DOI: 10.15439/2015F252
ACSIS, Vol. 6

A Unified Distributed Computing Framework with Mobile Multi-Agent Systems and Virtual Machines for Large-Scale Applications: From the Internet-of-Things to Sensor Clouds

Stefan Bosse
University of Bremen, Dept. of Mathematics & Computer Science,
Robert Hooke Str. 5, 28359 Bremen, Germany

Abstract— **A novel and unified design approach for reliable distributed and parallel data processing in wide-area and large-scale networks consisting of high- and of low-resource nodes (ranging from generic computers to microchips) using mobile agents is introduced. The development of sensor clouds of the future integrated in daily use computing environments and the Internet is enabled. 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. Agent mobility crossing different execution platforms, agent interaction by using tuple-space databases, and agent code reconfiguration enable the design of reliable distributed sensor and information processing networks. The Agent Processing Platform exists in hardware (microchip level), software (embedded system), and simulation. This works adds a JavaScript implementation including client-side browser applications. All implementations are compatibility on operational and communication level. A graph-linked multi-broker service and a distributed co-ordination layer are established for this platform class to provide service ports and the access of the agent platform from the outside in browser applications, which can usually only act as clients and are usually hidden by a private network and firewalls.**

I. Introduction

TRENDS recently emerging in engineering and micro-system applications such as the development of sensorial materials **[3][11]** show a growing demand for distributed autonomous sensor networks of miniaturized low-power smart sensors embedded in technical structures. Multi-agent systems (MAS) can be used for a decentralized and self-organizing approach of data processing in a distributed system like a resource-constrained sensor network (discussed in **[11]** and **[12]**), enabling smart and adaptive distributed information extraction, e.g., based on pattern recognition (e.g., referring **[13]** and **[14]**), by decomposing complex tasks in simpler cooperative agents. It can be shown that MAS-based data processing approaches are scalable from generic computer to single microchip level platforms which can aid the material-integration of Structure and System Monitoring applications. On one hand there are currently only few proposed agent processing platforms that can be scaled to microchip level, and on the other hand there are no unified solutions to integrate these low-resource nodes in large-scale networks and the Internet.

In **[11]** the agent-based architecture considers sensors as devices used by an upper layer of controller agents. Agents are organized according to roles related to the different aspects to integrate, mainly sensor management, communication and data processing. This organization isolates largely and decouples the data management from changing networks, while encouraging reuse of solutions.

The deployment of agents can overcome interface barriers and closes the gap arising between platforms and environments differing considerably in computational and communication capabilities, enabling, e.g., the integration of sensor networks in large-scale WWW applications and providing Internet connectivity, shown in Fig. **1**. This is addressed in this work by using a unified reactive agent-based programming and interaction model, independent of the underlying processing platform. For the proposed advanced agent processing platform architecture there exist suitable hardware (microchip), software (*C*, *OCaML*, *JavaScript*), and simulation model implementations, which can be functionally interconnected in networks creating one big machine. They are compatible on the operational and execution level, thus, agents can migrate between these different implementation platforms.

Agent mobility crossing different execution platforms, agent interaction by using tuple-space databases, and global signal propagation aid solving data distribution and synchronization issues in the design of distributed wide-area networks.

Usually sensor processing and information computation require known world models including mechanical models, e.g., in load monitoring use cases of technical structures. Self-organizing MAS **[2][14]** are useful in unreliable and partially unknown environments, which can overcome world environment and model limitations successfully. Adaptation of the agent behaviour, i.e., based on learning, offers a reliable reaction mechanism in the presence of environmental changes, e.g., changes in network connectivity or node failures, ensuring the QoS. This adaptivity is addressed in this work by a behavioural reconfiguration at run-time, which bases on Dynamic Activity-Transitions Graphs (DATG). Mobility - the ability to migrate an agent processing unit to a different execution platform or node - and autonomy together with a high degree of independency from the processing platform ensure robust data processing in large-scale networks.

It can be shown that agent-based computing can be used to partition complex computations in off-line and on-line parts resulting in an increased overall system efficiency (performance and energy demands), e.g., for Load and Structural Health Monitoring (LM/SHM) systems, outlined in **[2]**.

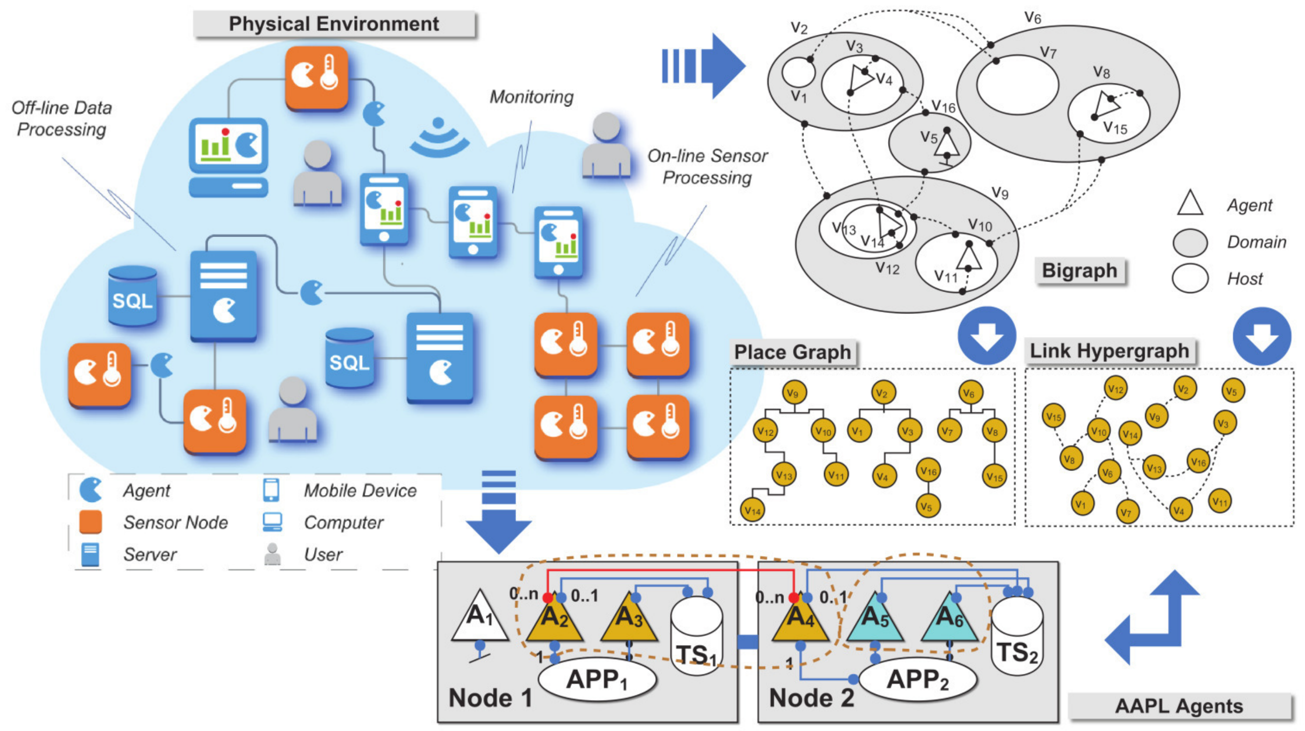


Fig. 1 (Left) Deployment of Agents in Sensor Clouds and Internet Applications (Right) Bigraph, composed of Link and Structure place graphs used for a unified modelling of network environments and networks of networks (Bottom) AAPL agents in the Bigraph Model with a bottom port for the APP link and top port for tuple space and signal link ports. Shown are two connected nodes. [A: Agent, APP: Agent Processing Platform, TS: Tuple Space].

One major goal of the deployment of MAS is overcoming heterogeneous platform and network barriers arising in large scale hierarchical and nested network structures (i.e., networks of networks), consisting and connecting, e.g., the Internet, sensor networks, body networks, production and manufacturing Cyber-Physical System (CPS) networks, shown in Fig. **1** on the left. The large diversity of execution platforms, network topologies, services provided by network nodes, and the programming environments require a unified and abstract behavioural and structural representation model. The Bigraphical model proposed by Robin Milner models the entire "computing" environment with place and link graphs, composing finally bigraphs **[15]**, shown on the right of Fig. **1**. They include agents, and they are offering a unified model and platform for ubiquitous systems and the foundation for an Ubiquitous Abstract Machine, and supporting reconfigurable spaces (dynamic topologies). Bigraphs virtualize communicating processes (agents) and information objects (tuple-spaces), and they originate in process calculi for concurrent systems, especially the pi-calculus **[16]** and the calculus of mobile ambients **[17]** for modelling spatial configurations of networks with a dynamic topology.

The environment consists of places where computation occurs, e.g., computers (processing agents), agents, rooms, buildings, machines, and so on. The links are abstract, providing the possibility of interaction between different places, i.e., transferring of agents and their mobile processes. Agents are treated as active computational units. Places introduce spatial and logical bindings. Bigraphs allow the nesting of nodes and places, natural for many real-world computing environments, and they can be applied for wide reactive systems. All nodes have a fixed number of ports, providing an endpoint for links. Agents have two ports: a processing port link and an interaction (communication) link. Bigraphs, which represents the system state, can be modified by the application of reaction rules, which changes the linking and place relations. Bigraphs can be composed of other bigraphs matching inner and outer interfaces.

A link is a hyperedge connection that connects nodes, outer, and inner names, where names are open linkings that support additional connectivity, i.e., used for the dynamic composition of bigraphs at "run-time". Connectivity not only provides the platform for agent migration between different places, it provides information exchange, which is provided here by place-bounded tuple-spaces and signals. Migration of mobile processes is just another form of interaction with and the modification of the environment.

To adapt this Bigraphical Reactive System (*BRS*) model to a MAS it is necessary to distinguish subjects (entities which can perform actions, the agents) and objects (here data, tuples, tuple-spaces, signals, and processing platforms themselves).

The novelty of this work can be summarized as follows:

- A unified Agent design and processing framework basing on a reactive activity-transition agent behaviour and programming model. Agent interaction is provided by tuple spaces and signal propagation between agents.
- Stack based Virtual Machines (SVM) are used to execute optimized program code embedding the agent behaviour, data and control state in code frames
- The SVM is operating system independent and can be implemented directly in hardware and software including JavaScript
- The JavaScript implementation of the SVM enables the integration of sensor networks and agent-based sensor and information processing in the Internet and Intranet domains.
- The SVM can be embedded in *HTML* content and turns a browser in an agent processing platform.
- A object-capability-based Remote Procedure Call (RPC) communication interface and a distributed graph-linked broker service enables the deployment of client-side applications like browser as agent processing platforms.

II. The State-based Reactive Agent Behaviour Model and AAPL Programming Language

The agent model summarized in this section (for details see **[1][3][4]**) bases on the mobile processes model introduced by Milner **[16]** several decades ago. An agent can be considered as a computational unit situated in an environment and world, which performs computation, basically hidden for the environment, and interacts with the environment to exchange basically data. A common computer is specialised to the task of calculation, and interaction with other machines is encapsulated by calculation and performed traditionally by using messages. An agent behaviour can be reactive or proactive, and it has a social ability to communicate, cooperate, and negotiate with other agents. Proactiveness is closely related to goal-directed behaviour including estimation and intentional capabilities.

II-A. Activity-Transition Graphs

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 transitions start activities commonly depending on the evaluation of agent data (body variables), representing the data state of the agent. The ATG behaviour model is fundamental for Activity-based Agent Programming Language (*AAPL*).

An activity-transition graph, related to the agent classes, discussed later, consists of a set of activities $\boldsymbol{A}=\{A_1,A_2,..\}$, and a set of transitions $\boldsymbol{T}=\{T_1\ (C_1),T_2\ (C_2),..\}$, which represent the edges of the directed graph. The execution of an activity, composed itself of a sequence of actions and computations, is related with achieving a sub-goal or a satisfying a prerequisite to achieve a particular goal, e.g., sensor data processing and distributions.

Usually agents are used to decompose complex tasks in simpler ones. Agents can change their behaviour based on learning and environmental changes, or by executing a particular sub-task with only a sub-set of the original agent behaviour.

An ATG describes the complete agent behaviour. Any sub-graph and part of the ATG can be assigned to a subclass behaviour of an agent. Therefore modifying the set of activities $\boldsymbol{A}$ and transitions $\boldsymbol{T}$ of the original ATG introduces several sub-behaviour for implementing algorithms to satisfy a diversity of different goals. The reconfiguration of activities $\underline{\boldsymbol{A}} = \{ A_1 \subseteq \boldsymbol{A}, A_2 \subseteq \boldsymbol{A},\ ..\}$ from the original set A and the modification or reconfiguration of transitions $\underline{\boldsymbol{T}} = \{ \boldsymbol{T}_1, \boldsymbol{T}_2, ..\}$ create dynamic supersets of ATGs and enable agent sub-classing at run-time.

II-B. The Activity-based Agent Programming Language (AAPL)

The AAPL programming model should optimally match the requirements of MAS deployed in unreliable sensor and wide-area distributed networks, keeping low-resource nodes with low computational power in mind. On one hand, *AAPL* should reflect the core concepts of agents, on the other hand *AAPL* should provide core concepts of traditional programming language to ease the programming of widely used algorithms.

The agent behaviour, perception, reasoning, and the action on the environment are 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 by other agents 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.

AAPL supports the following statements and constructors:

- Agent Class Definition consisting of body variables, activities, transitions, handlers, and common functions;
- Computational and control flow statements: assignment, branches, loops, exception handling;
- Cooperation and Communication with tuple spaces and signal messages (carrying simple data);
- Agent instantiation from agent classes, forking, destroying;
- Agent mobility by migration;
- Agent behaviour modification (e.g., ATG reconfiguration).

II-C. Multi-Agent Interaction

In parallel and distributed systems the communication, synchronization, and data exchange of a collection of data processing units (processes or agents) gains significant importance. A common approach for parallel systems is a shared memory based communication paradigm, but which generates a high computational dependency of the processing units among themselves and regarding the platform. Loosely coupled distributed systems like MAS require a different communication strategy.

Tuple-Spaces. One well known and common distributed interaction model is the tuple-space. Agents can communicate with each other by accessing a tuple space database service available on each network node and that is provided by the agent processing platform (a node in the Bigraph model, see bottom of Fig **1**), used for synchronized data exchange among a collection of individual agents, which was proposed in **[18]** and **[19]** as a suitable MAS interaction and coordination paradigm.. A tuple space is a logically shared memory and is used for synchronized data exchange between producer and consumer, a common approach for solving communication problems of loosely coupled autonomous or semi-autonomous processing units. Tuple spaces are generative, which means a tuple can survive the creator beyond its lifetime. The scope and visibility of a tuple space database can be unlimited and visible and distributed in the whole network, or limited to a local scope, e.g., network node level. A tuple space provides abstraction from the underlying platform architecture, and offers a high degree of platform independency, vital in a heterogeneous network environment.

For the sake of simplicity the scope of a tuple space can be limited to the node boundary, such that there are multiple tuple spaces distributed in the network. Information can be carried by mobile agents between nodes. A tuple space communication model has the advantage of shielding the underlying node and agent processing platform. Access of tuple spaces require only a small set of simple operations {out, in, rd, in?, rd?, rm, eval}, which transfer tuples between a producer or consumer and the database. Since tuples consist of type-tagged values and patterns the tuple space communication is type-safe and strong computational bindings can be avoided.

AAPL Agents. In the Bigraph model *AAPL* agents have different ports. One static port is the platform link, required to execute an agent process. Another port is used for the linking of an agent with a tuple-space (#=1). An *AAPL* agent can have only one tuple-space access and link at any time maximal. The propagation of signals introduce further ports and dynamic links to other agents (#=0..n), see Fig. **1**. The communication links introduce virtual domains, in Fig. **1** these are the agent groups $\{A_2, A_3, A_4\}$ and $\{A_5, A_6\}$. These virtual domains are dynamic, regarding the spatial location and extension, and the agents which are part of the virtual domain. Often agent parent-child trees spawn the virtual domains using signal interaction, but agents of initially different virtual domains can interact by using the tuple-spaces, extending and merging different virtual domains. The spatial extension of virtual MAS domains is constrained by the connectivity graph of the processing nodes.

Signal propagation from a source to a destination agent requires the connectivity of nodes if the agents are executed on spatially different nodes. Tuples stored in tuple-spaces are persistent. That means a tuple t, which was produced by an agent Ag_1 and stored in a tuple-space TS_1, and agent Ag_1 is finally migrating to another node location, can be consumed by a different agent Ag_2, now having a historical relation and link to the other agent Ag_1.

Signals. In contrast, *signals*, which can carry additional scalar data values, can be used for local (in terms of the node scope) and global (in terms of the network scope) domain agent interaction. In contrast to the anonymous tuple-space interaction, signals are directly addressed to a specific agent or a group of agents. The delivery of signals is not reliable in the case the agents raising and receiving the signal are not processed on the same node. An agent being ready to receive signals has to provide a signal handler for this signal, a function that is executed asynchronously to the agent ATG execution.

III. The Agent Code Processing Platform

In this work, the agents are implemented with Agent Forth program code that is executed on virtual stack machines, which can be implemented alternatively on hardware (System-on-Chip), simulation, and software level, which can be embedded in microcontroller, desktop applications, web applications, or server programs. The agent program code (see **[1]**) is a self-containing and self-initializing unit embedding the (private) agent data and the current control state of the agent, which simplifies migration significantly. This machine program is encapsulated in code frames with a specific layout. The program is able to modify itself by using code morphing, leading to a low computational dependency from the current execution environment, which is vital to strong heterogeneous environments. There is only a small set of knowledge about the program which is required by the VM to execute the agent program, and vice versa. Migration of agents requires only the transfer of the code frame from one platform to another. The data and control state of an agent program is stored in the code frame, too. There are two different *Agent FORTH* levels, one supporting high-level constructs like loops and branches (*AFL*), and one low-level machine subset (*AML*) that can be directly executed by the *AFVM* platform. *AFL* has similar operational semantics than *AAPL*. Thus the *AAPL* agent class behaviour definition can be directly compiled to the *AFL* level, finally compiled to *AML* with a specific code frame layout.

In **[2]** and **[3]** there is an example for the *AAPL* behaviour model of a simple explorer agent that is sent out from an agent on a specific network node. The explorer agent has the goal to find another node having a specific feature that is stored in the (local) tuple space database. If the explorer agent found the feature (activity check), it will return the original root node and stores the feature in the tuple space with the relative delta position of the node where the feature tuple was found (activity deliver). The explorer agent moves through the network in a random direction until a maximal number of hop counts is reached (parameter radius, activity migrate). The respective *AFL* program (see [1]) reflects roughly the operational semantics and structure of the *AAPL* program source. The compiled *AML* machine program that can be executed by the *AVM* consists of a boot section at the beginning of the code frame, followed by a data section storing the private agent variables and parameters. Finally all activities and the transition table conclude. The entire machine program requires less than 400 words (800 bytes for a 16 Bit machine), which can be efficiently transferred between different processing hosts.

III-A. AFVM Platform Architecture

The virtual machine (*AFVM*, discussed in depth in [1]) executing tasks bases on a traditional *FORTH* stack processor architecture and an extended zero-operand word instruction set (αFORTH). 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 organized by a composition of words (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, 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 is a standalone and auto-initializing unit that encapsulates basically 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 for the setup of agents in a new environment (i.e., after migration or on first run), and 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.

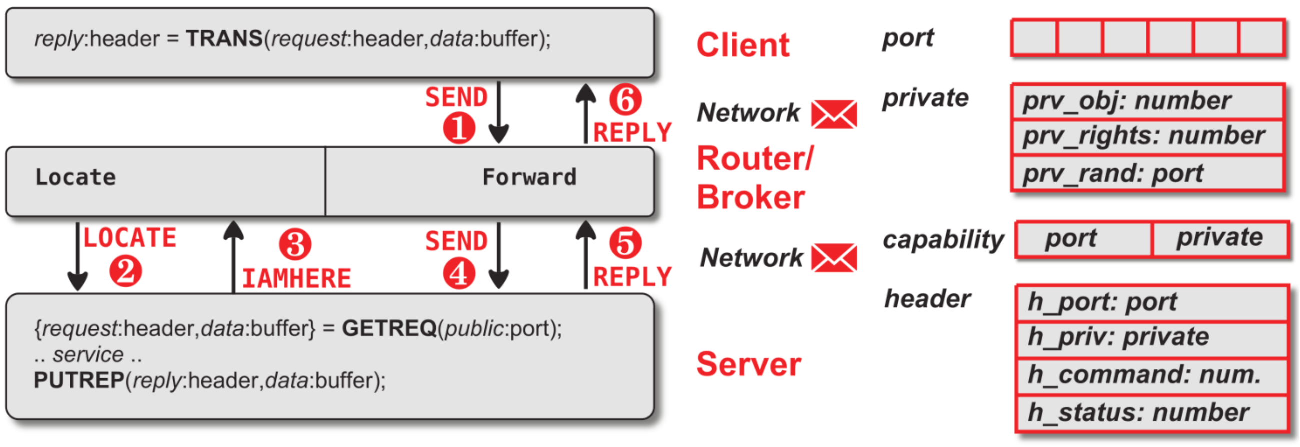
Each VM processor is connected with an agent process manager (AM). The VM and the agent manager share the same VM code segment and the process table (PT). The process table contains only basic information about processes required for the process execution.

Commonly the number of agent tasks N_A executed on a node is much larger than the number of available virtual machines N_V. Thus, efficient and well-balanced multi-task scheduling is required to get proper response times of individual agents. To provide fine grained granularity of task scheduling, a token based pipelined task processing architecture was chosen. 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). Therefore, the return from an agent activity word execution (leaving empty stacks) is an appropriate task scheduling point for a different task waiting in the VM processing token queue. This task scheduling policy allows fair and low-latency multi-agent processing with fine grained scheduling. Furthermore, this kind of task scheduling enables the *JavaScript* implementation, discussed in Sec. IV-E.

IV. The JavaScript WEB Platform JAVM

The mobility of agents is handled basically by the agents themselves, and there is no advanced routing provides by the platform. They make decisions about the migration direction and the selection of neighbour nodes, usually basing on some geometrical structures given by the network topology. For example, a material-integrated sensor network embedded in a wind energy wing used for Load Monitoring has a mesh-like network topology consisting of nodes that are connected with their nearest neighbours. Delivering of sensor data to dedicated computing nodes can be performed simply by travelling to the outside of the network and by searching. In the Internet context this geometrical structure and the neighbourhood connectivity do not exist, or at least they are not visible, increasing the decision and reducing the knowledge space of agents significantly. First of all, the migration decision of agents must base on different features and knowledge. Furthermore, the Internet consists of two different kinds of network nodes: Nodes capable of providing a public visible service, called servers, and nodes that cannot publish server ports. But in distributed systems each node must be capable of offering services. Two computers can only connect if at least one computer has public server ports, otherwise an external brokerage service is required. Web browsers are usually processed on client computer nodes and are not visible in the network. Therefore, agents can’t select a client-interface-only node or process for migration directly and autonomously due to the missing visibility in the communication network, as this is the case in traditional sensor or embedded networks.

Two main issues arising in Internet applications using mobile agents must be addressed: 1. The definition and the knowledge representation of virtual/artificial neighbourhood connectivity in loosely coupled and hierarchical graph-based networks based on semantic rather on physical connectivity. 2. The visibility and deployment of pure client-side applications like Web browsers and computers hidden in private or restricted networks as agent processing platforms capable of receiving, processing, and sending of agents.



Def. 2 RPC-based client-server communication types, operations, and protocol schema (phases of a transaction)

To enable the distributed agent processing in browser and applications running on generic computers connected by the Internet, the previously introduced *Agent Forth Virtual Machine* (*AFVM*) platform was implemented in JavaScript that can be executed either by a *node.js* interpreter or by any browser capable to execute *JavaScript* code. The *AFVM* was integrated in a distributed operating system layer, also implemented entirely in *JavaScript*, discussed in the following subsections, composing the *JAVM* platform. The transition from peer-to-peer networks to routed and hierarchical networks like the Internet requires some methodological and architectural changes, introducing the aforementioned broker service, discussed below.

IV-A. Inter-Node Communication and RPC

Nodes offering agent processing capabilities connected in the Internet domain usually not communicating peer-to-peer like in sensor networks with mesh topologies. Instead routing is used to establish communication between different application processes executed on nodes probably located far away. One well known inter-process communication approach is the Remote Procedure Call (RPC), e.g., extensively used in the distributed operating system Amoeba **[21]**, or on the top of existing operating system, e.g., offered by the distributed Common Object Request Broker Architecture (*CORBA*) framework. The capability-based RPC communication from the Amoeba OS was already successfully implemented in VM environments executed on top of existing operating systems (*VAMNET*, **[5]**).

The RPC communication interface is used in this work for the inter-platform communication, e.g., for transferring agent program code to another platform or to access distributed file and naming services. The RPC ontology consists of servers and clients communicating by using a set of operations. A server performs a GETREQ operation to publish a listening on a public server port, and a client performs a transaction TRANS operation to access a server identified by the public server port. Each server handles a set of objects, identified by capabilities that are tuples ⟨*port*, *obj*, *rights*, *rand*⟩, consisting of the server port, an object number, a rights field, and a private protection field authorizing the rights field. A transaction operation transfers object capabilities to the server that handles the request and finally replies by using the PUTREP operation. Therefore, a client transaction is synchronous and blocks the client process until the reply arrives or an error occurred (time-out). The localization of the server and the routing of the messages is hidden by the RPC layer, or more precisely by the underlying protocol layer, shown in Def. **1**. The localization is basically performed by broad- or multicasting LOCATE messages to nodes in the current domain and finally to a limited number of boundary domains. Each node monitors the locally registered servers, and replies with a IAMHERE message. Nodes are identified with ports, too.

The RPC communication is encapsulated in *HTTP* messages with *XML* content and transferred using the generic *HTTP* protocol, discussed in section IV-C. The RPC header and data is stored inside *XML* tags with compacted hexadecimal coded text, on one hand complaining with the *XML* standard, on the other hand reducing and optimizing the payload. The binary byte data is coded with two hexadecimal digits for each data byte. Each RPC server (process) can act as a client, too, and vice versa.

IV-B. Domains as Organization Structures and the Directory Name Service

Domains are groups of agent processing nodes that are coupled in a network. Agents can migrate between nodes of a group. A node can be assigned to more than one domain, enabling the migration of agents between domains. Node domain composition bases on

1. Geometrical localization and proximity, basically expressing and simulating neighbourhood connectivity
2. Information and data context
3. Tasks to be performed, cooperative goals to be satisfied
4. Logical network domains

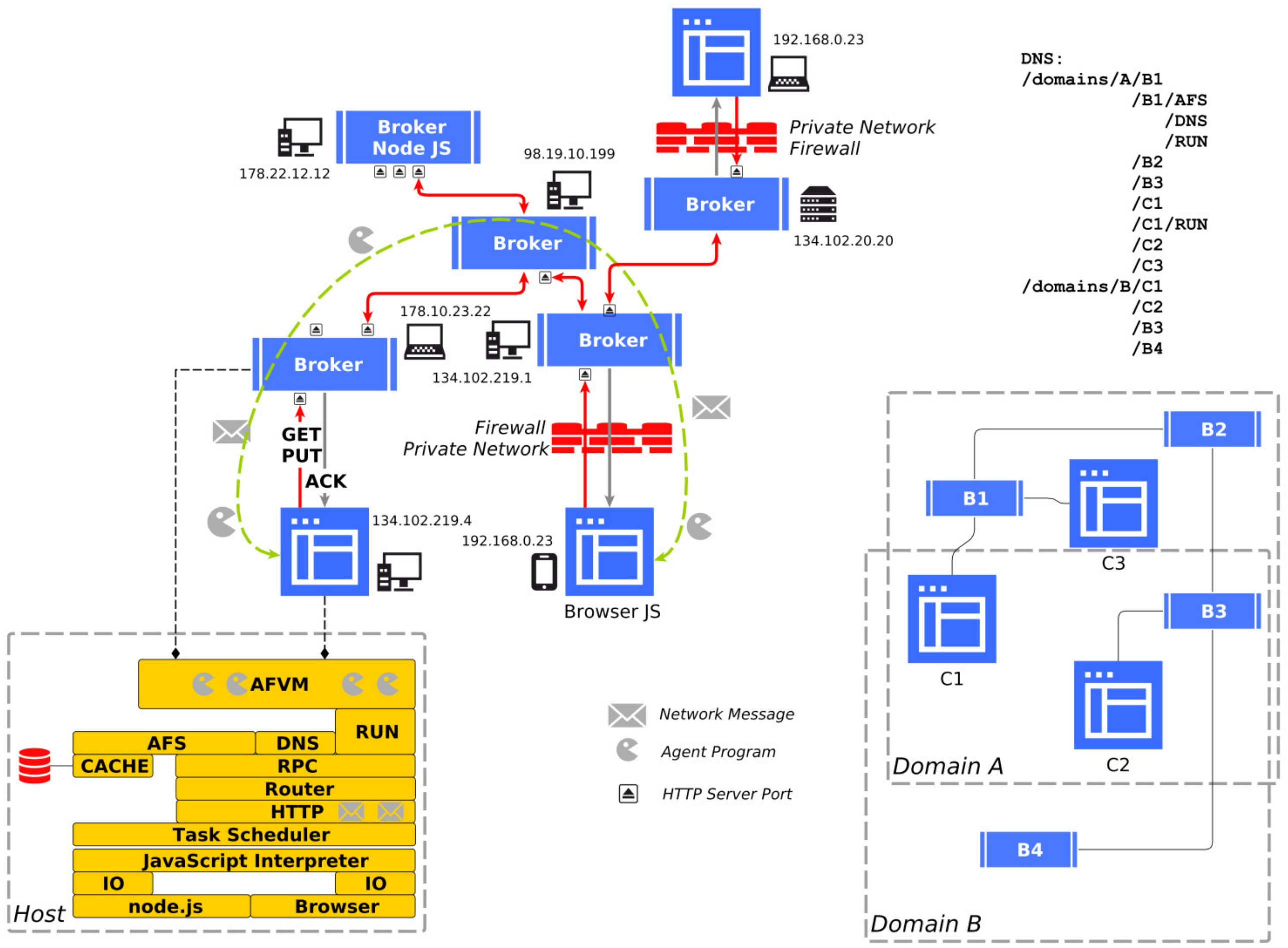


Fig. 2 (Left, Centre) Broker Network with HTTP server ports and client applications (browser, node.js client-side) connecting to the public visible broker server ports. Client-to-Client communications takes place over the broker servers. (Left, Bottom) The JavaScript agent platform JAVM and the modules and services available on each host (Right) Different nodes can be bound to (overlapping) domains published in the DNS.

Domains can be expressed by paths similar to directory trees that are handled usually by a file system. In this work a distributed and unified Directory Name Service (*DNS*) us used that provides a database to publish (capability-name) pairs organized in trees. Each object in the distributed system is related to a capability, which is serviced by a specific server. For example, a file containing the agent program code is serviced by a file server. A directory containing domains is an object, too, handled by the *DNS* server. An agent platform that processes agents programs is another kind of object, handled by a run server that exists on each node. Agents are objects in this sense, but they don't belong to a specific server, therefore they are handled as mobile and autonomous severs. In Fig. **2**, an example for a composition of domains consisting of network nodes that are not directly connected is shown.

IV-C. 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 domains. Though there are already some approaches for interconnecting browser applications directly (client-to-client communication using *WebSockets* or *WebRTC* **[20]** and *HTML5* standards), they are not supported by all browsers and require some external server for the connection brokerage, too. Furthermore, *WebSockets* are still under development and there are many browser incompatibilities. To provide compatibility with and among all existing browser applications none of these technologies were used. Instead, an object-capability-based RPC inter-process communication with a 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, e.g., a sensor network. A broker is just an application program capable of running on any computer visible globally in the Internet or more locally in some Intranet domains.

An agent processing node (e.g., a host application) that cannot publish IP server ports must connect to one of the broker servers visible in the network. Usually this should be a server located nearby. Each node is associated with a host port that is communicated to the broker server now handling and forwarding service requests for this specific host, shown on the lower left side of Fig. **2**. Each client-side host collects periodically pending and queued service request messages (or replies of services requests) from the broker server and passes services replies back to the broker server that forwards the reply to the appropriate host performing originally a transaction. If the two hosts involved in a RPC transaction are not handled by the same broker server, the source broker server must forward request and reply messages to the appropriate destination broker server, shown in Fig. **2** by the green dotted path line. Furthermore, a broker server must handle local RPC transactions and local RPC servers and, too.

IV-D. The Node Service Platform

In addition to the services provides by the agent processing platform (i.e., the agent manager and the tuple-space database), each network broker node and optionally each browser or client-side application provide a file system service (Atomic File System Service *AFS*), the aforementioned Directory and Naming Service (*DNS*), and a run server connected to the agent processing platform (required on each host). The run server provides the public port for agent execution, migration, and signal message propagation between agents.

IV-E. The JavaScript Implementation

There are basically two different execution environments for the execution of *JavaScript* (*JS*) programs: The server-side standalone *node.js* interpreter and the client-side *JS* interpreter embedded in browser applications. The *node.js* interpreter can execute a *JS* program directly (with source-to-machine code compilation on demand), whereas the browser executes *JS* embedded in *HTML* content only. There are *node.js* modules enabling the setup of *HTTP* servers, modules for accessing files on the local file system, and many more OS related programming interfaces not available in the client-side browser *JS*.

The implementation of the entire network node services, the RPC communication, and the agent processing platform with JavaScript is a challenge, but offers significant advantages with respect to portability, compatibility, and the design unification for server-side and client-side-only platforms (e.g., browsers). The basic modules implemented on each host (and browser application) are shown on the left bottom side of Fig. **2**, consisting at least of the RPC module, the *HTML* wrapper, and the agent processing platform *AFVM*.

JavaScript is executed strictly single threaded, though functions can be executed in parallel and concurrently, there is no concept of process blocking or any other synchronization. In JavaScript programs input-output operations are mainly performed with asynchronous callback functions. But all RPC services, the agent processing platform, and servers operate inherently multi-threaded and synchronously.

To overcome this execution limitation, a *Task Scheduler* (TSCH) was invented that simulates parallel multiprocess execution and enables virtual process blocking for the synchronization of processes. Each process consists of a set of activities (functions) that are enabled by a conditional transition expression (that can be a constant true value). The scheduler executes all activity functions sequentially that have a satisfied transition condition. Blocking of a process sets a process specific blocking variable (the guard GD) that is part of the transition condition from the blocking activity to the next one to be executed after the process was woken up again. Furthermore, there are block, conditional, and loop scheduling constructors easing the programming of processes. All RPC operations are prepared for the scheduler management. Though callback functions are still used, a single program flow of processes can be constructed on programming level.

The client-side Browser *JS* implementation is created by compacting and relocating server-side dependencies (using *browserify*, *envify*, and *uglifyjs* for minimizing), and requires typically about 500kB text size.

V. USE-CASE: CLOUD BASED ADAPTIVE MANUFACTURING AND ROBOTS AS PRODUCTS

This section outlines a big application use-case for the introduced agent processing platform with an architecture for additive and adaptive manufacturing based on a closed-loop sensor processing approach, extended 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 **[6]**, directly involved in the manufacturing process using distributed cloud computing and distributed storage solutions.

Robots 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 preprocessing, 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 next layer in cloud-based adaptive manufacturing process can be the inclusion of the products themselves delivering operational feedback to the current design and manufacturing process, leading to a closed-loop evolving design and manufacturing process with an evolutionary touch, shown in Fig. **3**. 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. This process can be considered as a continuously evolving improvement of the robot by refining and adapting design parameters and constraints that are immediately migrated to the manufacturing process. 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 product 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. One major question to be answered is the sensing of meaningful condensed product condition information and the delivery to the designer and factory. The proposed mobile agent model offers a self-contained and autonomous virtual processing unit that is well suited for such large-scale applications. The mobile agents represent mobile computational processes that can migrate in the Internet domain and as well in sensor networks.

Agents are already deployed successfully for scheduling tasks in production and manufacturing processes **[7]**, and newer trends poses the suitability of distributed agent-based systems for the control of manufacturing processes **[8]**, 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. The MAS paradigm offers a unified data processing and communication model suitable to be employed in the design, the manufacturing, logistics, and the products themselves.

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 **[8]**. Currently, many traditional processing platforms cannot yet handle big numbers with the robustness and efficiency required by industry **[9][10]**. In the past decade the capabilities and the scalability of agent-based systems have increased substantially, especially addressing efficient processing of mobile agents.

There programmable agent processing platform introduced in this work can be deployed in strong 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. 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 **[2]**. Distributed data mining and Map-Reduce algorithms are well suited for self-organizing MAS. Cloud-based computing, as a base for cloud-based manufacturing, means the virtualization of resources, i.e., storage, processing platforms, sensing data or generic information.

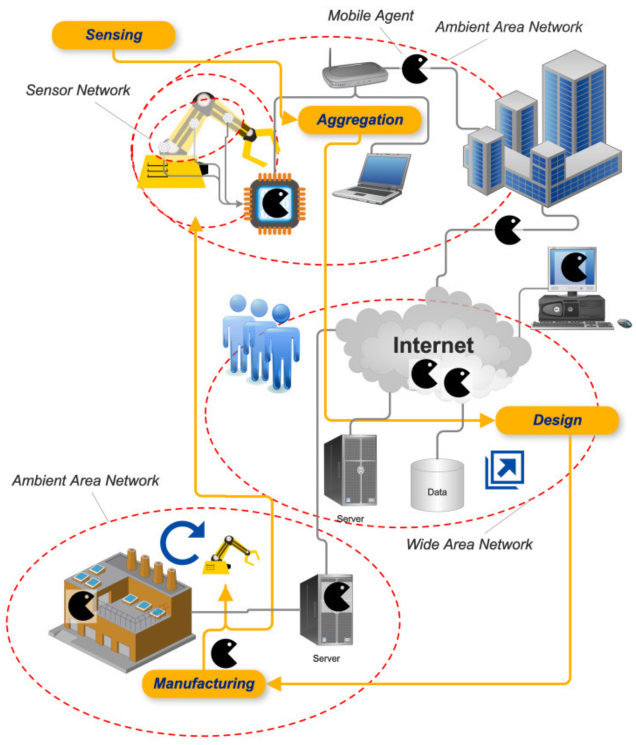


Fig. 3 Additive and adaptive Manufacturing with back propagation of sensing data using mobile *AAPL* agents and the JAVM/PAVM platform.

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, triggered by the data sources (the robots) themselves, can improve and reduce the allocation of computational, storage, and communication resources significantly.

A cloud in terms of data processing and computation is characterized by and composed of: 1. A parallel and distributed system architecture; 2. A collection of interconnected virtualized computing entities that are dynamically provisioned; 3. A unified computing environment and unified computing resources based on a service-level architecture; 4. 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.

Agent Classes. Different agent classes are defined that satisfy different sub-goals: event-based sensor acquisition including sensor fusion (Sensing), aggregation and distribution of data, preprocessing 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 transferred in messages with a size lower than 4kB.

VI. Conclusion and Outlook

In this work, a novel **Agent Processing Platform** architecture for code-based mobile agents in large-scale and wide-area heterogeneous networks including low-resource microchip nodes embedded in sensor networks was introduced. The standalone agent processing platform, a multi-core stack processor, can be implemented entirely on microchip level, and requires no operating system and no boot code. Alternatively, the processing platform can be implemented efficiently in software with code and operational compatibility, enabling the deployment in heterogeneous network environments, inter-connecting hardware and software platforms executed on generic microprocessors. The *JavaScript* implementation of the processing platform together with a minimal distributed operating layer consisting of a broker, RPC, run, file, and naming services enables the integration of body area, ambient, and sensor networks in the Internet domain, a prerequisite for the future of Internet-of-Things and Sensor Clouds in daily use computing environments. Agents can migrate between different hardware and software platforms (they are compatible on the execution level) by migrating the program code of the agent, embedding the state and the data of an agent, too. A broker service enables the integration of hosts (generic computers, mobile devices, ...) that are not visible in the Inter- or Intranet domains and that cannot publish server communication ports.

Using this broker service, which is composed of a graph-based network of single broker server applications, each computing device capable of executing *JavaScript* code can act as an agent processing platform. This agent processing platform is capable of receiving mobile agents form other platforms and hosts. The broker service creates virtual connectivity based on domains.

References

[1] S. Bosse, *Design and Simulation of Material-integrated Distributed Sensor Processing with a Code-based Agent Platform and mobile Multi-Agent Systems*, MDPI Sensors, 2015 (2), pp. 4513-4549, 2015, http://dx.doi.org/10.3390/s150204513

[2] S. Bosse and A. Lechleiter, *Structural Health and Load Monitoring with Material-embedded Sensor Networks and Self-organizing Multi-agent Systems*, Procedia Technology, Proceeding of the 2nd SysInt Conference, Bremen, Germany, 2014, http://dx.doi.org/10.1016/j.protcy.2014.09.039

[3] S. Bosse, *Distributed Agent-based Computing in Material-Embedded Sensor Network Systems with the Agent-on-Chip Architecture*, IEEE Sensors Journal, http://dx.doi.org/10.1109/JSEN.2014.2301938

[4] S. Bosse, *Design of Material-integrated Distributed Data Processing Platforms with Mobile Multi-Agent Systems in Heterogeneous Networks*, ICAART 2014, http://dx.doi.org/10.5220/0004817500690080

[5] S. Bosse, *VAMNET: the Functional Approach to Distributed Programming*, SIGOPS Oper. Syst. Rev., 40, pp. 108-114, 2006, http://dx.doi.org/10.1145/1151374.1151378.

[6] D. Wu, J. L. Thames, D. W. Rosen, and Dirk Schaefer, *Towards A Cloud-based Design and Manufacturing Paradigm: Looking Backward, Looking Forward*, in Proceedings of the ASME 2012 International Design Engineering Technical Conference & Computers and Information in Engineering Conference, IDETC/CIE 2012 August 12-15, 2012, Chicago, Illinois, USA, 2012

[7] 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.

[8] P. Leitão and S. Karnouskos (ed.), in *Industrial Agents Emerging Applications of Software Agents in Industry*. Elsevier, 2015.

[9] V. Marík, and D.C. McFarlane, 2005. *Industrial adoption of agent-based technologies*. IEEE Intell. Syst. 20 (1), 27–35.

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

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

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

[13] X. Zhao, S. Yuan, Z. Yu, W. Ye, and J. Cao, *Designing strategy for multi-agent system based large structural health monitoring*, Expert Systems with Applications, 2008, 34(2), 1154–1168. doi:10.1016/j.eswa.2006.12.022

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

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

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

[17] L. Cardelli and A: Gordon, *Mobile Ambients*. Theoretical Computer Science, Special Issue on Coordination 240(1), 177–213 (2000)

[18] 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

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

[20] S. Loreto and S. Pietro Romano, *Real-time communications in the web: Issues, achievements, and ongoing standardization efforts*, IEEE Internet Computing, vol. 16, no. 5, pp. 68–73, 2012.

[21] 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.