DOI:10.1016/j.procs.2015.08.312

6th International Conference on Emerging Ubiquitous Systems and Pervasive Networks, EUSPN-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

Stefan Bosse*

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

Abstract A novel and unified approach for reliable distributed and parallel computing using mobile agents is introduced. The agents can be deployed in large scale and hierarchical network environments crossing barriers transparently. The networks can consist of high- and low-resource nodes ranging from generic computers to microchips, and the supported network classes range from body area networks to the Internet including any kind of sensor and ambient network. Agents are represented by mobile program code that can be modified at run-time. The presented approach enables the development of sensor clouds and smart systems of the future integrated in daily use computing environments and the Internet. Agents can migrate between different hardware and software platforms by migrating the program code of the agent, embedding the state and the data of an agent, too. The entire information exchange and coordination of agents with other agents and the environment is performed by using a tuple space database. Beside architecture specific hardware and software implementations of the agent processing platform, there is a JavaScript (JS) implementation layered on the top of a distributed management layer. The JS platform enables the integration of Multi-agent Systems (MAS) in Internet server and application environments (e.g., WEB browser). Agents can migrate transparently between hardware-level sensor networks and WEB browser applications or network servers and vice versa without any transformation required.

Keywords: Sensor Networks, Cloud Computing, Mobile Agents, Heterogeneous Networks, Embedded Systems, Agent Processing Platform

1. Introduction

Trends emerging in engineering and micro-system applications such as the development of sensorial materials composed of high-density miniaturized active sensors pushes the Internet-of-Things, requiring distributed autonomous computing in large-scale heterogeneous networks consisting not only of miniaturized low-power smart sensors embedded in technical structures. Large scale sensor networks with hundreds and thousands of sensor nodes integrated in cloud-based service environments require data processing concepts far beyond the traditional centralized approaches. Multi-Agent systems (MAS) can be used to implement smart and optimized sensor data processing in these distributed sensor networks **[1][7]** and information distribution across network domains. The MAS paradigm offers a unified data processing and communication model suitable to be employed, e.g., in the design, the manufacturing, and the logistics of products, and the products themselves (e.g., robots).

Agents are already deployed successfully for scheduling tasks in production and manufacturing processes **[5]**, and newer trends poses the suitability of distributed agent-based systems for the control of manufacturing processes **[6]**, facing not only manufacturing, but 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 successfully deployed in sensing applications, e.g., structural load and health monitoring, with a partition in off- and online computations **[3]**. Distributed data mining and Map-Reduce algorithms are well suited for self-organizing MAS. Cloud-based computing with MAS, as a base for cloud-based manufacturing, means the virtualization of resources, i.e., storage, processing platforms, sensing data or generic information.

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 **[6]**. Currently, many traditional processing platforms cannot yet handle big number of agents with the robustness and efficiency required by the industry **[6]**. 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 sensor networks in generic computer networks and 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 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-connectivity 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 the traditionally separated sensing, aggregation, and application layers, offering a more unified design approach and more generic and unified architectures. Smart systems glue software and hardware components to an extended operational unit.

Growing system complexity requires an increase in the autonomy feature of distributed data processing systems, addressed, e.g., by the deployment of mobile Multi-agent systems carrying out information processing and context-based distribution. Self-organizing systems are one major approach to solve complex tasks by decomposing them into smaller and simpler tasks performed by a large group of individuals **[8]**.

A cloud in terms of data processing and computation is characterized by and composed of: A parallel and distributed system architecture, a collection of interconnected virtualized computing entities that are dynamically provisioned, a unified computing environment and unified computing resources based on a service-level architecture, and a dynamic reconfiguration capability of the virtualized resources (computing, storage, connectivity and networks). Cloud-based design and manufacturing is composed of knowledge management, collaborative design, and distributed manufacturing. Adaptive design and manufacturing enhanced with perception delivered by the products incorporates finally the products in the cloud-based design and manufacturing process.

The central approach in this work focuses on mobile agents and the ability to support mobile code embedding the agent behaviour, the agent data, the agent configuration, and the current agent control state, finally encapsulated in code frames. This agent-specific mobile program code can be executed on a variety of different platforms ranging from microchip (System-on-Chip) hardware to generic software levels, now including a *JavaScript* and WEB browser capable platform implementation with a distributed co-ordination / management layer and a broker service for IP client-side only applications. This extends the scope and visibility of sensing devices and sensor networks to the Internet domain and enables cloud computing in strong heterogeneous networks.

In the following sections the deployment and processing of mobile agents in heterogeneous large-scale network environments is discussed from a top-down perspective giving an outline of the architecture layers.

2. The Activity-based Agent Behaviour and the AAPL Programming Model

The implementation of mobile multi-agent systems with a particular focus on resource constrained embedded systems (microchip level) is a complex design challenge. High-level agent programming and behaviour modelling languages can aid to solve this design issue. Activity-based agent models can aid to carry out multi-agent systems on hardware platforms **[1]**. The behaviour of an 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 Activity-Graph Agent Programming Language *AAPL* **[1]** was designed to offer the modelling of the agent behaviour on programming level, defining activities with procedural statements and transitions between activities with conditional expressions (predicates) based on agent data. Though the imperative programming model is quite simple and closer to a traditional programming language it can be used as a common source and intermediate representation for different agent processing platform implementations (hardware, software, simulation) by using a high-level synthesis approach.

There is a multi-agent system (MAS) consisting of a set of individual agents $\{A_1, A_2, ..\}$. There is a set of different agent behaviours, called classes $\mathcal{C}=\{AC_1, AC_2,..\}$. An agent belongs to one class. In a specific situation an agent A_i is bound to and processed on a network node N_l (e.g. microchip, computer, virtual simulation node) at a unique spatial location l. There is a set of different nodes $N=\{N_1, N_2, ..\}$ arranged in networks with peer-to-peer neighbour connectivity (e.g., a two-dimensional grid). Each node is capable to process a number of agents $n_i(AC_i)$ belonging to one agent behaviour class AC_i, and supporting at least a subset of $\mathcal{C}' \subseteq \mathcal{C}$. An agent (or at least its state) can migrate to a neighbour node where it continues working.

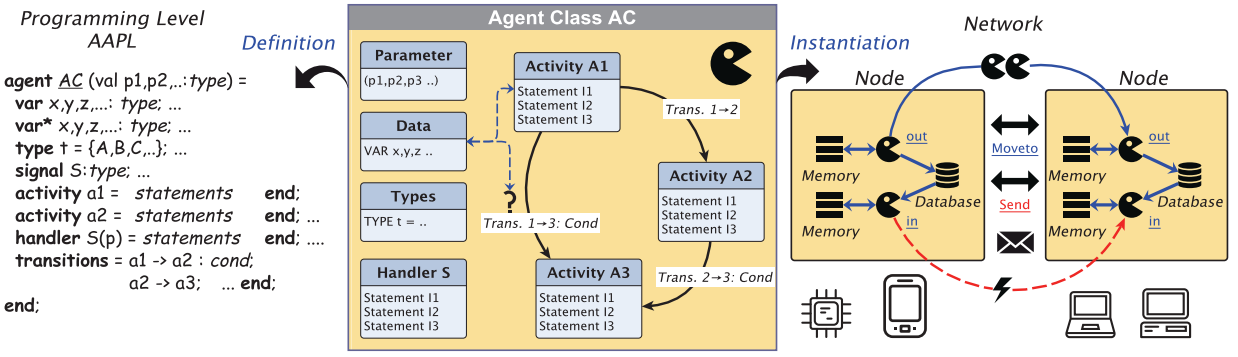

Fig. 1. Agent behaviour programming level with activities and transitions (*AAPL*, left); agent class model and activity-transition graphs (middle); agent instantiation, processing, and agent interaction on the network node level (right).

Therefore, the agent behaviour and the action modifying the environment is encapsulated in agent classes, with activities $\{a_1,a_2,..\}$ representing the control state of the agent reasoning engine, and conditional transitions $\{t_1,t_2,..\}$ 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.

The activity-graph based agent model is attractive due to the proximity to the finite-state machine model, which simplifies the hardware 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. **1**. 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.

Instantiation. The *AAPL programming language* (detailed description in **[1]** and **[2]**) offers statements for parameterized agent instantiation, like the creation of new agents and the forking of child agents, using the `new(`*args*`)` and `fork(`*args*`)` statements, respectively.

Modification. There are statements enabling the modification of the agent behaviour at run-time applied to the ATG. ATG transitions can be added, deleted, or replaced by the `transition`^{op}`(`a_i`,`a_j`,`*cond*`)` operation. New classes can be composed of activities by using these and `activity`^{op}`(`a_1`,`a_2`,..)` statements, with *op*={+,-,*}.

Mobility. Agent mobility is offered by migration using the `moveto(`*dst*`)` statement specifying a cartesian direction (distance vector) or a destination node identifier. Only migration to a neighbour node is provided, though neighbourhood can be based on physical or logical connectivity (e.g., in the Internet domain).

Interaction. Agent interaction is offered by coordinated 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*`)`, `mark(`*timeout*`,`*T*`)` primitives (*T*: n-dimensional tuple with actual parameters, *TP*: n-dimensional tuple with actual and formal patterns). The `in`/`rd`/`rm` operations extract tuples from the database based on pattern matching. The **out** operation stores tuples in the database (generating persistent tuples), whereby the `mark` operation assigns a time-out flag to the tuple that enables the destruction of the tuple by a garbage collector if the time-out is reached (generating temporary tuples). A Remote-Procedure Call operation is supported by the `eval(`*TP*`)` primitive that stores a partially evaluated tuple in the database that is consumed by a service agent, processing it and returning the fully evaluated tuple to the database again, which is finally passed to the original client evaluation call. A signal *sig* can be sent to an agent with the identifier *id* by using the `send(`*id*`,`*sig*`,`arg`)` statement. Signals are mostly used in parent-child agent coordination.

3. The Network Architecture

The agent processing platform architecture itself does not require any distributed coordination and management layer to be operational. But in the context of basically loosely coupled and highly dynamic network environments like the Internet coordination, logical (virtual) connectivity management, and structuring is required to provide meaningful environments for agent mobility in terms of tasks and goals to be fulfilled by agents. The basic global and distributed coordination and management for the agent processing and mobility is provided by a capability-based RPC layer with some dedicated services like a file and directory naming service, discussed in the following subsections. The distributed management and coordination layer bases on previous work discussed in **[9]**.

A global distributed coordination layer including a file and naming service is not required for the processing of mobile agents in low-distance and directly connected networks, e.g., considering sensor networks embedded in technical structures or deployed in wearable computing use cases. In these networks the physical network topology represents the communication network topology, and the agent mobility relies on the provided (and approximately static) neighbourhood connectivity of network nodes.

3.1. Capability based RPC

Object-orientated Remote Procedure Calls (RPC) 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 **[4]**. 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 **[4]** for details). All parts are merged in a capability structure `[srvport] obj(rights)[protport]`.

3.2. AFS: Atomic File System Service

The Atomic File System Server (*AFS*) provides a unified and reliable file system storage suitable for the deployment in unreliable environments and is independent form lower level storage capabilities. Files are associated with a capability. The capability port is given by the server, and the capability object number identifies the file uniquely. The protected rights field of the capability determines the authorized operations to be applied to a file or the file system. A file is stored always in a contiguous block cluster of the file system, avoiding a linked free and used block management that offers a low-resource and low-overhead file system with basic real-time feature capabilities. A committed file is immutable (locked, read-only mode) and occupies only one internal node (i-node). Modification of locked files require an uncommitted (unlocked) copy of the original immutable file. Though this approach seems to be inefficient for post modifications of files, it avoids the requirement of file system logging required for a fast crash recovery. Here, after a crash only uncommitted files (occupying only one i-node) must be cleared. The *AFS* is used in this work mainly for storing agent program code and persistent tuple space data. The simplicity of the *AFS* enables the implementation in JavaScript and the embedding in browser applications, discussed in the next section.

The set of operations embraces the reading, modification, commitment, creation, and destruction of files. Each file object has a limited lifetime that is decremented periodically by a garbage collector that removes unused entries. Therefore, there is a touch and age operation modifying the lifetime of a file. Only the explicit commitment of a file makes the file persistent. Reading data from and writing data to storage devices is performed through a cache module, speeding up reading and modifying of file data and i-nodes.

3.3. DNS: Directory and Name Mapping Service

The Directory and Name Server (*DNS*) provides 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. A capability set binds multiple capabilities associated with the same semantic object, e.g., a file that is replicated on multiple file servers. A capability set marks one object capability as the current and reachable object, though this may change any time. A directory is associated with an internal node and the content table (the rows). The directory content is stored in an *AFS* file. Redundancy is offered by the capability sets themselves (replication of objects) and by the *DNS* using two file servers for storing directories in two-copy mode (replication of directories). The immutability of *AFS* files immediately qualifies the immutability of directories, enabling a robust directory system with a fast an stable recovery after a server crash. The simplicity of the *DNS* enables the implementation in *JavaScript* and the embedding in browser applications, too, discussed in the next section.

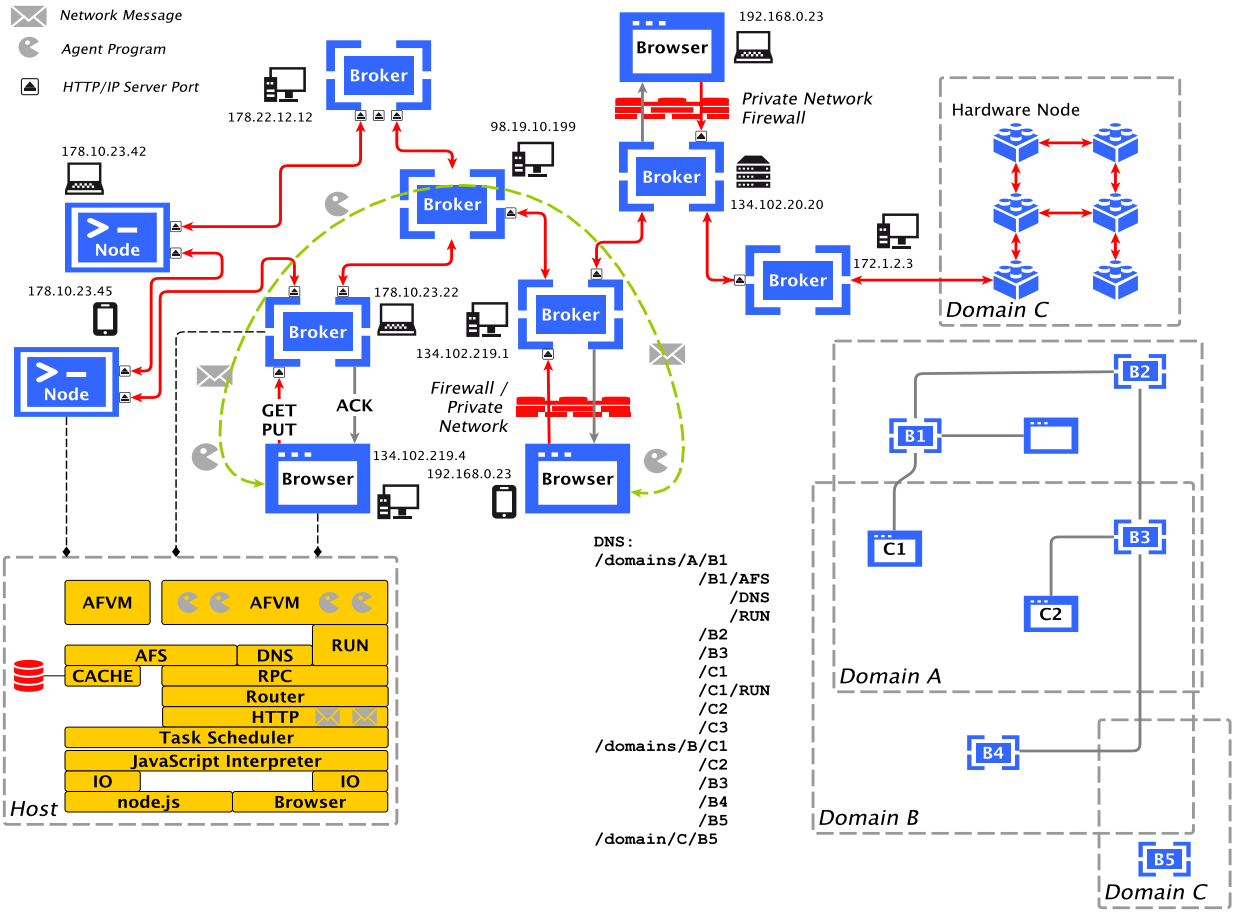

Fig. 2. Different agent processing platforms and nodes are connected in Inter-, Intranet, and dedicated Sensor network domains including hardware nodes (embedded and microchip level platform implementations). Broker servers are used to connect IP client-only applications (e.g., browser applications or nodes in private networks).

The set of operations embraces the reading, modification, creation, and destruction of directories. Each directory has a limited lifetime that is decremented periodically by a garbage collector that removes unused entries that are not linked anymore. Therefore, there is a touch and age operation modifying the lifetime of a directory.

3.4. Broker Service

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, shown in Fig. **2**. 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.

But the deployment of one central broker server introduces a single-point-of-failure and is limiting the communication bandwidth and the scaling capability significantly. To overcome these limitations, a hierarchical broker server network is used. Each broker in this broker graph can be the root of a sub-graph and can be a service end-point (i.e., providing directory and name services), a router between clients and other broker servers, and an interface bridge to a non-IP based network, i.e., a sensor network. A broker is just an application program capable of running on any com-

puter visible in a network domain (globally in the Internet, locally in an Intranet). Each node in the network act always as a service end-point and as a logical router, regardless of the IP server- or client-side visibility.

4. The Modular (J)AVM Platform Architecture

The JavaScript Agent Processing Platform (*JAVM*) is highly modular, shown in Fig. **3**, consisting of various modules. Basically it consists of the agent code processes (Agent Forth Virtual Machine, *AFVM*, discussed later), an agent manager (*AMAN*) responsible for agent processing control, migration, and interaction, some kind of communication layer (the bare-bone NET module or an OS layer), and optional a distributed operating and coordination layer consisting of the file- and naming services including the previously introduced RPC, implemented in *JavaScript* (*JS*) and executed either using the *node.js* VM or a *JS* capable WWW browser application. The *JS* platform requires a task scheduler to implement synchronization of parallel tasks. At least one broker server is required in Intra- and Internet domains for connecting pure client-side applications (WWW browser). The agent processing platform is implemented in hardware (SoC design, pure digital logic), in software (standalone *C*/*OCAML*), and in JavaScript. All platform implementations are compatible on communication, operational, and execution level. Platforms that should be visible in Intra- and Internet domains and that are connected indirectly require the RPC, a message Router (used for inter-node server-client communication, too), and *HTTP* connection modules to establish at least agent migration.

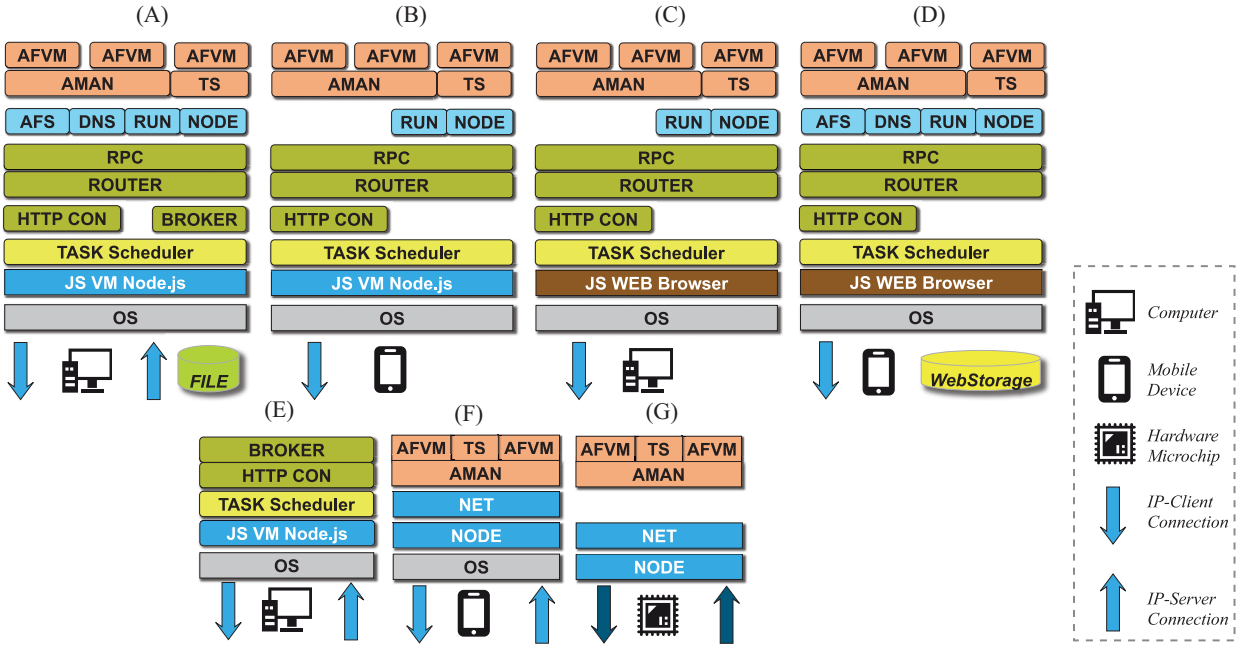

Fig. 3. The modular host platform architecture: (A) Full Server-side *JavaScript* (*JS*) Implementation with File and Naming Service (B) Client-side *JS* Implementation (C) Client-side Browser *JS* Implementation (D) Client-side Browse *JS* Implementation with File- and Naming Services using *WebStorage* (E) Broker-only Node (F) Native Software Implementation of the *AFVM* (G) Native Hardware Implementation of the *AFVM*

5. The Agent Processor Architecture

An agent consists of a behaviour and a state. The behaviour is given by the previously introduced ATG, and its state is given by the content of the body variables, the configuration, an identifier, and the control state (current activity). In this work the entire agent behaviour, the agent data, and the configuration and control state is encapsulated in program code organized in frames, shown on the left side of Fig. **4**. A code frame can be modified by the agent itself or by the agent manager, responsible for the agent process control and the migration of the program code.

The virtual machine (*AFVM*, discussed in depth in **[2]**) executing tasks bases on a traditional *FORTH* processor architecture and an extended zero-operand word instruction set (*αFORTH*/*AFL*). Most instructions operate directly on the data stack ***DS*** and the control return stack ***RS***. A code segment ***CS*** stores the program code with embedded data. The program is mainly composed of words (stack functions). A word is executed by transferring the program

control to the entry point in the *CS*; arguments and computation results are passed only by the stack(s). There are multiple virtual machines with each attached to (private) stack and code segments. There is one global code segment *CCS* storing global available functions and code templates which can be accessed by all programs. A dictionary is used to resolve *CCS* code addresses of global functions and templates. The program code frame of an agent consists basically of four parts: 1. A look-up table and embedded agent body variable definitions; 2. Word definitions defining agent activities and signal handlers (procedures without arguments and return values) and generic functions; 3. Bootstrap instructions which are responsible to setup the agent in a new environment (i.e., after migration or on first run); 4. The transition table calling activity words and branching to succeeding activity transition rows depending on the evaluation of conditional computations with private data (variables). The transition table section can be modified by the agent by using special instructions. Code morphing can be applied to the currently executed code frame or to any other code frame of the VM. The agent high-level behaviour specified in *AAPL* can be compiled directly to *AFL* and finally to a machine subset *AML*.

A task of an agent program is assigned to a token holding the task identifier of the agent program to be executed. The token is stored in a queue and consumed by the virtual machine from the queue. After a (top-level) word was executed, leaving an empty data and return stack, the token is either passed back to the processing queue or to another queue (e.g., of the agent manager), enabling self-scheduling of different agent processes, shown in Fig. **4**.

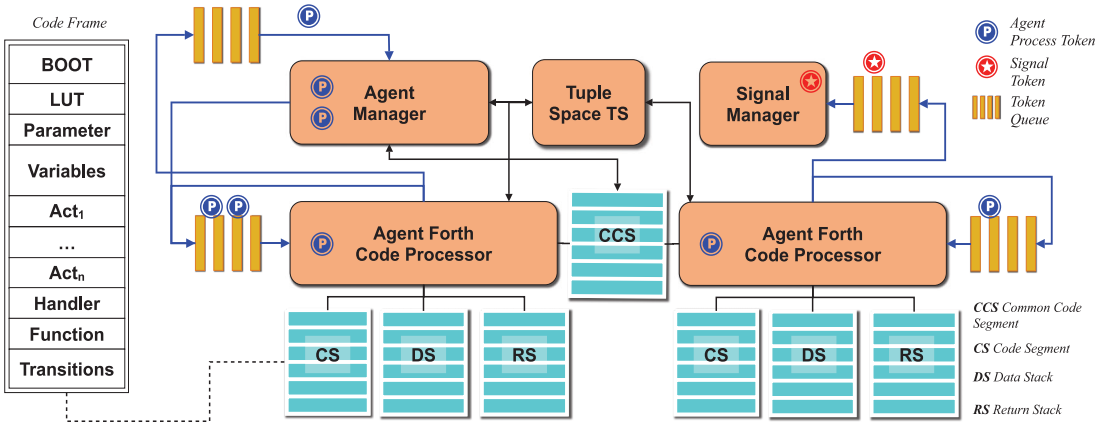

Fig. 4. (Left) Code Frame Layout (Right) The Agent Forth Virtual Machine Architecture with a token-based agent processing.

6. Big Use-Case: Cloud Based Adaptive Manufacturing and Robots as Products in a Closed-Design-Loop

This section outlines a big application use-case for the introduced agent processing platform environment deployed in additive and adaptive manufacturing processes based on a closed-loop sensor processing approach with data mining concepts combined with Internet-of-thing architectures. Additive and adaptive cloud-based design and manufacturing are attractive in the field of robotics, not only limited to industrial production robotics, mainly targeting service robots and semi-autonomous carrier robots. In cloud-based manufacturing, the consumer of the products is integrated in the cloud-based manufacturing process, directly involved in the manufacturing process using distributed cloud computing and distributed storage solutions.

Robots as products can be considered as active, mobile, and autonomous data processing units that are commonly already connected to computer networks and infrastructures. Robots use inherent sensing capabilities for their control and task satisfaction, commonly using integrated sensing networks with sensor pre-processing, deriving some inner state of the robot, e.g., mechanical loads applied to structures of the robot or operational parameters like motor power and temperature. The availability of the inner perception information of robots enable the estimation of working and health conditions initially not fully considered at design time. The perception of the products delivering operational feedback to the current design and manufacturing process leads to a closed-loop evolving, shown in Fig. **5**. This evolutionary process adapts the product design, e.g., the mechanical construction, for future product manufacturing processes based on a back propagation of the perception information (i.e., recorded load histories, working and health conditions of the product) collected by living systems at run-time. The currently deployed and running series of the

product enhances future series, but not in the traditional coarse-grained discrete series iteration. A robot consists of a broad range of parts, most of them are critical for system failures. The most prominent failures are related to mechanical and electro-mechanical components, which are caused by overload conditions at run-time under real conditions not to be considered or unknown at initial design time.

The integration of robots as products and their condition monitoring in a closed-loop design and manufacturing process is a challenge and introduces distributed computing and data distribution in strong heterogeneous processing and network environments. The mobile agents representing mobile computational processes can migrate in the Internet domain as well in sensor networks part of the robot.

The agent processing platforms introduced in this work can be deployed in those massive heterogeneous network environments, ranging form single microchip up to WEB *JavaScript* implementations, all being fully compatible on operational and interface level, and hence agents can migrate between these different platforms.

Traditional closed-loop processes request data from sources (products, robots) by using continuos request-reply message streams. This approach leads to a significant large amount of data and communication activity in large-scale networks. Event-based sensor data and information distribution from the sources of sensing events to sinks performed by agents and triggered by the data sources (the robots) themselves, can improve and reduce the allocation of computational, storage, and communication resources significantly.

Agent Classes. The entire MAS society is composed of and instantiated from different agent classes that satisfy different sub-goals and reflect the sensing-aggregation-application layer model: event-based sensor acquisition including sensor fusion (Sensing), aggregation and distribution of data, pre-processing of data and information mapping, search of information sources and sinks, information delivery to databases, delivery of sensing, design, and manufacturing information, propagation of new design data to and notification of manufacturing processes, notification of designer, end users, update of models and design parameters. Most of the agents can be encapsulated and transferred in code frames with a size lower than 4kB, which additionally depends on the data payload they carry.

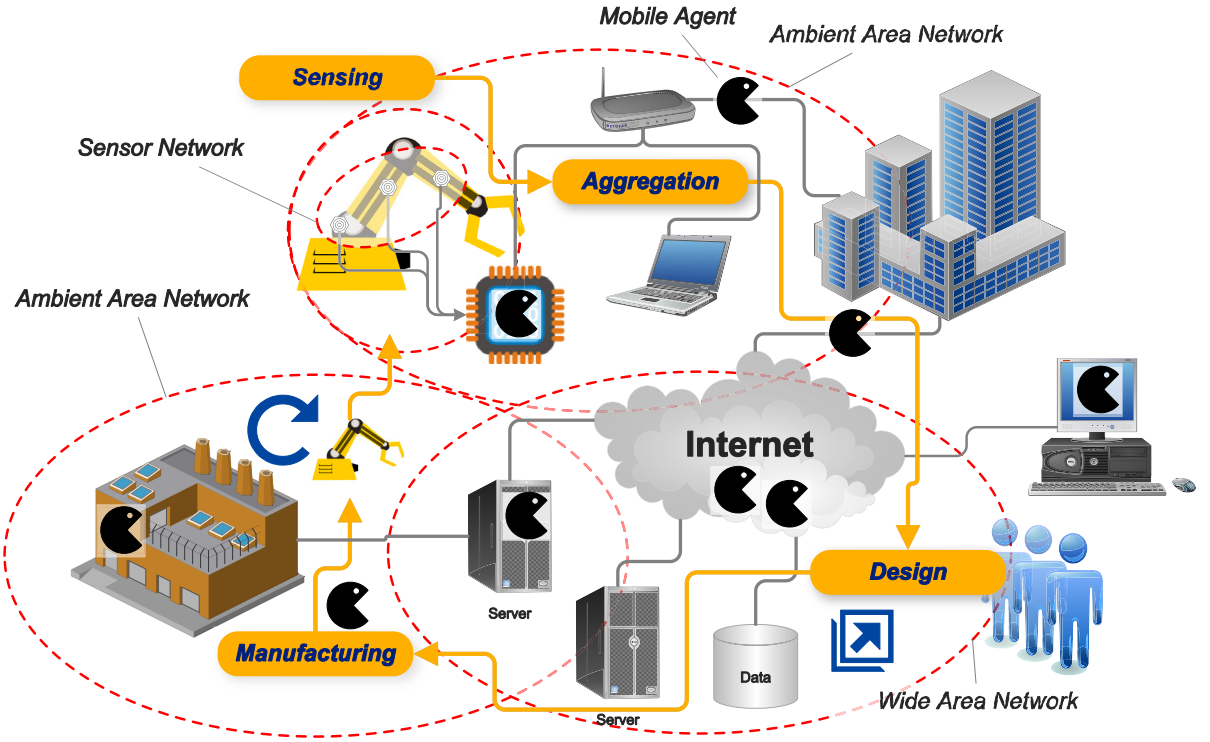

Fig. 5. Cloud-based computing with agents: Additive and adaptive Manufacturing with aggregation and back propagation of sensing data from robots to the design process using mobile agents resulting in semi-continuos series improvements.

7. Conclusions

Agents are represented by mobile program code that can be modified at run-time by agents and that is processed by a modular and portable agent platform. The presented approach enables the development of sensor clouds and smart systems of the future integrated in daily use computing environments and the Internet. Agents can migrate between different hardware and software platform implementations including WEB browsers and *JavaScript* platforms by migrating the program code of the agent, embedding the state and the data of an agent, too. 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.

References

[1] S. Bosse, *Distributed Agent-based Computing in Material-Embedded Sensor Network Systems with the Agent-on-Chip Architecture*, IEEE Sensors Journal, Special Issue on Mateiral-integrated Sensing, DOI 10.1109/JSEN.2014.2301938

[2] S. Bosse, *Design and Simulation of Material-Integrated Distributed Sensor Processing with a Code-Based Agent Platform and Mobile Multi-Agent Systems*, Sensors (MDPI), 15 (2), pp. 4513-4549, 2015, DOI:10.3390/s150204513.

[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] 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

[5] 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.

[6] 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

[7] A. Rogers, D. D. Corkill, N. R. Jennings, *Agent Technologies for Sensor Networks*, IEEE Intelligent Systems, vol. 24, no. 2, 2009

[8] J. Liu, *Autonomous Agents and Multi-Agent Systems,* World Scientific Publishing, 2001 (ISBN 981-02-4282-4

[9] S. Bosse, *VAMNET: the Functional Approach to Distributed Programming*, SIGOPS Oper. Syst. Rev., 40, pp. 108-114, 2006, DOI:10.1145/1151374.1151376