Distributed Computing and Reliable Communication in Sensor Networks using Multi-agent Systems

Stefan Bosse[(1,3)], Florian Pantke[(2,3)]

(1) University of Bremen, Department of Computer Science, Workgroup Robotics

(2) TZI-Center for Computing and Communication Technologies, Univ. of Bremen

(3) ISIS Sensorial Materials Scientific Centre, Univ. of Bremen

Abstract

There is a growing demand for robust distributed computing and systems in sensor networks. Interaction between nodes is required to manage and distribute information. One common interaction model is the mobile agent. An agent approach provides stronger autonomy than a traditional object or remote-procedure-call based approach. Agents can decide for themselves, which actions are performed, and they are capable of flexible behaviour, reacting on the environment and other agents, providing some degree of robustness. The focus of the application scenario lies on sensor networks and low-power, resource-aware single System-On-Chip (SoC) designs, i.e., for use in sensor-equipped technical structures and materials. We propose and compare two different data processing and communication architectures for the implementation of mobile agents in sensor networks consisting of single microchip low-resource nodes. Furthermore, a reliable smart communication protocol for incomplete and irregular networks are introduced. Two case studies show the suitability of agent-based approaches for distributed computing.

Keywords

Distributed Computing, Agent, Sensor Network, Energy Management, Data Fusion

1. Introduction

Trends recently emerging in engineering and micro-system applications such as the development of sensorial materials show a growing demand for autonomous networks of miniaturized smart sensors and actuators embedded in technical structures **[6]**. With increasing miniaturization and sensor-actuator density, decentralized network and data processing architectures are preferred or required. A multi-agent system can be used for a decen-

tralized and self-organizing approach of data processing in a distributed system like a sensor network, enabling the mapping of distributed data sets to related information, for example, required for object manipulation with a robot manipulator.

Simplification and reduction of synchronization constraints owing to the autonomy of agents is provided by the distributed programming model of mobile agents **[5]**.

Traditionally, mobile agents are executed on generic computer architectures **[7][8]**, which usually cannot easily be reduced to single microchip level like they are required, e.g., in sensorial materials with high sensor node densities.

In the following sections, we propose and compare two different data processing and communication architectures suitable for the implementation of mobile agents in sensor networks consisting of single microchip low-resource nodes. A reliable communication protocol suitable for robust communication in agent based systems is introduced and analysed. Finally, the two agent processing architectures are compared.

2. Distributed Data Processing with State-based Agents

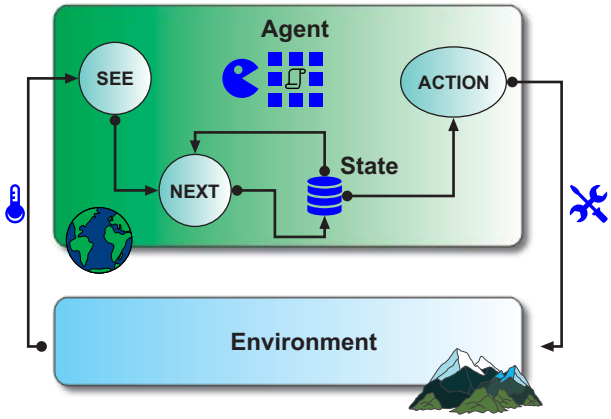
Initially, a sensor network is a collection of independent computing nodes. Interaction between nodes is required to manage and distribute data and computed information. One common interaction model is the mobile agent. An agent is capable of autonomous action in an environment with the goal to meet its delegated objectives. An agent is a data processing system, a program executed on a computer system, that is situated in this environment **[1]**. A multi-agent system is a collection of loosely coupled autonomous agents migrating through the network. Agents can be used in sensor networks for

- Sensor data processing and extraction
- Sensor data fusion, filtering, and reduction of sensor data to information in a region of interest
- Sensor data and information distribution and transport
- Global energy management, exploration and negotiation

Agents can operate state-based. Such an agent consists of a state, holding data variables and the control state, and a reasoning engine, implementing

behaviours and actions. In this proposed data processing and communication architecture, the state of an agent is completely kept in messages transferred in the network providing agent mobility. The functional behaviour of an agent can be easily implemented statically with a finite-state machine part of the local data processing system on register-transfer level (RTL), or dynamically by using a programmable code approach.

Fig. 1. State-based agents and interaction with environment.



Agents record information about the environment state $e \in E$ and history. Let I be the set of all internal states of the agent. An agent's decision-making process is based on this information. The perception function *see* maps environment states to perceptions, function *next* maps an internal state and percept to an internal state, the action-selection function *action* maps internal states to actions (see also Fig. **1**):

$$see : E \rightarrow Per$$
$$next : I \times Per \rightarrow I$$
$$action : I \rightarrow Act$$

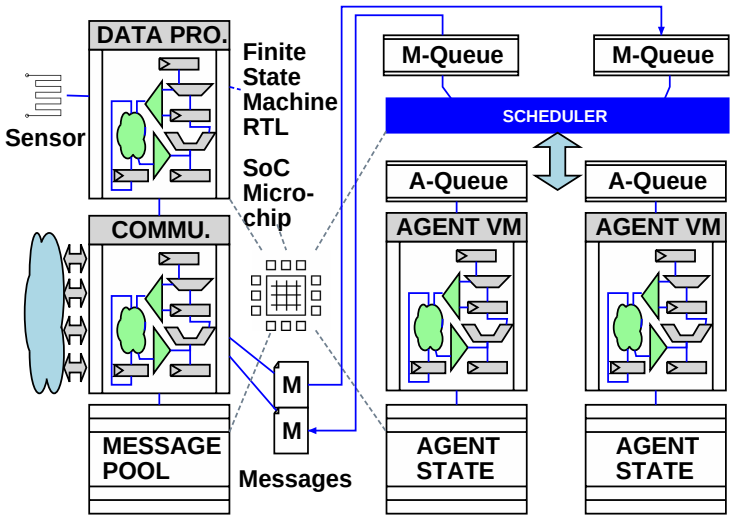
3. Approach I: Non-programmable Message-Based/State Machine Agent Processing Architecture

Figure **2** shows the first proposed non-programmable execution environment used for the data processing of mobile agents. This execution environment is preferred for low-resource implementations of mobile agents with low algorithm complexity. All nodes must comply with data structures and message formats specified at design time required for the cooperation of agents.

There is a message module implementing smart adaptive delta-distance routing of messages (SLIP, the Scalable Local Intranet Protocol, explained

later), providing some kind of fault-tolerance regarding interconnect failures, and several finite-state machines implementing the agent behaviours and providing virtual machines able to process incoming agents. All parts are mappable to digital logic on RT level and single-SoC system architectures, a prerequisite for miniaturized sensor nodes embedded in structures and sensorial materials.

Fig. 2. **Sensor node building blocks providing mobility and processing for multi-agent systems: parallel agent virtual machines, agent-processing scheduler, communication, and data processing. All parts are mappable to digital logic on RTL and SoC system architecture.**



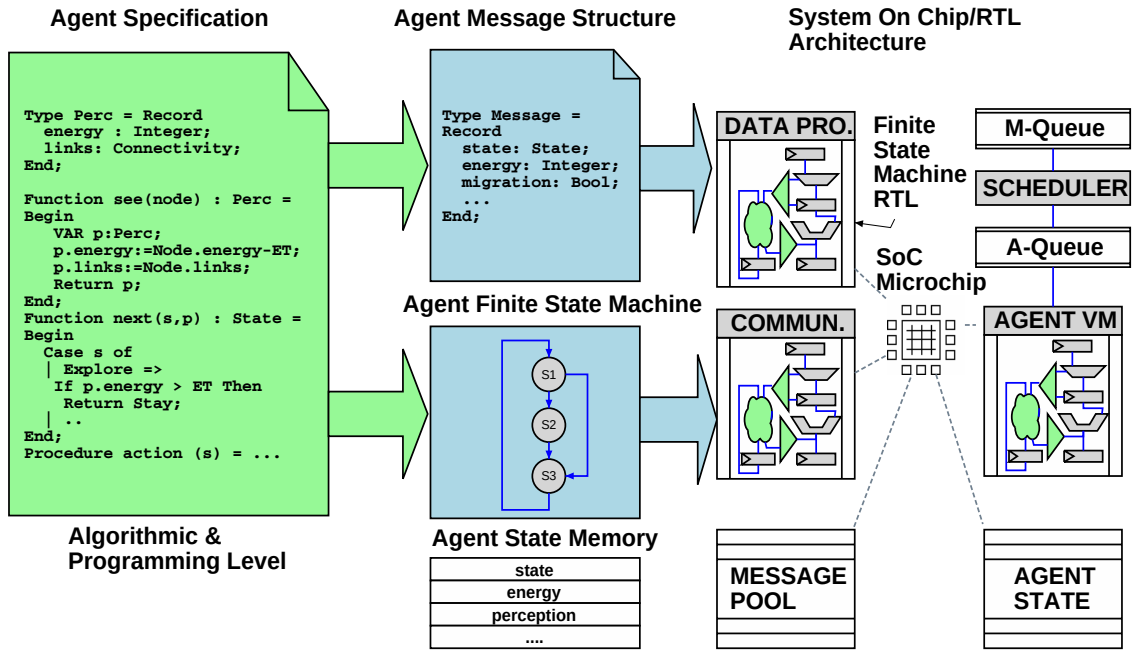
The functional agent behaviour is implemented with a (non-mobile) finite state machine (virtual machine) built in the sensor node, modelled with a high-level synthesis approach on an imperative multi-process programming language level **[3]**.

Inter-agent communication is provided by shared data structures, available on each sensor node. Each node is represented by a node agent, too, to ensure interaction and information exchange between mobile agents and the sensor node. All interacting agents must comply about the data structures and types, fixed at design time.

A scheduler is responsible to map incoming messages (M), holding information about the agent class and agent state, to agent executions frames (A), passed to an agent virtual machine by using queues. Finally, the scheduler transforms the state of finished agents ready for migration to messages and passes them to the communication unit by using queues, too.

The design process is shown in Fig. **3**, and requires the textual specification of the agent on algorithmic and programming level (left part). This specification is transformed into an abstract agent finite state machine (FSM) model, the state memory layout (middle part), and the message structure. Finally the microchip implementation on RTL (right part) can be synthesized from this intermediate representation, creating an application specific processing environment for mobile agents.

Fig. 3. **Design process for state-machine based agent implementation**



4. Approach II: Programmable Multi-Agent Processing Architecture using Code Morphing

Multi-agent systems providing migration mobility using code morphing can help to reduce the communication cost in a distributed system **[4]**. The second proposed hardware architecture and run-time environment is specifically designed towards the implementation of mobile agents by using dynamic code morphing under the constraints of low-power consumption and high component miniaturization. Code morphing is the ability of a program to modify its own program code to reflect state changes and embedding computational results.

The advantage of this distributed computation model using code morphing is the computational independence of each node and the eliminated neces-

sity for nodes to comply with previously defined common data types and structures as well as message formats. Computing nodes perform local computations by executing code and cooperate by distributing modified code (carrying embedded information) to execute a global task.

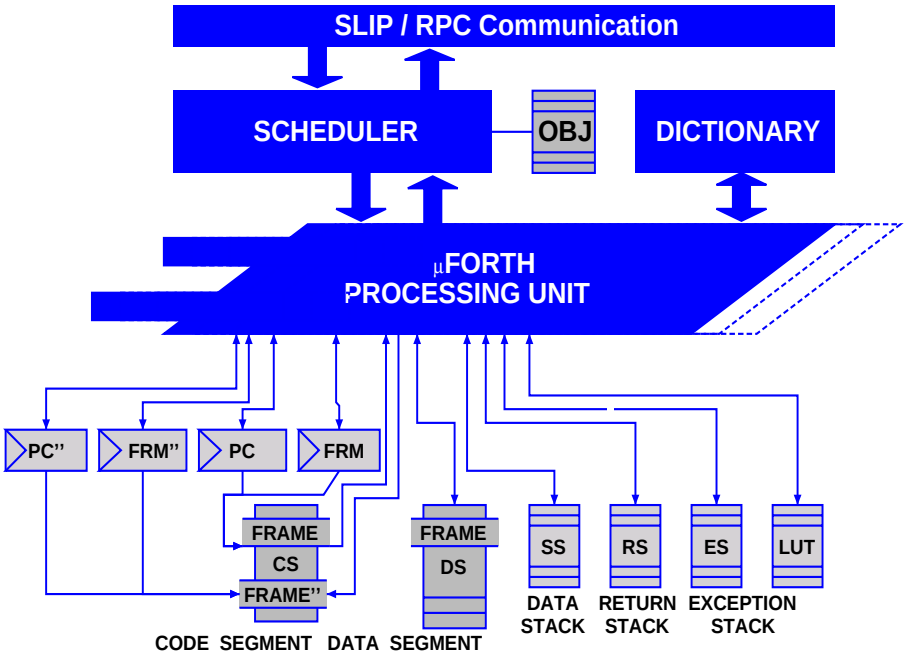
It uses a modified and extended version of FORTH as the programming language for agent programs. FORTH is a stack-based interpreted language whose source code is extremely compact. Furthermore, FORTH is extensible, that is new language constructs (called words, zero-operand functions) can be defined on the fly by its users. A FORTH program contains built-in core instructions directly executed by the FORTH processing unit and user-defined high-level word and object definitions that are added to and looked up from a dictionary data structure. This dictionary plays a central role in the implementation of distributed systems and mobile agents. Words can be added, updated, and removed (forgotten), controlled by the FORTH program itself. User-defined words are composed of a sequence of words.

The principal system architecture of one μ**FORTH processing unit** (PU, part of the node runtime environment) is shown in Fig. **4**. A complete runtime unit consists of a communication system with the smart routing protocol stack SLIP, one or more μFORTH processing units with a code morphing engine, resource management, code relocation and dictionary management, and a scheduler managing program execution and distribution, that are normally part of an operating system, which does not exist here. A μFORTH processing unit initially waits for a frame (a FORTH program) to be executed. During program execution, the μFORTH processing unit interacts with the scheduler to perform program forking, frame propagation, program termination, object creation (allocation), and object modification.

The **scheduler** is the bridge between a set of locally parallel executing μFORTH processing units, and the communication system, a remote procedure call (RPC) interface layered above SLIP, providing fault-tolerant message-based communication system used to transfer messages (con-

taining code) between nodes using smart XY delta-distance vector routing.

Fig. 4. **Mobile-agent run-time architecture providing code morphing, consisting of FORTH data processing units, shared memory and objects, dictionary, scheduler, and communication (PC: Program Counter, FR*M*: Frame Pointer, OBJ: Object Pool, CS: Code, LUT: Lookup Table, *S: Stack).**



The simple FORTH instruction format is an appropriate starting point for code morphing, i.e., the ability of a program to modify itself or make a modified copy, mostly as a result of a previously performed computation. Calculation results and a subset of the processing state can be stored directly in the program code, which changes the program behaviour. The standard FORTH core instruction set was extended (see Tab. **1**) and adapted for the implementation of agent migration in mesh networks with two-dimensional grid topology. In our system, a µFORTH program is contained in a contiguous memory fragment, called a frame. A frame can be transferred to and executed on remote nodes and processing units.

The virtual µFORTH machine can execute most of the core words from the FORTH core programming language.

Tab. 1. **μForth extensions for code morphing and agent migration support**

Word	Description
frame `c!`	`SETC`: Sets frame of shadow environment for code morphing.
m1 m2 >>c	`COPYC`: Switches to morphing state: Transfers code from program frame between two markers m1 and m2 into shadow frame (including markers)
`>c`	`TOC`: Copy next word from program frame into shadow frame
n `s>c`	`STOC`: Pop *n* data value(s) from stack and store values as word literals in shadow frame
`<m>`	`MARKER`: set a marker position anywhere in a program frame.
`dx dy fork`	Send contents of shadow frame for execution to node relative to actual node. If dx=0 and dy=0, the shadow frame is executed locally and concurrently on a different FORTH processing unit.

All architecture parts of the multiprocessor-FORTH node, including SLIP communication, μFORTH processing units, scheduler, dictionary and relocation support, are mapped entirely to **hardware** multi-RT level and a single SoC design using the ConPro compiler **[3]**. The resource demand depends on the choice of design parameters and is in the range of 1M - 3M equivalent gates (in terms of FPGA architectures). The entire design is partitioned into 43 concurrently executed sequential processes, communicating by using 24 queues, 13 mutex, 8 semaphores, 52 RAM blocks, 59 shared registers, and 11 timers.

5. Robust and Reliable Communication for Mobile Agent Systems

Most actual work in communication focusses on wireless networks **[9]**. But sensorial materials and highly integrated robotics systems require basically wired networks **[6]**. The Scalable Local Intranet Protocol (SLIP) and a communication controller design was developed for message based robust communication in low-resource and low-power sensor networks **[2]**. To meet the goal of miniaturization and low-power capability, the protocol must be capable of implementation in SoC and RTL designs, and adaptable to

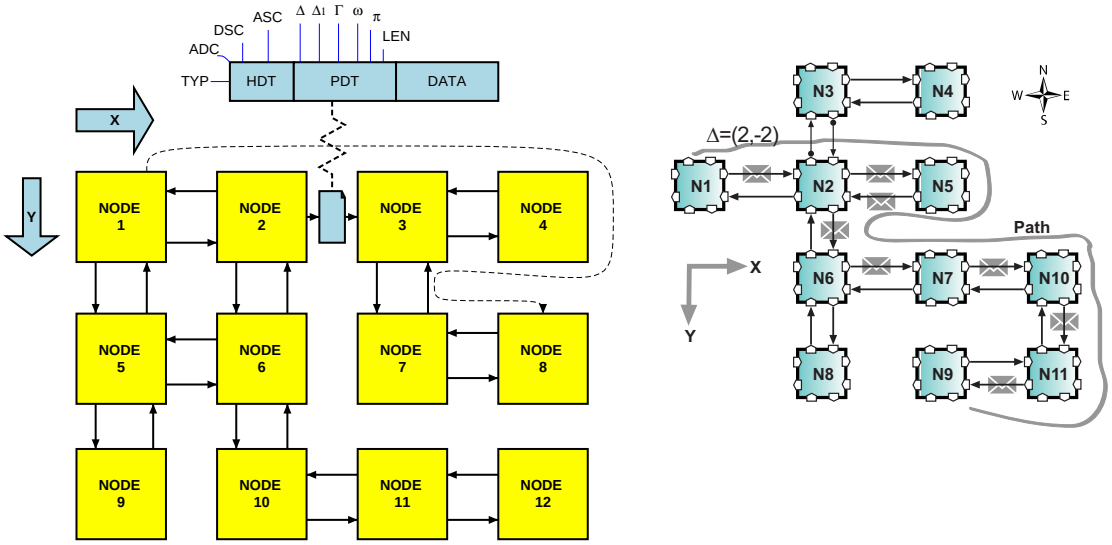
local communication requirements.

5.1. Reliable Communication Protocol SLIP

SLIP is scalable with respect to network size (address size class (ASC), ranging from 4 to 16 bit), maximal data payload (data size class (DSC), ranging from 4 to 16 bit length) and the network topology dimension size (address dimension class (ADC), ranging from 1 to 4).

Network nodes are connected using (serial) point-to-point links, and they are arranged along different metric axes of different geometrical dimensions: a one-dimensional network (ADC=1) implements chains and rings, a two-dimensional network (ADC=2) can implement mesh grids, a three-dimensional (ADC=3) can implement cubes, and so on. Both incomplete (missing links) and irregular networks (with missing nodes and links) are supported for each dimension class, shown in Fig. **5**.

Fig. 5. **Message based communication in two-dimensional networks using delta-distance vector routing. Networks with incomplete (missing links) and irregular (missing nodes) topologies are supported by using smart routing routes.**



The main problem in message-based communication is routing and thus addressing of nodes. Absolute and unique addressing of nodes in a high-density sensor network is not suitable. An alternative routing strategy is delta-distance routing, used by SLIP. A delta-distance vector Δ specifies the way from the source to a destination node counting the number of node hops for each dimension.

A message packet contains a header descriptor specifying the type of the

packet and the scalable parameters ASC, DSC, and ADC, shown in Tab. **2**. The network adress dimension ADC and the size class ASC reflect the network topology, the data size class DSC the data payload. There are two different main message types: requests and replies.

A packet descriptor follows the header descriptor, containing: the actual delta-vector Δ, the original delta-vector Δ^0, a preferred routing direction ω, an application layer port π, a backward-propagation vector Γ, and the length of the following data part. The total bit length of the packet header depends on the {ASC, DSC, ADC} scalable parameter tuple setting, which optimises application specific the overhead and energy efficiency (spatial & temporal). Each time a packet is forwarded (routed) in some direction, the delta-vector is decreased (magnitude) in the respective dimension entry. For example, routing in x-direction results in: $\Delta_1=\Delta_1-1$. A message has reached the destination iff $\Delta=0$ and can be delivered to the application port π. There are different smart routing rules, applied in order showed below until the packet can be routed (or discarded), shown in Alg. **1**. First the normal XY routing is tried, where the packet is routed in each direction one after another with the goal to minimize the delta count of each particular direction. If this is not possible (due to missing connectivity), the packet is tried to send to the opposite direction, marked in the gamma entry Γ part of the message packet descriptor. Opposite routing is used to escape small area traps, backward routing is used to escape large area traps or to send the packet back to the source node (packet not deliverable). The routing decision is based on the actual message entries $\{\Delta,\Gamma,\omega\}$ and achieves adaptive routing reflecting the actual network topology and the path the message already had travelled, including back-end traps, resulting in alternative paths by choosing different routing directions.

Tab. 2. **SLIP message format (HDT: header descriptor, PDT: packet descriptor,)**

Entry	Size [bits]	Description
HDT:ADC	2	Address Dimension Class
HDT:ASC	2	Address Size Class
HDT:DSC	2	Data Size Class
HDT:TYP	2	Message type = {Request, Reply, Alive, Acknowledge}
PDT:Δ	Num(ADC)*Bits(ASC)	Actual delta vector

Entry	Size [bits]	Description
PDT:Δ^0	Num(ADC)*Bits(ASC)	Original delta vector
PDT:Γ	2*Num(ADC)	Backward propagation vector
PDT:ω	Bits(ADC)	Preferred routing direction
PDT:π	Bits(ASC)	Application layer port
PDT:LEN	Bits(DSC)	Length of packet
DATA	LEN*Bits(DSC)	Data

A message is only send to a neighbour node using the particular link iff the connection to the neighbour node was negotiated and is fully operational concerning the sending and the receiving of messages to and from the neighbour node. For this purpose, the communication controller sends periodically ALIVE messages to all direct surrounding nodes and waits for ACKNOWLEDGE messages send back from the neighbour node to check the state of a connection. Non-existing nodes can be detected this way, too.

Alg. 1. Smart Routing Protocol SLIP (simplified)

```
M: Message(Δ,Δ⁰,Γ,ω,π,Len,Data)

PRO smart_route(M):
  IF Δ=0 THEN DELIVER(M,π) ELSE
  TRY route_normal(M) ELSE
  TRY route_opposite(M) ELSE
  TRY route_backward(M) ELSE DISCARD(M);

PRO route_normal(M):
  FORONE δi ∈ Δ TRY minimize δi:
   route(Δ,M) WITH δi:=(δi+1)|δi<0 ∨(δi−1)|δi>0;

PRO route_opposite(M):
  FORONE δi ∈ Δ TRY minimize δi:
    route(Δ,M) WITH δi:=(δi−1)|δi<0∨(δi+1)|δi>0;

PRO route_backward(M):
   SEND M (received from direction δi)
     back to direction -δi WITH Γi=-δi/|δi|;
```

The hardware implementation (using Conpro and standard cell ASIC synthesis) requires about 244k gates, 15k FF ≅ 2.5mm^2 assuming ASIC standard cell technology 0.18µm. The design is partitioned on programming

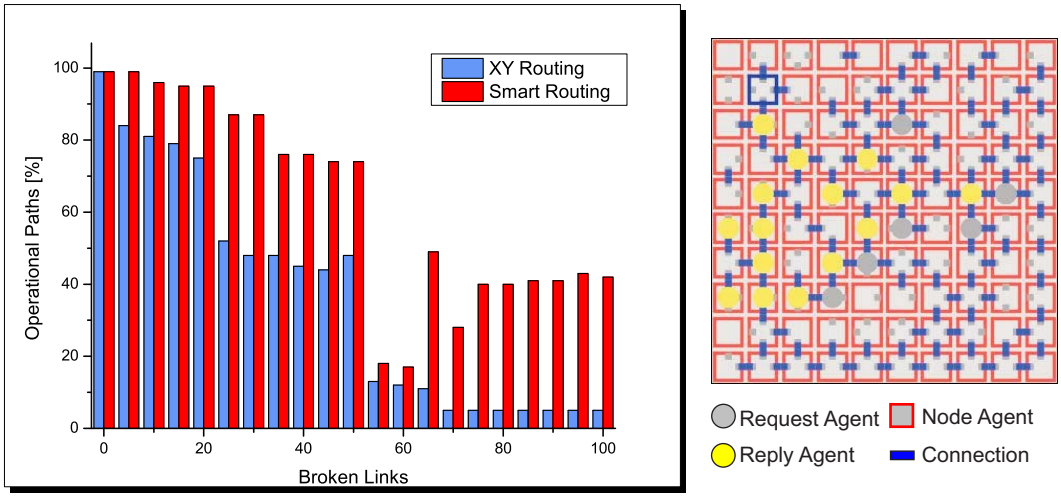
level in 34 processes, communicating by using 16 queues.

5.2. Robustness and Stability Analysis

A simulation of a sensor network consisting nodes arranged in a two-dimensional matrix with 10 rows and 10 columns was performed by using a multi-agent model. Messages and sensor nodes were modelled with agents. A comparison of XY and smart routing using the routing rules introduced in Alg. **1** is shown in Fig. **6**. The diagram shows the analysis results of operational paths depending on the number of link failures. A path is operational (reachable) iff a node (device under test), for example node at position *(2,2)*, can deliver a request message to a destination node at position *(x,y)* with $x \neq 2 \vee y \neq 2$, and a reply can be delivered back to the requesting node. A failure of a specific link and node results in a broken connection between two nodes. The right image in Fig. **6** shows an incomplete network with 100 broken links.

With traditional XY routing there is a strong decrease of operational paths, from a specific node (DUT) to any other node, if the number of broken links increases. Using smart routing increases the number of operational paths significantly, especially for considerable damaged networks, up to 50% compared with XY routing providing only 5% reachable paths any more.

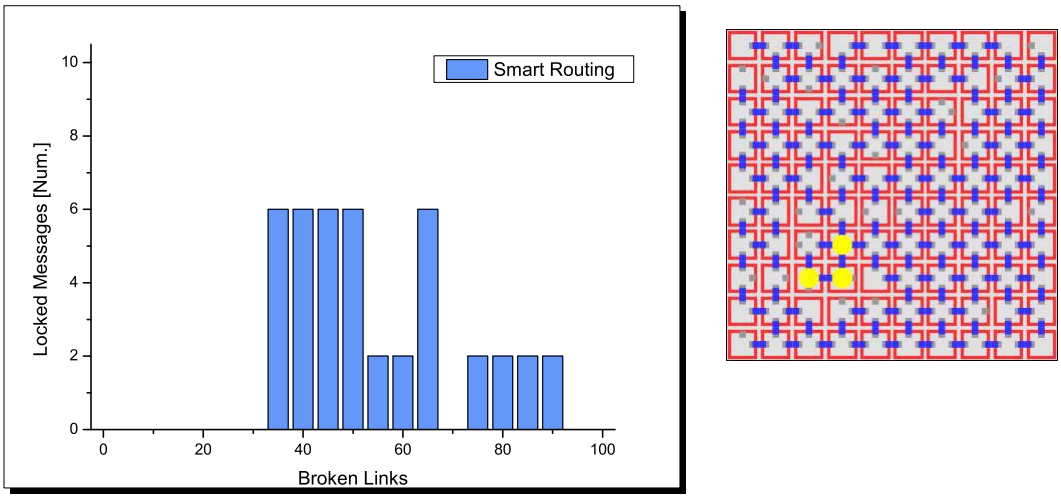
Fig. 6. **Robustness analysis with results obtained from simulation (left) and snapshot of sensor network (right)**



Results from stability analysis shown in Fig. **7** point out unstable behaviour under some particular network topologies. Though most situations are live

lock free, there are some live locked messages circulating for ever in some isolated traps, shown for example in the snapshot on the right side of this figure.

Fig. 7. **Stability analysis (live locked messages, left) and snapshot of sensor network with trapped messages (right)**



6. Case Study I: Energy Management in Sensor Networks

Global energy management and distribution in sensor networks introduces the first application for distributed computing using agents. Energy management in sensor network can take place:

Locally

- **At design time**: low-power, application specific single System-On-Chip design
- **At run-time**: computation on demand, parametrization of algorithms with cost-feedback analysis, control of duty-cycle of computation and sleep mode

Globally

- Distribution and collection of energy between nodes (demonstrated by simulation and experiment in **[11]**)
- Energy Management by exploring and exploiting the neighbourhood of nodes
- The data processing system can use the communication unit to

transfer data (D) and superposed **energy** (E).

6.1. Implementation of Smart Energy Management with Agents

For the following smart energy management (SEM) implementation it is assumed that nodes are supplied by local energy sources, for example by using energy harvesting techniques. Each node is capable of storing energy in a local energy storage. Additionally, nodes can be supplied by energy from neighbour nodes, transferred using the communication system.

Nodes having an energy level below a threshold E_T can send out mobile help agents. Help agents explore the neighbourhood of the requesting node. The state of an agent is stored and transferred in messages. A message can carry energy E, too. The sensor node itself contains a (non-mobile) node agent performing local energy management.

There are four different agent classes:

NODE
The node agent implements inter-agent communication and local energy management. In the case the local stored energy is below a critical threshold, it sends out `HELP` agents to the surrounding area.

HELP
The goal of this agent is to find a good node with local energy above a certain threshold $E>E_{T1}$. If such a node was found, the `HELP` agent stays on this good node, and sends periodically `DELIVER` agents.

DELIVER
This agent has the goal to return energy to the requesting bad node with help-on-way behaviour. Help-on-way behaviour supplies bad nodes found on the return path before the final requesting node was reached. This ensures a path from the source to the destination node with fully operational nodes having enough energy stored for energy distribution and propagation.

DISTRIBUTE
Very good nodes with an energy level above a threshold $E>>E_{T1}$ can distribute energy to surrounding neighbourhood with the goal to find bad nodes.

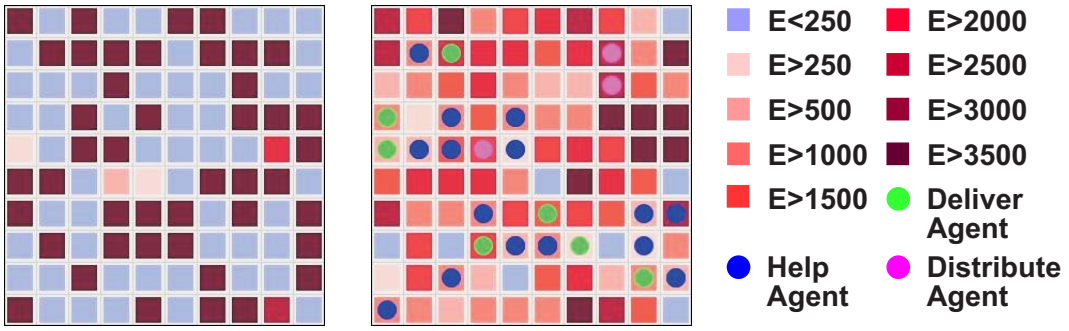
The agent behaviour was implemented using the state-based agent model introduced in section **2.**. The agent state machine implementing the behaviour consists of only nine control states and nine state variables, preferred

for the low-resource state machine approach.

6.2. Simulation Results

A simulation was performed to carry out the suitability of the proposed smart energy management approach using agents and the SeSAm Multi-Agent System (MAS) simulation framework **[10]**. It is assumed that nodes are charged randomly with energy from a local source and discharged by activity. Though there are nodes collecting enough energy to be always operational, there are nodes collecting not enough energy to be operational. Without energy management, there are about 60% non-operational bad nodes (E<250 energy units). Using smart energy management with agents, the averaged fraction of bad nodes is below 10%, and permanently below 2%, shown in Fig. **8**!

Fig. 8. **Left: Energy distribution population map without SEM, Right: with SEM (after 10000 time steps)**



Because energy transfer is not lossless the simulation contains simple efficiency considerations. Actually the spatial agent exploration is initially random, with limited memory of already visited nodes. This simple exploration algorithm leeds to a low overall system efficiency below 10% (fraction of distributed energy compared with total harvested energy). Future investigations must improve the exploration and distribution strategy with the goal to meet higher energy efficiency.

7. Case Study II: Distributed Sobel Filter

Originally applied in image processing and computer vision, the Sobel filter is a edge detection filter, which can also be used for crack detection in sensorial materials. As an example, a simple technical structure such as a met-

al plate can be considered, on which a high number of strain-gauge equipped sensor nodes have been distributed in a grid network **[6]**. The development and growth of cracks due to overloading situations or material fatigue changes the way load-induced strains propagate in the material. These structural defects can appear as edges in the two-dimensional sensor data field and can, in principle, be detected in a convolution process with a Sobel operator. It is assumed that each network node can read data from only one local sensor, i.e., an accumulated centralized view of the structural state does not exist.

A Sobel operator matrix S is used for a neighbourhood operation on the original image A composed of sensor data, for example a 4x4 matrix, and each matrix entry represents a node in the sensor network. There are two different operators, each for a different direction sensitivity (x/y), shown in eq. **1**.

Eq. 1. Sobel operator definition and image convolution

$$S^x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}, S^y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}, \begin{matrix} G^x = S^x \bullet A \\ G^y = S^y \bullet A \end{matrix}$$

Eq. 2. Partial calculation of G-elements containing actual image value $a_{x,y}$ at node (x,y)

$$\forall (i, j) \in \{x-1, x, x+1\} \times \{y-1, y, y+1\} \text{ DO}$$

$$g_{i,j} \leftarrow g_{i,j} + s_{2-i+x,\,2-j+y} \cdot a_{x,y}$$

A µFORTH program executed on the proposed architecture in section **4.** implements a mobile agent moving and migrating through the area of interest and performing the image processing. Initially, a master agent is sent to the upper left corner node, sampling data and performing a partial image convolution. Each node calculation carries out partial calculation of sum terms of $g_{i,j}$ elements containing only the local sensor data $a_{x,y}$, updating $g_{i,j}$ with pseudo-code shown in Eq. **2** (assuming array index numbers within range 1...N).

The FORTH program is compiled to a word-code machine program consisting of about 600 words by using a µFORTH compiler. A word requires 2 byte of memory storage. Due to the low code size the entire program can be fit in one message.

To summarize, this case study showed the implementation of a complex algorithm with multi-agent systems executable on single microchip nodes using code morphing for carrying computational results and the suitability of the proposed runtime environment approach II.

8. Comparison and Conclusions

Tab. 3. Comparison of the two data processing approaches for mobile agents

	Approach I. State-Machine	Approach II. Code morphing
Agent behaviour is	fixed, nodes must comply with previously defined common data types and structures as well as message formats.	not fixed, can change dynamically, and nodes do not require knowledge of data structures and types in advance.
Functional behaviour is implemented	statically in local data processing machine.	dynamically in programming code, which can be modified by the program itself.
Implementation in	Hardware, single chip	Hardware, single chip
Agent state is kept in	data storage	code, stacks, and data storage
Message size depends on	data complexity and size, data and control state, but is independent of code complexity.	code complexity and size, but is independent of state.
Hardware resources are	small (< 1M eq. logic gates including storage)	large (> 1M-3M eq. logic gates including storage)
Storage resources are	small (< 5000 register cells)	large (> 10000 register cells)
Speed is	high (1-2 clock cycles per statement)	medium (5-20 clock cycles per core word instruction)
Power consumption is	low and depends on code complexity.	medium and is independent of code complexity.

Table **3** compares both run-time architectures and agent implementations. Both approaches allow the implementation of agent mobility and processing on hardware single-chip level. Flexibility and design time versus resource requirements is the main difference. The state-machine based approach I with fixed and hard implemented functional agent behaviour is well suited for a small set of different agents with simple algorithm complexity, whereas the code morphing approach II is suited for a larger set of different agents with higher algorithm complexity.

A program-controlled approach II is less power efficient and requires more resources, but provides a higher lever of implementation and design freedom. The code morphing approach II reduces communication complexity.

One main issue addressed in the design of multi-agent systems is cooperation and communication of agents, and to ensure how can agents understand each other. Message based systems require some kind of communication language. Each node, which processes agents must comply about well known data structures used for inter-agent communication, fixed at design time. There are only limited capabilities to handle data type inconsistency and the non-availability of expected data. In contrast, the code based approach II uses named code and data words resolved by a dictionary, with a well known interface, and the capability to check and handle type inconsistency. The hardware implementation of the dictionary and the operational interface produces a fairly high overhead of the resources compared with the traditional shared data approach using memory references (as used in the state-machine-based approach I).

The smart routing protocol must be modified to overcome the message live lock issues and to improve stability by preserving reliability and robustness. Future experimental investigations using real sensor networks with different classes of data processing algorithms should clarify the advantages and disadvantages of both approaches.

9. References

[1] M. Wooldridge, *An Introduction to MultiAgent Systems*, Wiley (2009)

[2] S. Bosse and D. Lehmhus, *Smart Communication in a Wired Sensor- and Actuator-Network of a Modular Robot Actuator System Using a Hop-Protocol with Delta-Routing*, Proceedings of Smart Systems Integration Conference, Como, Italy, 23-24.3.2010 (2010)

[3] S. Bosse, *Hardware-Software-Co-Design of Parallel and Distributed Systems Using a unique Behavioural Programming and Multi-Process Model with High-Level Synthesis*, Proceedings of the SPIE Microtechnologies 2011 Conference, 18.4.-20.4.2011, Prague, Session EMT 102 VLSI Circuits and Systems

[4] S.Bosse, F. Pantke, and F. Kirchner, *Distributed Computing in Sensor Networks Using Multi-Agent Systems and Code Morphing*, IC-AISC Conference, Prague, 2012

[5] A. Kent and J. G. Williams (Eds.), *Mobile Agents*, Encyclopedia for

Computer Science and Technology, New York: M. Dekker Inc., 1998

[6] F. Pantke, S.Bosse, D. Lehmhus, and M. Lawo, *An Artificial Intelligence Approach Towards Sensorial Materials*, Future Computing Conference, 2011

[7] H. Peine and T. Stolpmann, *The Architecture of the Ara Platform for Mobile Agents*, MA '97 Proceedings of the First International Workshop on Mobile Agents,Springer-Verlag London, 1997

[8] A.I. Wang, C.F. Sørensen, and E. Indal., *A Mobile Agent Architecture for Heterogeneous Devices*, Wireless and Optical Communications, 2003

[9] K. Römer and F. Mattern, *The design space of wireless sensor networks*, IEEE Wireless Communications 11 (2004), Dezember, Nr. 6, p. 54-61

[10] F. Klügel, *The Multi-Agent Simulation Environment SeSAm*, In: H. Kleine Büning (Ed.): Proceedings of Workshop "Simulation in Knowledge-based Systems", Paderborn, April 1998

[11] A. Kansal, J. Hsu, S. Zahedi, and M. B. Srivastava, *Power management in energy harvesting sensor networks,* ACM Transactions on Embedded Computing Systems, vol. 6, no. 4, p. 32-es, 2007.