) and can be efficiently used in low-resource platform networks.

Keywords - Agent Platforms, Self-organizing Systems, Distributed and Incremental Learning, Earthquake Monitoring, Pervasive and Ubiquitous computing, Disaster Management

INTRODUCTION AND OVERVIEW

The IoT and mobile networks emerge in today's life and are becoming relevant parts of pervasive and ubiquitous computing networks with distributed and transparent services. One major goal is the integration of sensor networks in the Internet and Cloud environments, with emerging robustness and scalability requirements. Robustness and scalability can be achieved by self-organizing and self-adaptive systems (self-*). Agents are already deployed successfully in sensing, production, and manufacturing processes, proposed by **Caridi (2000)**, and newer trends poses the suitability of distributed agent-based systems for the control of manufacturing processes, shown by **Pechoucek (2008)**, 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, shown in **Bosse (2015A)**. 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, discussed by **Lehmhus (2015)**. 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 as in **Lecce (2013)**. 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. The devices have strictly limited storage capacities and computing power, and the Cloud consists of large-scale computers with arbitrary and extensible computing power and storage capaci-

ties 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, posed by **Pechoucek (2008)**. Currently, many traditional processing platforms cannot yet handle a big number of agents with the robustness and efficiency required by the industry (**Pechoucek (2008)**), 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. In this work, the JavaScript Agent Machine (*JAM*) is used, supporting mobile *JavaScript* agents (*AgentJS*). *JAM* is entirely written in *JavaScript*, which can be executed on a wide variety of host platforms including WEB browsers. The platform is discussed in detail in **Bosse (2016B)**. **Lecce (2013)** concluded that a sensor network as part of the IoT is composed of low.-resource nodes. 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, **Lecce (2013)**, 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. Mobile *AgentJS* agents can operate in such strongly heterogeneous environments by using *JAM* and a new low-resource *JavaScript* engine *JVM*, suitable for embedded systems, in contrast to common approaches using Java-based frameworks (e.g., JADE and Jason, **Bellifemine (2007)**) or object-oriented systems like SWARM by **Minar (1996)**.

Machine Learning (ML) is used in a wide range of fields for state prediction or recognition. The concept of ML is closely related to the agent behaviour model, basically consisting of perception, planning (reasoning), and action. In recent years ML gets attraction for earthquake prediction and monitoring, supporting disaster management, e.g., using neuronal networks shown by **Alarifi (2012)**.

The *JAM* platform provides ML as a service for agents. Commonly, ML used to derive a classification model is performed centralized, i.e., all input data is collected and processed by one learner instance. For large scale distributed systems such a central instance is not suitable, is not conforming to the MAS architecture, and introduces a single point of failure. In **Bosse (2016C)**, a first attempt was made to deploy distributed decision tree learning (IDT) using agents to recognize earthquake events based on historic recorded seismic data. In this paper, an advanced incremental and distributed learning algorithm (I^2DT) is introduced, evaluated, and compared with the non-incremental algorithm. Incremental DT models are attractive due to their compact size and low resource requirements, compared, e.g., with *kNN* classifiers, which require a permanent data base. In the next section the *JAM* platform is summarized, followed by an introduction of distributed learning and the new incremental learning approach, that is finally evaluated with a simulation of a large-scale seismic network and historic real earthquake data.

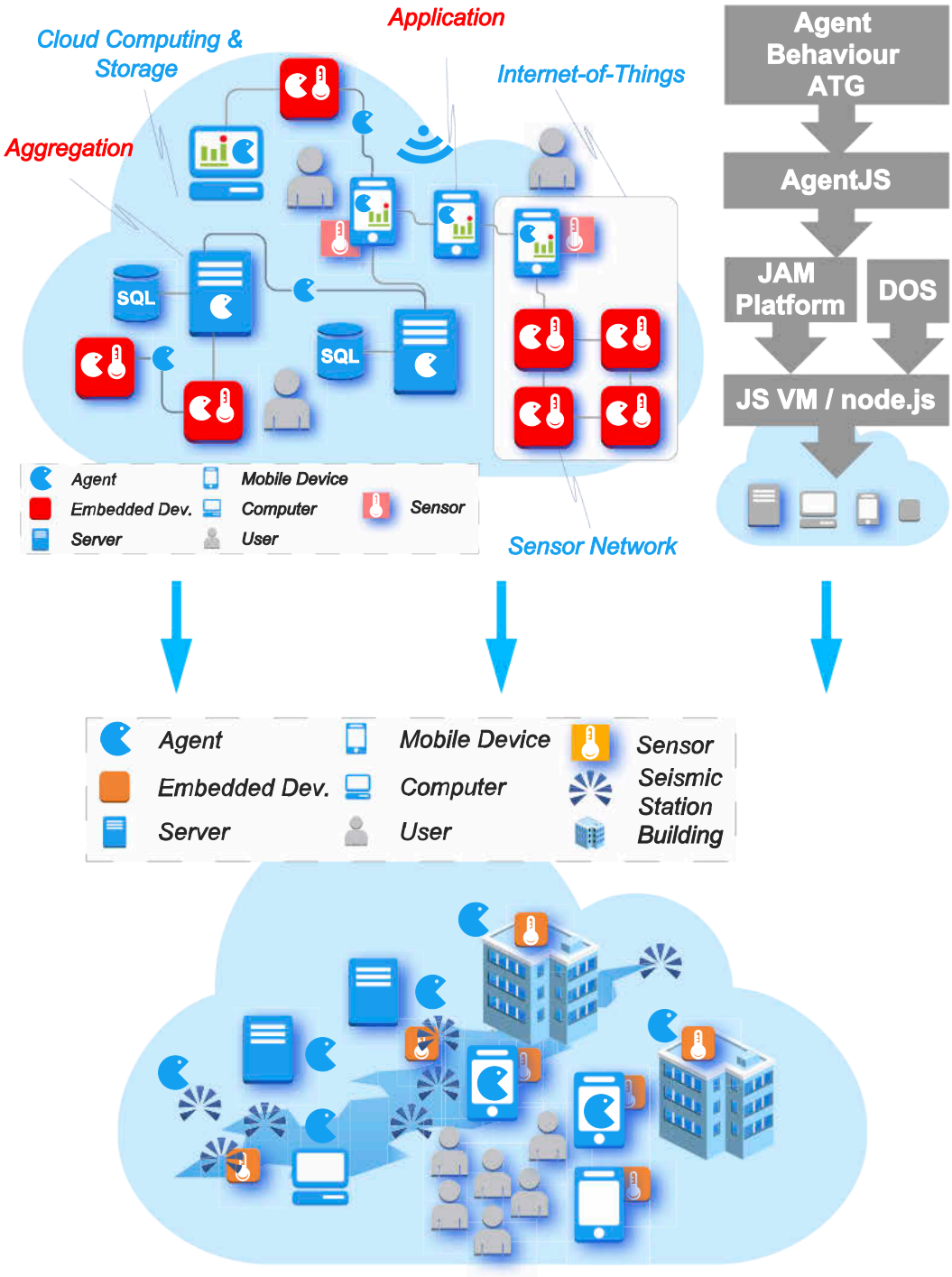

Fig. 1. *(Top, left) Unified IoT - Cloud Distributed Perception and Information Processing with mobile agents (Top, right) Portable JavaScript (JS) Agent Machine Platform (JAM) and an optional Distributed Organization System (DOS) layer adds connectivity and security to JAM in the Internet domain. (Bottom)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

JAM PLATFORM

In the use-case section a MAS is deployed in a large-scale strongly heterogeneous seismic network, which can be extended with smart phones. This heterogeneous network requires a unified agent processing platform, which can be deployed on a wide variety of host platforms, ranging from embedded devices, mobile devices, to desktop and server computers. E.g., some measuring stations are attached to buoy or installed on small islands, equipped only with low-power low-resource computers. To enable seamless integration of mobile MAS in Web and Cloud environments, agents are implemented in *JavaScript* (JS), executed by the JS Agent Machine (*JAM*), implemented entirely in *JS*, too. *JAM* can be executed on any *JavaScript* engine, including browser engines (Mozilla's *SpiderMonkey*), or from command line using *node.js* (based on *V8*) or *jxcore* (*V8* or *SpiderMonkey*), or a low-resource engine *JVM*, shown in Fig. **1**. The last three extend the JS engine with an event-based (asynchronous using callback functions) IO system, providing access of the local file system and providing Internet access. But these JS engines have high resource requirements (memory), preventing the deployment of *JAM* on low-power and low-resources embedded devices. For this reason, *JVM* was invented. This engine is based on *jerryscript* and *iot.js* from Samsung, discussed in **Gavrin (2015)**. *JVM* is a Bytecode engine that compiles JS directly to Bytecode from a parsed AST. This Bytecode can be stored in a file and loaded at run-time. *JVM* is well suited for embedded and mobile systems, e.g., the Raspberry PI Zero equipped with an ARM processor. *JVM* has approximately 10 times lower memory requirement and start-up time compared with *nodes.js*.

JAM consists of a set of modules, with the *AIOS* module as the central agent API and execution level. The deployment of agents in the Internet requires an additional Distributed Organization Layer (DOS, capability-based, see **Bosse (2015B)** & **Mullender (1990)**). *JAM* is available as an embeddable library (*JAMLIB*). The entire *JAM* and *DOS* application requires about 600kB of compacted text code (500kB Bytecode), and the *JAMLIB* requires about 400kB (300kB Bytecode), which is small compared to other APPs. *JVM*+*JAMLIB* requires only 2.7 MB total RAM memory on start-up.

JAM is capable of handling thousands of agents per node, supporting virtualization and resource management. Depending on the used JS VM, agent processes can be executed with nearly native code speed. *JAM* provides Machine Learning as a service that can be used by agents. Different algorithms can be selected by agents, the IDT and I^2DT are discussed in the next section. The agent only saves a learned model, but not the learner code. Agent interaction and synchronization is provided by exchanging data tuples stored in a tuple space on each *JAM* node.

The agent behaviour is modelled according to an Activity-Transition Graph (ATG) model. The behaviour is composed of different activities representing sub-goals of the agent, and activities perform perception, computation, and inter-action with the environment (other agents) by using tuple spaces and signals. Using tuple spaces is a common approach for agent communication, as proposed by **Chunlina (2002)**, much simpler than **Bordini (2006)** proposed with *AgentSpeak*. The transition to another activity depends on internal agent data (body variables). The ATG is entirely programmed in *JavaScript* (*AgentJS*, see **Bosse (2016B)** for details).

JAM agents are mobile, i.e., a snapshot of an agent process containing the entire data an control state including the behaviour program, can migrate to another *JAM* platform. *JAM* provides a broad variety of connectivity, shown in Fig. **2**, available on a broad range of host platforms. Although *JAM* is used in this work only as a simulation platform in the *SeJAM* simulator, it is ready to use in real-world networks and is capable to execute thousands of agents. The *SeJAM* simulator is built on top of a *JAM* node adding simulation control and visualization, and can be included in a real-world closed-loop simulation with real devices.

In real-world application security is an important key feature of a distributed agent platform. The execution of agents and the access of resources must be controlled to limit Denial-of-Service attacks, agent masquerading, spying, or other abuse, agents have different access levels (roles).

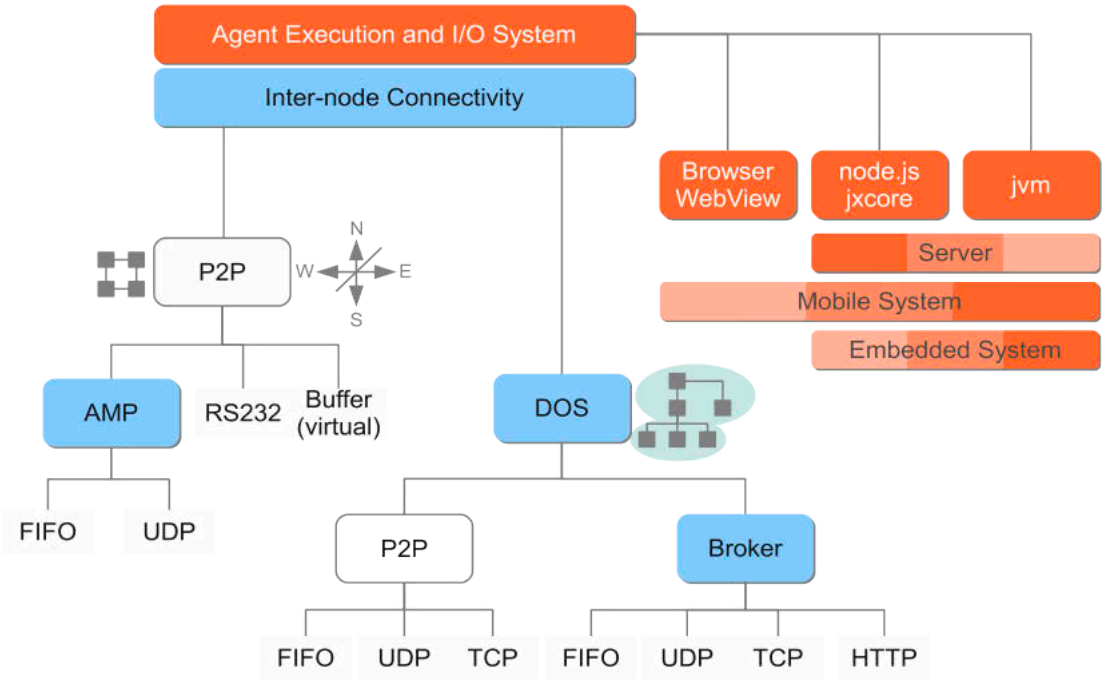

Fig. 2. *JAM connectivity and a broad range of Host and JavaScript platforms*

There are four levels:

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

The lowest level (1) does not allow agent replication, migration, or the creation of new agents. The *JAM* platform decides the security level for new received agents. An agent cannot create agents with a higher security level than its own. The highest level (4) has an extended AIOS with host platform device access capabilities. Agents can negotiate resources (e.g., CPU time) and a level raise secured with a capability-key that defines the allowed upgrades. The system level can not be negotiated. The capability is node specific. A group of nodes can share a common key (identified by a server port). A capability consists of a server port, a rights field, and an encrypted protection field generated with a random port known by the server (node) only and the rights field.

Among the AIOS level, other constrain parameters can be negotiated using a valid capability with the appropriate rights:

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

EVENT-BASED AND SELF-* DISTRIBUTED LEARNING

Many sensing applications operate stream-based collecting sensor data periodically in a centralized structure, 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 from **Bosse (2015A)** it was shown that different data processing and distribution behaviour 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 is 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. Regions of interest are explored using self-organizing agent systems with divide-and-conquer exploration, distribution, replication, and interval voting behaviour based on feature marking. Details can be found in **Bosse (2015A)**.

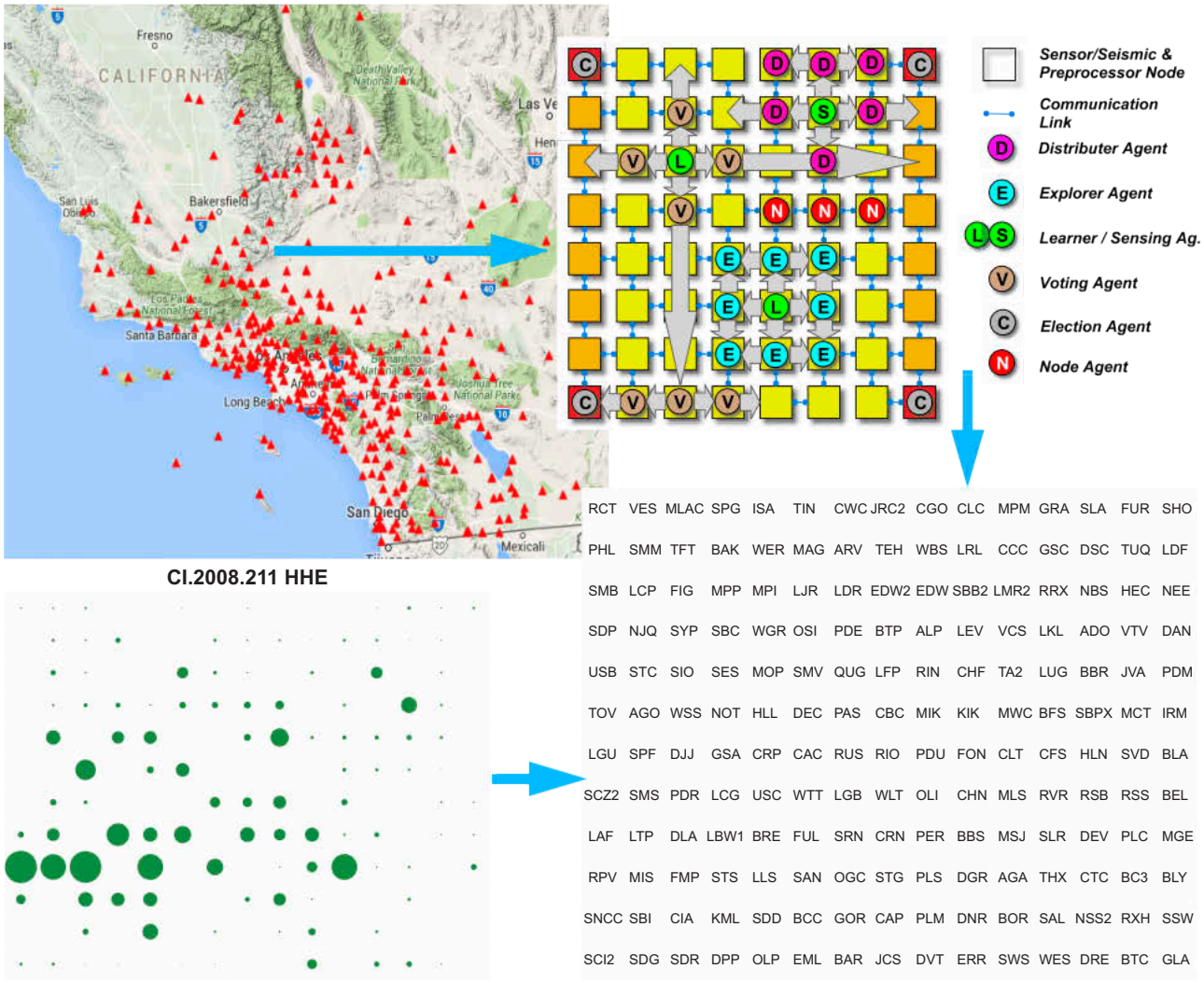

Fig. 3. *(Top, Left): The South California Seismic Sensor Network CI [Google Maps] (Top, Right) Sensor Network with stations mapped on a logical two-dimensional mesh-grid topology with spatial neighbourhood placing, and example population with different mobile and immobile agents. (Bottom, Left) A stimulated network situation (Bottom, Right) The station map.*

In this work, decentralized and self-organizing learning performed by mobile agents is used for classification and prediction tasks. 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. **3**. Each sensor or computational node represents an agent processing entity, which can be populated with mobile and immobile agents.

Jiang (2013) showed that it is possible to perform incremental learning at run-time using trees, very attractive for agent and SoS approaches, addressed later. A learned model (carried by the learner agent) is used to map data vectors (of an input variable set $x_1,x_2,..$; the feature vector) on class values (of an output variable y). The tree consists of nodes testing a specific feature variable, i.e., a particular sensor value, creating a path to the leaves of the tree containing the classification result, e.g., a mechanical load situation. Among the distribution of the entire learning problem, event-based activation of learning instances can improve the system efficiency significantly, and can be considered as part of the distributed learning algorithm (a pre-condition). 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 (DT) (e.g., using the ID3 and *C4.5* algorithms) select appropriate data set variables (feature variables) for decision making only based on information-theoretic entropy calculation to determine the impurity of training set columns (i.e., the gain of a feature variable). This is well suited for non-metric symbolic attribute values, like color names, shapes, and so on. The distinction probability of two different symbols of a feature variable is usually one. 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 best data set separation, each feature variable value v_i is mapped on a 2ε-interval $[v_i-\varepsilon,v_i+\varepsilon]$, that extends each value of a column vector with an uncertainty interval. The entropy is computed basing on these 2ε value intervals. 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 firstly on the entropy of a column of the training data sets, and secondly if this is not possible it performs separation based on the best column value deviation to find a good feature variable separation, shown in Fig. **4**. The prediction (analysis and classification) algorithm is a hybrid approach consisting of the tree iteration with a simple nearest-neighbourhood estimation for selecting the best matching feature values (or intervals) (details in **Bosse (2016A)**).

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.

To suppress wrong local predictions, a global vote election with majority decision is applied. All (activated) 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.

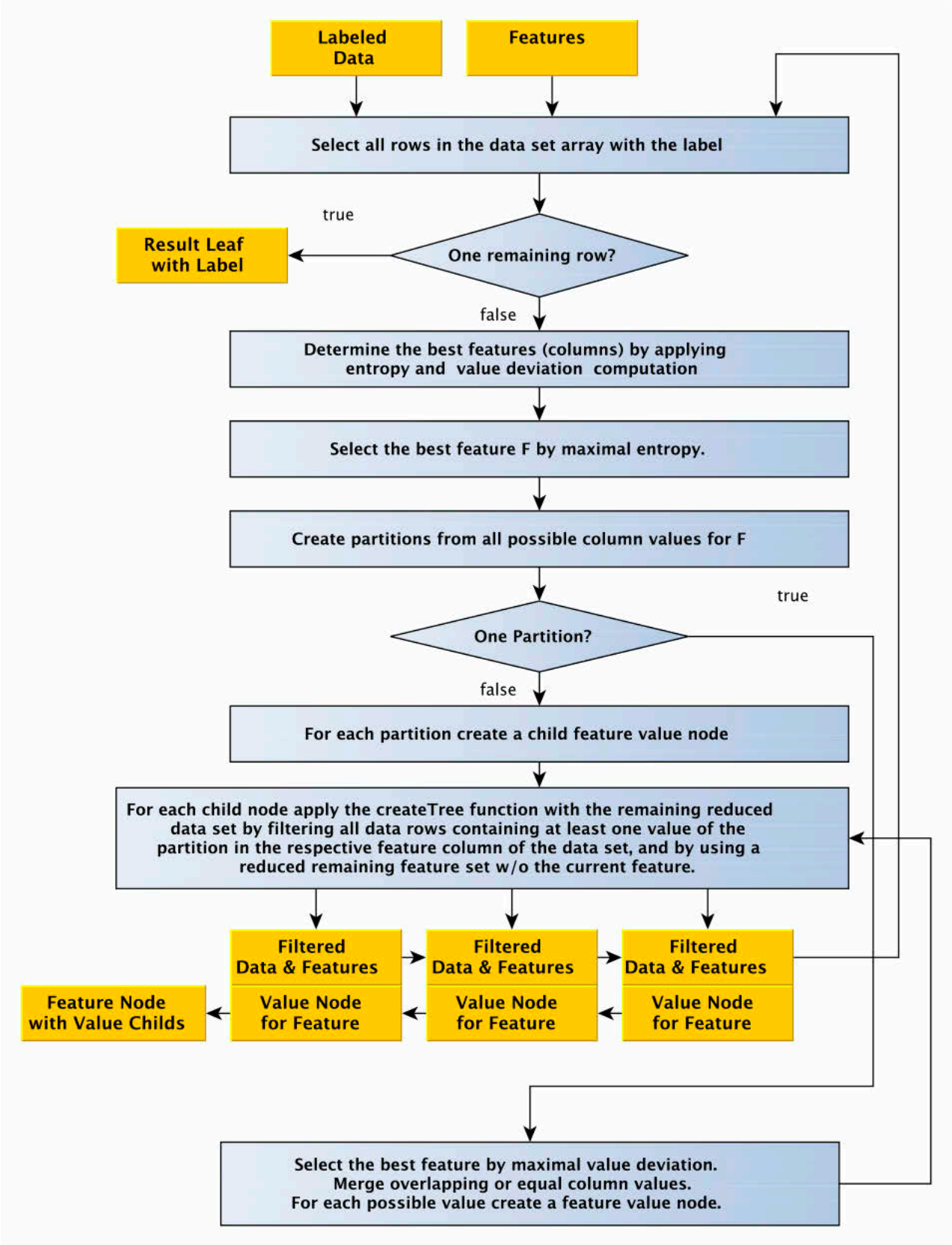

Fig. 4. *Algorithm of the original Interval Decision Tree Learner (IDT)*

INCREMENTAL BACK PROPAGATED LEARNING

Incremental learning algorithms can be divided in two approaches: (1) A learned model is updated with new training data sets by adding the new sets to a stored database of old training sets; (2) A learned model is updated with new training data sets but without a data base of old sets.

In this work agents perform learning and classification. An agent must store the training data and the learned model and it is useful to store only the learned model that can be updated at run-time without saving the entire history data. The new algorithm for the incremental updating of a learned model (decision tree) with new training set(s) is shown in Fig. **5** (in detail defined in Alg. **1**). The initial model can be empty. The current decision tree can be weakly structured for a new training set (new target), i.e., containing variables not suitable to separate the new data from old, that can result in a classification of the new target with insignificant variables. Therefore, if a new node is added to the tree the last node is expanded with an additional strong (most significant) variable of the new data set (it is still a heuristic for future updates), i.e., creating an overdetermined tree.

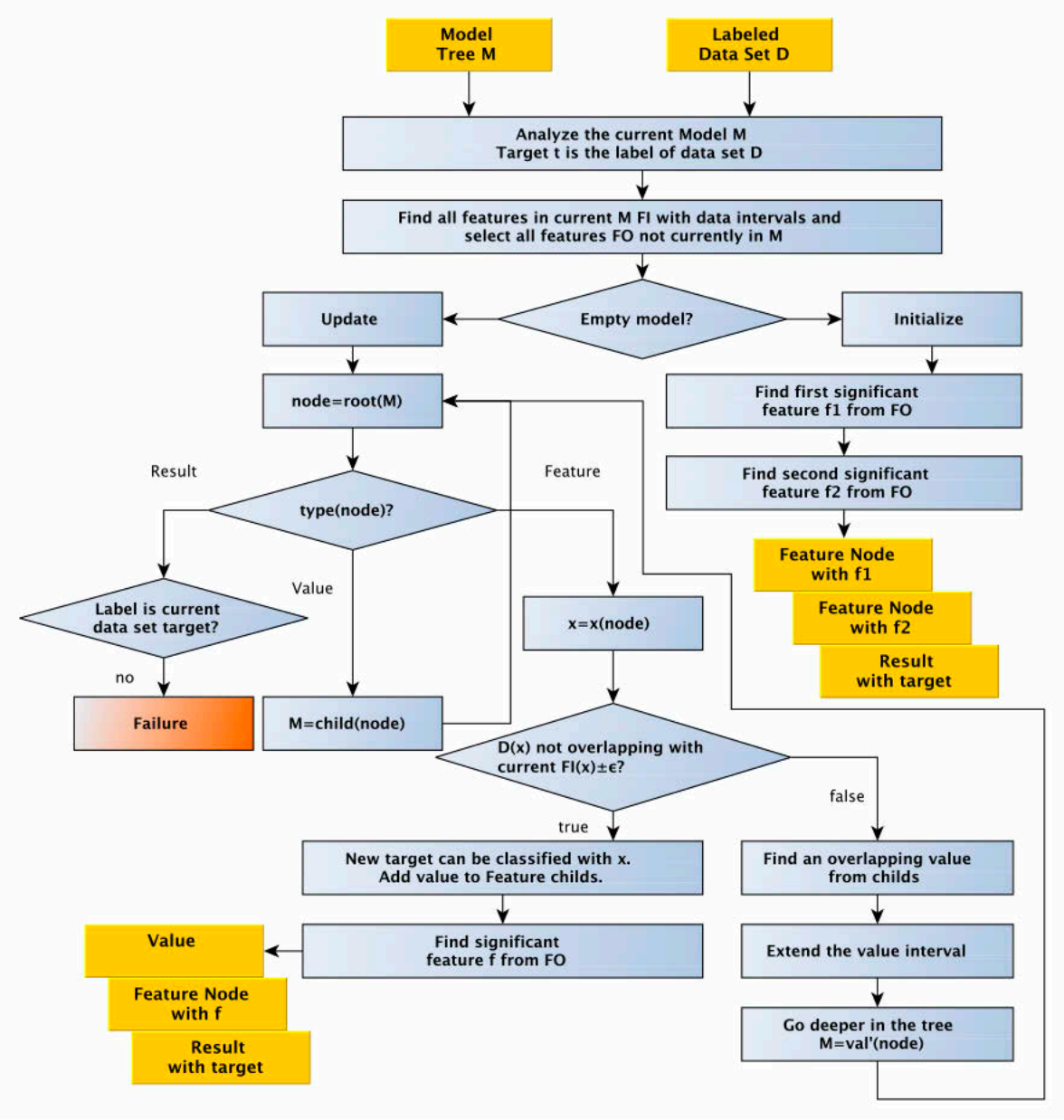

Fig. 5. *New I²DT Algorithm*

Alg. 1. *Incremental Interval Decision Tree Learner Algorithm (I^2DT) (x: feature variable, y: output target variable, $\lfloor x$: lower bound of variable x, $\lceil x$: upper bound, v(x): value of x)*

```
type node = Result(t) |
            Feature(x,vals:node []) |
            Value(v,child:node)
type of dataset = (x1,x2,x3,..,y) []

function learnIncr(model,datasets,features,target,options) {
  y = target
  ε = options[ε]
  Analyze the current model tree
  featuresM = { (xi,⌊xi,⌈xi) | Feature(xi) ∈ model }

  features' = { f ∈ features | f ∉ featuresM };
  Create a root node
  function init(model,set) {
    f1 = significantFeature(set,features)
    features' := { f | f ∈ features' ∧ f ≠ f1 }
    f2 = significantFeature(set,features')
    features' := { f | f ∈ features' ∧ f ≠ f2 }
    featuresM := featuresM ∪ {(f1,v(f1)-ε,v(f1)+ε),(f2,v(f1)-ε,v(f2)+ε)}
    model =
     Feature(f1,[Value([v(f1)-ε,v(f1)+ε],
       Feature(f2,[Value([v(f1)-ε,v(f2)+ε],
         Result(v(y))]]
  }
  Iterate and update tree
  function update(node,set,feature) {
    when node is Result:
     if t(node) ≠ y(set) then Failure!
    when node is Feature:
     x = x(Feature)
     if set[x] not in ⌊featuresM(x)±ε⌉ then
       New target can be classified
       f1 := significantFeature(set,features')
       featuresM := featuresM ∪ {(f1,v(f1)-ε,v(f1)+ε)}
       Extend interval
       ⌊featuresM(x) := min(⌊featuresM(x), set[x]-ε)
       ⌈featuresM(x) := max(⌈featuresM(x), set[x]+ε)
       leaf =
         Value([set[x]-ε,set[x]+ε],
          Feature(f1,[Value([v(f1)-ε,v(f1)+ε],
           Result(v(y))]]
       add leaf to vals(Feature)
     else
       Go deeper in the tree, find an
       overlapping value and extend the interval
       with val ∈ vals(Feature) | ⌊v(val)⌉ overlap ⌊set[x]±ε⌉ do
         Extend interval
         ⌊v(val) := min(⌊v(val), set[x]-ε)
```

```
        ⌈v(val) := max(⌈v(val), set[x]+ε)
        update(val,set,x(node))
    when node is Value:
      update(child(node),set)
  }
  Apply all new training sets
  ∀ set ∈ datasets do:
    if model = ∅ then init(model,set)
    else update(model,set)

  return model
}
```

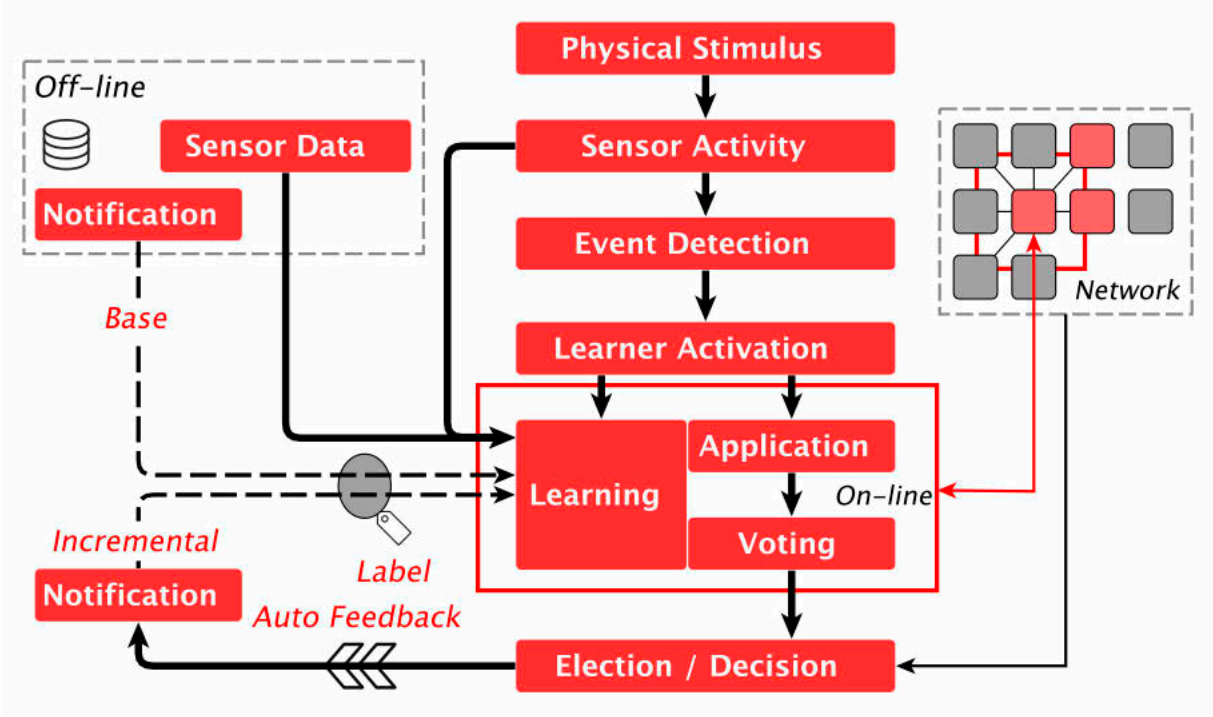

Fig. 6. *The principle concept: Global knowledge based on majority decision is back-propagated to local learning instances to update the learned model.*

A decision tree (DT) consists of *Feature* and *Value* nodes, and *Result* leaves. First the current DT (model) is analyzed. All feature variables x and their value bounds found in the tree in `Feature(x,vals)` nodes are collected in the `featureM` set (a list of (x, lower bound, upper bound) tuples). New feature variables added to the tree should not be contained in this set. Now each new training set, consisting of data variables x and a target result variable y, is applied to the DT. If the DT is empty, and initial *Feature* node is created. using the most significant data variable As mentioned before, another *Feature* node is added for future DT updates. If the DT is not empty, the tree is iterated from the root node with the current training data until a feature variable separation is found, i.e., a new `Value(x,..)` node can be added with an non-overlapping 2ε-interval around the current value of the variable x. If the current 2ε-interval of a value of a feature variable overlaps an existing *Value* interval, this interval is expanded with the new variable interval and the DT is entered one level deeper. If the last Result leave is found and its value is not equal to the current target variable value, the update has failed.

This simple learning algorithm has a computational complexity of $\Theta(N)$ with respect to the number of training sets N to be added, and for each data set $\Theta(\log n)$ with respect to the number of nodes n in the current tree, assuming a balanced tree. The incremental learning is significantly simpler than the entropy-based feature selection IDT algorithm.

The run-time behaviour flow of the decentralized learning system is shown in Fig. **6**. A physical stimulus results in sensor activity at multiple positions in the sensor network that is analyzed by an event recognition algorithm. If a sensor node detects a local sensor event the local learner is activated. It performs either a learning of a new training set or applies the learned model with the current data set consisting of ROI data. In the case of a prediction, it will make a vote. After the election of all votes, the result is back propagated to the network and all learners can update their model with the current data set as a new training set.

THE MULTI-AGENT SYSTEM

The real sensor network consists of seismic stations that are transformed in a two-dimensional mesh-grid, placing station nodes based on a spatial neighbourhood relation to other stations, shown in Fig. **3**. The sensor network is populated with different mobile and immobile 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. Agents interact with each other by exchanging tuples via the tuple space and by sending of signals. All agents were implemented in *AgentJS*, and allocate about 1k-10k Bytes for the entire process including code and data. Some agents, e.g., the explorer, is partitioned in a main and smaller sub-classes used only for the creation of child agents.

Node Agent

The immobile node agent performs local sensor acquisition and pre-processing:

- Noise filtering;
- Validation of sensor integrity;
- Sensor fusion;
- Down sampling of sensor data and storing data in tuple space;
- ROI monitoring by sending out explorer agents (optional) performing a correlated cluster recognition;
- Energy Management;
- Activation of learner agent.

Explorer Agent

The explorer agent is used to collect sensor data in a ROI by performing a divide-and-conquer approach with child agent forking. This approach provides robustness against communication failures and weak node connectivity.

- On the starting node, initially a set of explorer agents is sent out from to all possible directions.
- Each explorer agent migrates to the neighbour node, collect and processes local sensor data, and sends out further child explorer agents to all neighbourhood nodes except the previous node.
- All child explorer agents collect sensor data, divide themselves until the boundary of the ROI is reached, and return to the parent agent and deliver the collected sensor data. The approach is redundant, and hence multiple explorer agents can visit one node. The first explorer on a new node stores a marking in the tuple space, notifying other explorers to return immediately.
- After all explorer agents returned, the collected sensor matrix is delivered in the tuple space.

Learner Agent

The learner agent has the goal to learn a local classification model with data from a ROI. It can operate in two modes: (1) Learning (2) Classification (Application).

- The learner sleeps after start-up until it is woken up by the node agent. The synchronization takes place via the tuple space by consuming a `TODO` tuple.
- Learning mode: The `TODO` tuple contains the target variable value. The learner sends out explorer agents to collect the sensor data (three sensors *HHE*, *HHN*, *HHZ*) in the ROI.
- If it uses the IDT algorithm, the learner stores the data set in its own data base. The current training data base is used to learn the model.
- It it uses the I^2DT algorithm, the learner only updates the current model and discards the current training data.
- Application mode: The learner sends out the explorer agents to collect sensor data in the ROI. It uses this sample data for prediction. If the classification was successful, it will send out voter agents.

Distributor Agent

Among the local learning approach, sensor data is collected by central instances. e.g., performing model-based seismic data analysis. A distributor agents is used to deliver a set of sensor data from a ROI. The distributor agent is activated only if there was a significant sensor change in the ROI. The distributions process can be performed with different approaches:

- Peer-to-peer, i.e., the destination node is known or a path to the destination must be explored.
- Broadcasting, i.e., using divide-and-conquer with agent replication.
- Data sink driven, i.e., the distributor agent follows a path of marking (stored in the tuple space) to deliver the data along this path.

Voter Agent

The voter agent distributes a particular vote to election agents. There can be multiple election agents in the network, hence a row-column network distribution approach is used.

- The initial node agent sends out multiple voter agents to all possible directions (four directions in a mesh-grid network).
- If a distributor agent reaches a boundary of the network, it will replicate and distribute the vote in perpendicular and opposite directions until a node with an election agent is found.

Election Agent

An election agent is some kind of a central instance of the MAS.

- Collecting of votes delivered by voter agents within a time interval (after that votes are discarded)
- Back propagation of winner votes to the learner agents by sending out notification agents.

Notification Agent

The notification agents are sent out by some central instance to notify all nodes that a new training set is available with a specific target variable value, e.g., the parameters of an earthquake event (identifier, location, magnitude,..). One central instance can be the election agent that evaluate votes in application mode. The winner vote fraction is carried by notification agents to update learner agents.

- A divide-and-conquer approach with agent replication is used to broadcast the notification to all nodes in the network.

Disaster Management Agent

Although not considered in this work and currently not existing in the MAS, disaster management agents are high level central instances in the network that monitor the election results and activity in

the sensor network and plan the co-ordination of disaster management based. See **Fiedrich (2007)** for a consideration of MAS-based disaster management.

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 were used and processed by the learning MAS, retrieved from **SCEDEC (2016)**.

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

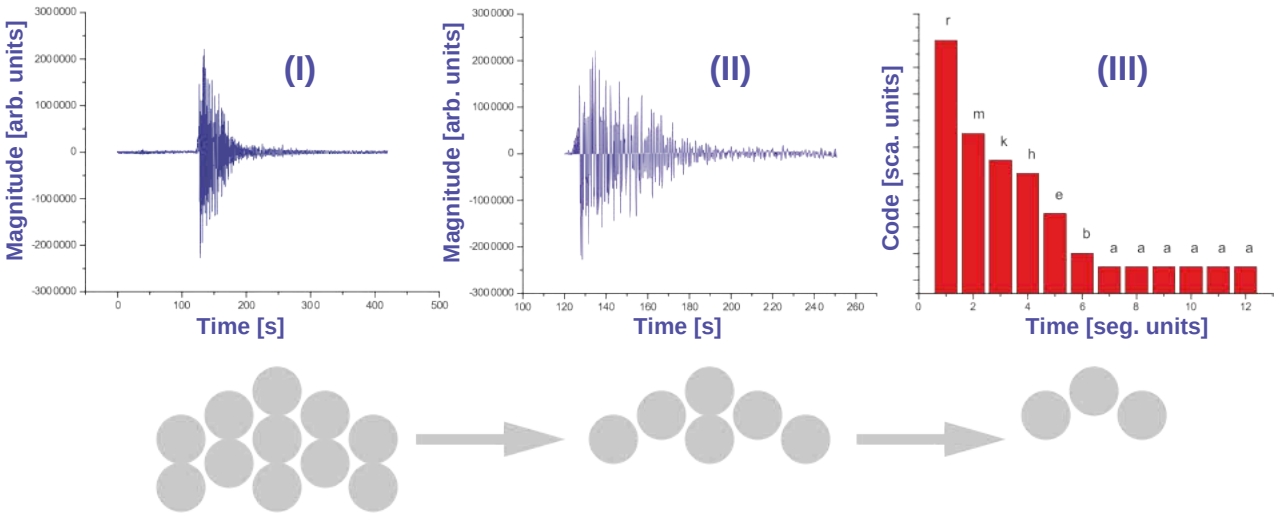

Fig. 7. *Seismic data reduction: (I) Down sampling (1:16) with absolute peak value detection, (II) ROI analysis and ROI clipping, (III) Down sampling (1:64) and scaling/normalization with 55-string coding (0,a-z,A-Z,!,*)*

Event / Date	Location	Depth	Magnitude
1999-289 /16-10-1999	34.5940, -116.2710	0.02 km	7.2
2002-246 / 03-09-2002	33.9173, -117.7758	12.92 km	4.75
2003-053 / 22-02-2003	34.3098, -116.848	1.23 km	4.9
2004-045 / 14-02-2004	35.0385, -119.1315	11.66 km	4.34
2004-167 / 15-06-2004	32.3287, -117.9175	10.0 km	4.98
2005.006 / 06-01-2005	34.1250, -117.4387	4.15 km	4.42
2005.106 / 16-04-2005	35.0272, -119.1783	10.3 km	4.59
2005.163 / 12-06-2005	33.5288 -116.5727	14.2 km	5.2

Tab. I. Selected Earthquake events [Location: °N/W, Magnitude: Richter Scale Units, Row colors: Pairs of earthquake events with similar location]

Event / Date	Location	Depth	Magnitude
2005.243 / 31-08-2005	33.1648, -115.6357	4.0 km	4.59
2005.245 / 02-09-2005	33.1598, -115.6370	9.8 km	5.11
2008.211 / 29-07-2008	33.9530, -117.7613	14.7 km	5.39
2010.094 / 04-04-2010	32.2862, -115.2953	10.0 km	7.2
2010.188 / 07-07-2010	33.4205 -116.4887	14.0 km	5.43
2013.070 / 11-03-2013	33.5025, -116.4572	13.1 km	4.7
2013.149 / 29-05-2013	34.413, -119.926	8.0 km	4.8
2014.088 / 29-03-2014	33.932, -117.917	4.8 km	5.10
2014.186 / 05-07-2014	34.280, -117.028	8.7 km	4.58

Tab. I. Selected Earthquake events [Location: °N/W, Magnitude: Richter Scale Units, Row colors: Pairs of earthquake events with similar location]

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 sampling the ROI data again with a final magnitude normalization and a 55-value string coding. The process is shown in Fig. **7**. 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 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 (implementing a GUI/Visualization layer on the top of *JAM*). It consists of multiple full operational virtual *JAM* nodes connected by virtual links enhanced with a GUI, proposed by **Bosse (2016A)**. The seismic stations of the CI network are mapped on a two-dimensional grid with spatial proximity, shown in Fig. **3**. 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.

The MAS simulation was performed with seismic data from a set of 17 different earthquake events, shown in Tab. **I**. Some of the events have similar earthquake center location, marked by colored rows.

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. Example situations of the event-stimulated temporal agent population for the event-based learning and application phases are shown in Fig. **8**. The peak agent population of the network is about 1500 agents, the lower bound is about 180 agents.

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

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 election leads to a major vote decision with a confidence value [0.0,1.0] defined by Eq. **(1)**.

$$confidence = 1.0 - \frac{2}{n} \sum_{j=1, j \neq i_{\max}}^{n} \frac{x_j}{x_{\max}} \tag{1}$$

The entire election consists of n different votes, x_j is the vote weight (or number of votes) for each fraction, and x_{max} the weight of the winner fraction. E.g., if there are two fractions with equal number of votes, the confidence is zero, because no decision can be made. Negative confidence values denote a failed classification for a specific fraction. A confidence value greater 0.5 denotes a high probability that the winner is trustful.

Classification results are shown in Fig. **9**, all based on global majority election and Monte Carlo simulation. Multiple learning runs were performed to train the network using a random sequence of different earthquake events with noisy data (complete set). During the classification (application) phase, a random sequence of noisy seismic data was applied, too. All earthquake events can be recognized with a high confidence and prediction accuracy (suing complete set). Some tests were made with incomplete training sets (last three rows in Fig. **9**) to find similar events, marked in Tab. **I**. The pair (2005.245,2005.243) could be recognized with a high confidence (with the incremental approach using weighted votes), pair (2004.186,2003.053) failed completely with both approaches.

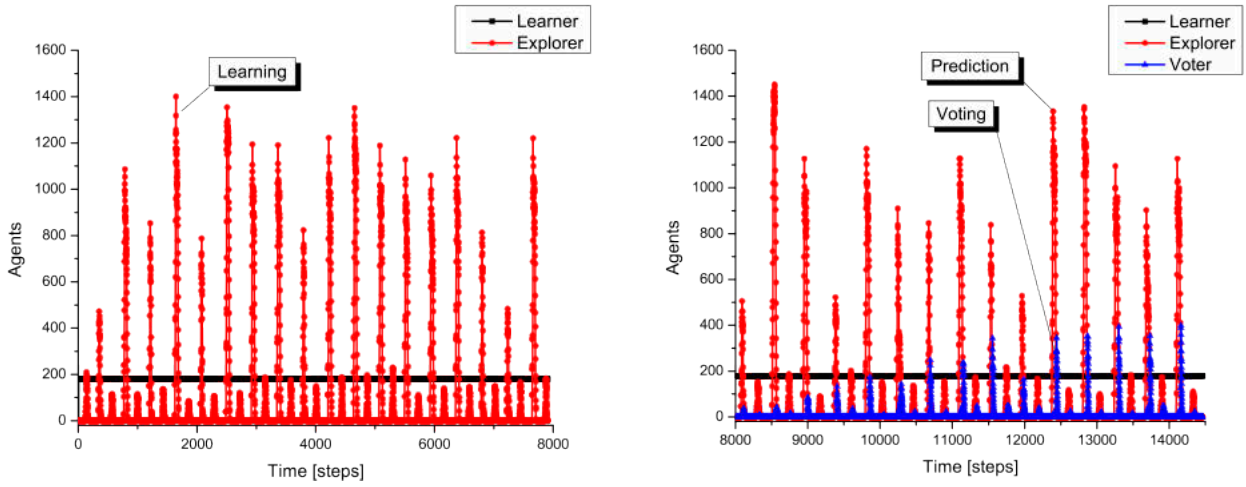

Fig. 8. *Temporal agent population with 180 seismic nodes during learning (training, left) and classification (prediction and voting, right) phases.*

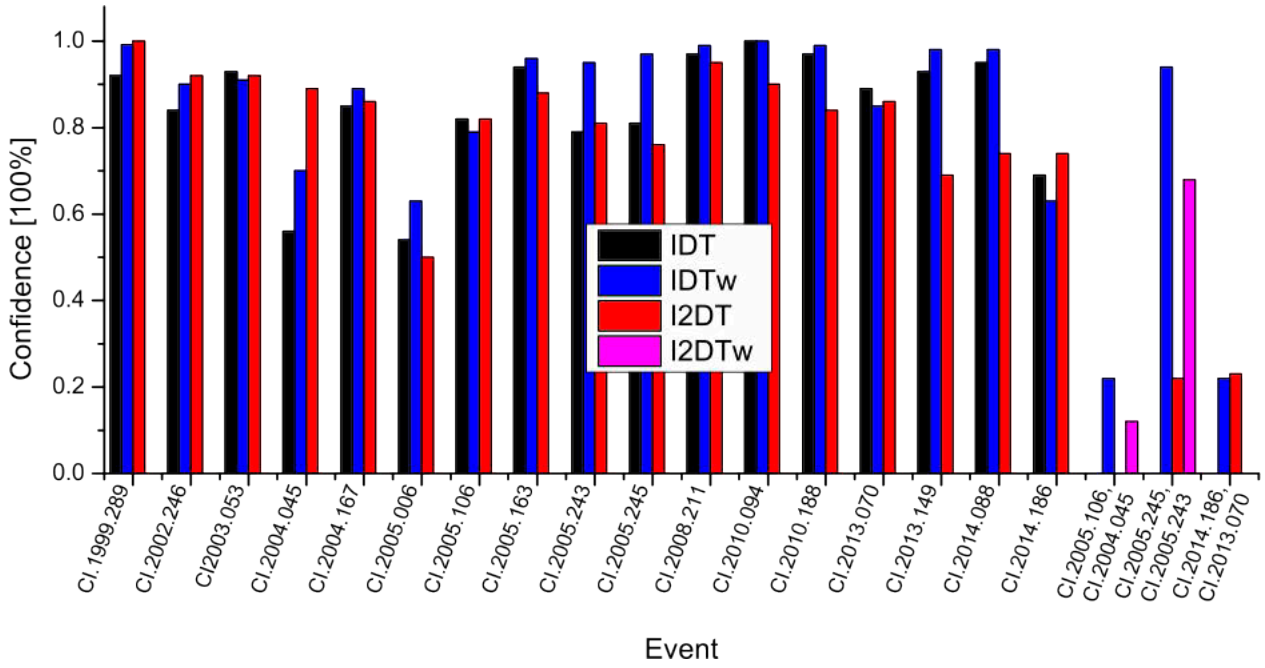

Fig. 9. *Comparison of the prediction results of the distributed IDT and incremental I²DT learner algorithms (noise:20 a.u., eps: 20 a.u., roi=3, w: weighted votes, 1: Event was not contained in training set). In all full test cases the election winner was always the correct event*

The incremental I²DT learner algorithm was about 200% faster than the non-incremental IDT algorithm with the same training data sets (accumulated computation time observed in the entire network with all participating learners). The accuracy of the incremental learner is comparable to the IDT learner. The confidence of the IDT election result can be slightly improved if weighted votes are processes, i.e., the local accumulated sensor data is used as a weight, dominating the election by nodes with a high stimulus. But the I²DT does not profit from vote weighting.

The transition from learning to prediction is seamless and is based on the node/learner experience (learned events). Furthermore, after an event is elected by the majority decision, 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). The usage of I²DT lowers the entire communication costs about 30% compared with the IDT approach (due to the data base).

OUTLOOK: UBIQUITOUS DEVICES AS AN EXTENSION

In **Kong (2016)**, smart phones were successfully used to enhance the earthquake prediction by extending the seismic database with sensor data from mobile devices. **Pournaras (2015)** proposed an open participatory platform for privacy-preserving social mining (*Nervousnet*, Planetary Nervous System) was introduced, i.e., basically a virtualization of sensors that can profit from the proposed agent framework and distributed learning. The new *JVM* engine executing *JAM* is well suited for low-resource systems, e.g., deployed in the *Nervousnet* environment. 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). The *JAM-JVM* architecture is well suited for low-resource mobile devices, and *JAMLIB* can be embedded in any application program using *Webview* or *UIWebView* frameworks, easing App. development and deployment on Android and iOS devices.

CONCLUSIONS

Distributed agent.-based learning with agents basing on spatially limited local 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, with a back-propagation of global knowledge to local learner instances.

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*, and by using the JS engine *JVM* satisfying low-resource requirements, too. A broad range of node connectivity and host platforms are supported. JAM is therefore a suitable enabling technology for the IoT and Clouds. The new incremental and distributed I^2DT learner outperforms the non-incremental IDT learner, evaluated with a large-scale seismic measuring network and historic earthquake data. A typical DT requires less 1k Byte memory, and the incremental learning eliminates the storage of historic data sets. With a training base, all earthquake events from this base could be recognized with a high confidence, and some selected unknown events could be matched with similar historic events. This ability is a prerequisite for disaster monitoring to fastly classify a large-scale event. The entire perdition (response) time of the MAS is below 10 seconds after an event occurred.

The self-* and distributed MAS was composed of multiple different specialized agents. 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. 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 strongly heterogeneous environments.

REFERENCES

Alarifi, A., Alarifi, N., & Al-Humidan, S. (2012) *Earthquakes magnitude predication using artificial neural network in northern Red Sea area*, Journal of King Saud University – Science, vol. 24, pp. 301-313, .

Bellifemine, F. & Caire, G. (2007) *Developing Multi-Agent Systems with JADE*. John Wiley & Sons, Ltd, .

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

Bosse, S., & Lechleiter, A. (2015) *A hybrid approach for Structural Monitoring with self-organizing multi-agent systems and inverse numerical methods in material-embedded sensor net-*

works, Mechatronics, , doi:10.1016/j.mechatronics.2015.08.005

Bosse, S. (2015) *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.

Bosse, S. (2016) 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

Bosse, S. (2016) *Mobile Multi-Agent Systems for the Internet-of-Things and Clouds using the JavaScript Agent Machine Platform and Machine Learning as a Service*, in The IEEE 4th International Conference on Future Internet of Things and Cloud, 22-24 August 2016, Vienna, Austria, 2016.

Bosse, S. (2016) *Distributed Machine Learning with Self-organizing Mobile Agents for Earthquake Monitoring*, in 2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS*W) , SASO Conference, DSS, 12 September 2016, Augsburg, Germany, 2016.

Caridi, M., & Sianesi, A. (2000) *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, .

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

Fiedrich, F., & Burghardt, P. (2007) *Agent-based systems for disaster management*, Communications of the ACM - Emergency response information systems: emerging trends and technologies CACM, vol. 50, no. 3, pp. 41-42, .

Gavrin, E. , Lee, S.-J., Ayrapetyan, R., & Shitov, A. (2015) *Ultra lightweight JavaScript engine for internet of things*, in SPLASH Companion 2015 Companion Proceedings of the 2015 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity, 2015, pp. 19-20.

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

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

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

Lehmhus, D., Wuest, T., Wellsandt, S., Bosse, S., Kaihara, T., Thoben, K.-D., & Busse, M. (2015) *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.

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

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

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

Pournaras, E., Moise, I., Helbing, D. (2015) *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).

SCEDEC (2016), http://scedc.caltech.edu/research-tools/eewtesting.html