# Chapter 6

## Networking and Communication

*The Sensor-network Level*

A smart sensor network is composed of a set of smart sensor nodes, already discussed in Chapter **5**, arranged in some kind of a regular communication structure (the network topology). A smart sensor node operates basically on locally acquired sensor data, whereas a smart sensor network can be seen a big virtual machine operating on global sensor data, possible segmented. The network composition requires different processing and communication layers:

- Communication Hardware - The Physical Layer
- Network Communication Protocols for Messaging
- Message Passing and Routing in arbitrary network topologies
- Distributed and Cloud Computing addressing large-scale networks, distributed storage, and distributed data processing.
- Agent-based Computing addressing autonomous and reliable communication and data processing in large-scale networks with a unified programming and exeuction model.

### 6.1 Communication Hardware [Tiedemann]

When designing "Networking and Communication" bottom-up, the hardware to be used for communication becomes the starting point. In case of material-integrated intelligent systems special demands and limits are given by the applications. However, there is still a broad spectrum of different communication hardware types, and even standards, that could be applicable.

 ISBN 978-3-527-33606-7

As it is true for other components of material-integrated intelligent systems, too, a source of potential communication hardware solutions is given in the wide field of embedded systems. Therefore, this section gives an overview of communication hardware options used in embedded applications today.

In Sec. **6.1.2** an overview of potential requirements for communication hardware in material-integrated intelligent systems is given. A list of requirements should be made for the specific target application as a first design step. These requirements determine the physical communication media and the potential communication options and standards being eligible for the further communication design process.

The sections **6.1.3**, **6.1.4**, **6.1.5**, and **6.1.6** describe the different types and especially the respective standards of communication hardware together with example applications.

A final summary is given in Sec. **6.1.7** and a short motivation showing communication hardware in specific applications follows in the next section. Not covered is CPU communication hardware like standard buses for CPU-peripheral or CPU-memory data transfer. Furthermore, only digital communication is dealt with here. Analog properties are described only where necessary and as physical side effects of digital communication.

### *6.1.1 Communication Hardware in Their Applications*

Communication can be applied in material-integrated intelligent systems in various ways. One field is the integration in textiles (sometimes called smart fabrics), e.g., with electronics for health monitoring. An early overview of research projects and applications is given by Lymberis and Paradiso **[LYM08]**. They see smart fabrics and interactive textiles (SFIT) being part of an interactive communication network. Embedded into textile are potentially sensors, actuators, computing, and power sources - needing communication within the textile.

For the application field of firefighters van Torre et al. presented a communication solution using textile antennas **[TOR11]**. Communication is used here for data exchange with external, potentially non-embedded devices. In this case radio frequency (RF) transmission is used as communication media. RF communication needs two specific problems to be solved. One is the integration of antennas the other is the power consumption which is usually very limited in material-integrated intelligent systems. Magno et al. review different power management methodologies for wireless sensor networks **[MAG14]**. This article is outlined in Sec. **6.1.5**.

Finally, a completely different way of communication is presented by Jacobs (2005) with the research project "Reach" (as part of the interdisciplinary research platform "IT+Textiles") **[JAC05][ITT16]**. Here, embedded optoelectronics is used to generate (potentially dynamical) patterns on textiles for textile-external, inter-human communication.

 ISBN 978-3-527-33606-7

### *6.1.2 Requirements for Embedded Communication Hardware*

The requirements for communication hardware in material-integrated intelligent systems can be derived from the requirements generally given for embedded systems. The latter are summarized by Marwedel **[MAR11]**.

The requirements for specific material-integrated intelligent systems\ have to be identified depending on the respective applications. As a basis to define requirements, metrics are needed. Examples for such metrics used by Marwedel are

- data throughput rate (including delays needed for the communication), measured, e.g., in bits per second (bit/s, bps),
- latency (delay between communication start at the sender side and receiving of the first data at the receiver -- notice that latency and throughput can both be high at the same time), measured, e.g., in s,
- noise/interference parameters, e.g., error rate in percent,
- power consumption,
- dimensions and/or weight, and
- costs.

Important requirements of communication as indicated by Marwedel **[MAR11]** are:

- **Efficiency:** An application can give certain (and strict) efficiency requirements in different respect. Efficiency is determined as a ratio of output (e.g., data throughput rate) by needed resource (e.g., power consumption, weight, costs). Typically for embedded systems are dimension, power, and weight requirements since they are usually limited by the structures they are embedded in. To improve the efficiency for the communication, the network topology can be selected appropriately (e.g., using bus instead of star topology, see Sec. **6.2**). On the communication hardware level an appropriate standard can be chosen (e.g., serial instead of parallel wired connection, wireless instead of wired).

- **Data throughput** and/or **latency:** In most cases the application gives specific requirements for a minimum data throughput. These requirements can vary a lot. In some applications the minimum data-throughput and/or the maximum latency are fixed limits and need to be respected. Such cases are sometimes called real-time capability.
- **Polling vs. event-triggered data transfer:** The communication protocol and/or the application can give specific requirements regarding the communication triggering. One option is to let one system (sometimes called "master") ask the others (sometimes referred to as "slaves") if communication is needed i.e. if they need data or if they have data to be send. The slaves can not initiate communication in

 ISBN 978-3-527-33606-7

this "polling" called mode. Another option is event-triggered data transfer. In this case an event (usually application-related) initiates communication. Here, also slaves can start the communication.

- **Error protection:** In many applications communication can be disturbed leading to single or multiple bit errors. In cases where the application can not tolerate such errors countermeasures need to be taken. One usual method is the transfer of additional special data leading to redundancy in the transferred data. By error detection or error correction codes (EDC/ECC) the occurrence of an error can be detected. In some applications it might be needed to introduce error protection measures on the protocol level as well (e.g., to trigger a repetition of data transfers or an interruption of current processes).
- **Robustness regarding environmental conditions:** Especially in embedded applications the communication sometimes needs to cope with harsh environmental conditions. Examples are extreme temperature ranges, radiation, pressure, chemicals, etc. the communication hardware needs to withstand. The most often occurring environmentally caused requirement is the robustness against electromagnetic interference (EMI). EMI can cause disturbances of the transferred data and, therefore, lead to bit errors (see error protection above).
- **Security:** Depending on the application, the data to be transferred might be classified as sensitive. In this case special attention should be drawn on the environment and to what extent a third party could be interested in obtaining or modifying the data. To avoid this, security measures like encryption and/or authentication can be used.
- **Maintainability:** Finally, a maintainability of the communication software or hardware can be required. Needed might be the option to repair or exchange the communication hard-/software itself (e.g., to improve the data throughput). Furthermore, it might be desired to add communication means to enable *system* maintenance, e.g., software updates or debugging.

In the case of material-integrated intelligent systems the general requirements can be weighted and have to be extended depending on the specific application.

### *6.1.3 Overview of Physical Communication Classes*

To get an overview of the different types of communication at first the layers involved have to be understood. As defined in **[MAR11]** communication is set up in so-called channels. Channel is an abstract term which is linked to a) an abstract relation between one sender (component) and one or multiple recipients (components) and b) linked to communication properties. Such communication properties can be data throughput or noise/ interference parameters.

 ISBN 978-3-527-33606-7

The physical basis the communication in a channel is based on is called a medium. Important classes of media are wireless electromagnetic transfer, wireless optical transfer, connection via optical wave-guides, and transport of charges in electrical wires.

*Electrical wires*

The latter class is the communication medium used in the vast majority of communication cases. It can be further subdivided into a) a single-ended (SE) and b) differential signal or differential pair (DS/DP) called connections. SE connections use usually one reference connection between sender and recipient(s) (called ground) and an arbitrary number of additional wires transferring usually one data bit per point in time per wire (see Fig. **6.1**).

The names of the latter vary a lot and depend on the application. In the following descriptions they are called data lines ($D_0$, $D_1$). In most cases bit data ("0" or "1") are defined by voltage thresholds to be measured between ground and a data line. The thresholds are fixed in communication standards. The sender applies a voltage (e.g., 3.3V, above the given threshold) to send a "1" and another voltage (e.g., 0V, below another given threshold) to send a "0". The recipient(s) measures the voltage between their wire ends of ground and data line to identify a bit value of "0" or "1" respectively. In some standards the wires are connected via a resistor at (one of) the recipient(s).

In this case, the sender applies a voltage between two wires to generate a constant current through the first wire, the recipient's resistor, and back through the second wire. In this case, "0" and "1" are defined using current thresholds. One advantage of using current compared to voltage is an independence of the wire length (within limits defined by the sender driving electronics and usually defined by the standards.

 ISBN 978-3-527-33606-7

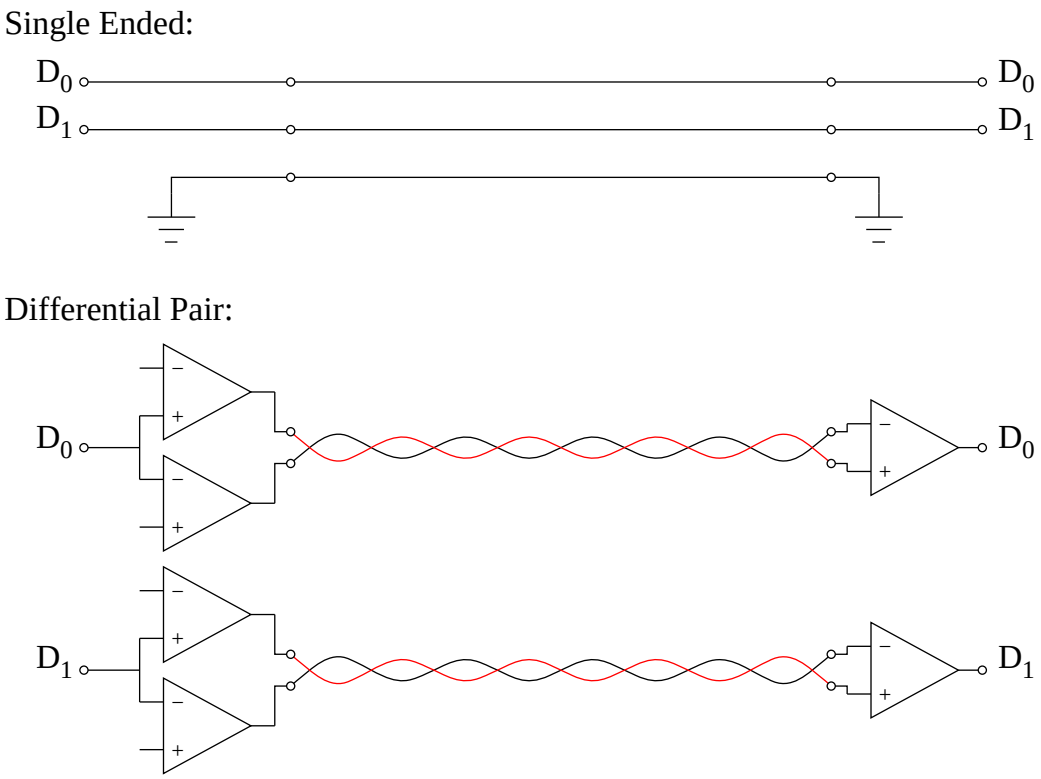



**Fig. 6.1** *In wired electrical communication single ended (SE) or differential pair (DP) connections can be chosen. SE has the advantage of a simple and less expensive set up while DP can realize more robust and, thus, higher data throughput or longer distance connections.In the figure two bits $D_0$ , $D_1$ are transferred in parallel.*

In DP connections always pairs of wires are used with one dedicated positive (+) and one dedicated negative (-) wire. The sender applies a voltage to both wires.

Often, a ground connection is given, too, and standards limit the voltage between ground and positive/negative wire. Again, bit values "0"/"1" can be defined via voltage thresholds (now measuring between positive and negative wire. The sender applies, e.g., 3.3V between ground and positive wire and 0V between ground and negative wire to send a "1". For a "0" bit value the sender applies 0V between ground and negative wire and 3.3V between ground and negative wire. Consequently, the recipient measures +3.3V between negative and positive wire in the former case and -3.3V in the latter situation. Hence, the difference between both measurements (bits) is 6.6V now.

One benefit is an increased robustness against \elm ally induced disturbances. This is especially the case for disturbances which induce an erroneous offset voltage to the almost same degree on both positive and negative wire. Thus, this constant component (direct

 ISBN 978-3-527-33606-7

component, DC) part applied to both wires does not change the difference signal measured between both wires.

Additionally, if possible both wires are twisted to equalize the electro-magnetic influence for both. One application field of differential signaling are environments with large electro-magnetic interference (EMI), thus, with high noise levels.

Another application are high-speed connections. Here, the voltage levels the sender applies are lowered compared to standard speed SE connections in the same environment. This is called low-voltage differential signaling (LVDS). The advantage of lower voltage levels is a possible reduced switching time between low and high voltage level on the two (positive and negative) wires. This leads to shorter times needed per bit and, therefore, a higher bit rate.

In general, the bit length (duration) can be defined in different ways. Possible are fixed lengths / bit rates, a switching between different (but fixed) lengths / bit rates (e.g., to support low-speed and high-speed communication), or the lengths dependent on the data bit to be transferred (e.g., short period between previous and next edge to transfer a "0", long period for a "1"). Furthermore, in addition to the payload data (determined by the application) often further bits are added to compensate for transfer errors by introduced redundancy or to ensure specific signal conditions (e.g., start/end marker, maximum time spans between edges). See Sec. **6.2** for details.

*Wireless electro-magnetic transfer*

In cases where a wired connection is not possible, e.g., because sender and recipients have a varying and potentially larger distance, data transfer via electro-magnetic waves is chosen most often. This is called radio-frequency (RF) transfer.

When an electrical current flows in a wire an electro-magnetic field is generated circularly surrounding it. Furthermore, changes of such an electromagnetic field induce a voltage in electrically conducting materials.

Therefore, if a sinusoidal electrical signal is applied to a wire, a sinusoidal \elm\ wave of the same frequency is generated. If an (electrically conducting) wire is applied to this field, a sinusoidal electrical signal of the very same frequency is generated in this receiver antenna.

In the most simple (yet still efficient) case a straight wire with one end open (and length matching 1/4 of the wavelength) can be used at the transmission and at the receiving side. This is called a mono-pole antenna. On the transmission side the sinusoidal voltage is applied between the mono-pole antenna and ground (e.g., the Earth). On the receiving side the signal between ground and the receiving mono-pole antenna is measured.

The energy received is much less than the energy spent to generate the electro-magnetic field. However, depending on the transmission power, the antennas, the dielectric medium between the two antennas (usually air), and the chosen frequency the receiving signal can

 ISBN 978-3-527-33606-7

still be measured with limited effort for distances of meters (low-power applications) or distances up to hundreds or thousands of kilometers (high-power on transmission side).

*Further physical communication classes*

Besides the most often used electrically or electro-magnetically based communication methods two other physical media are used in some off-the-shelf communication devices. One is the data transfer using light. Here, a light transmitter (usually a light emitting diode, LED, or a laser diode) emits visible or invisible light and a light receiver transforms receiving light back in electrical signals (usually a photo diode or photo transistor). As in electrical communication, two types of media are used: Either the light is transferred via optical wave-guides (flexible optical fibers or fixed solid media on a carrier) or an optical transfer in air is used between transmitter and receiver.

A third physical communication class is using acoustic data transfer. The transmitter emits mechanical oscillations (depending on the transfer medium usually in the Hz or kHz, subsonic, sonic, or ultrasonic range) and as receiver special microphones or hydrophones can be used. The transfer media is usually air or water.

### *6.1.4 Examples of Wired Communication Hardware*

Wired communication can be subdivided into a) parallel communication with multiple bits being transferred in parallel on multiple wires and b) serial communication with a transfer of one bit per time on one (SE or DP) connection (see Sec. **6.1.3**, Electrical Wires). In the case of material-integrated intelligent systems\ serial communication is often preferred to reduce the effort needed for the usually complex hardware implementation. A parallel wired communication standard is IEEE-488 {renamed "IEEE 488.1, 2004", renamed "IEEE 60488.1:2004", in parallel developed by IEC as IEC-60625-1, short IEC-625, called HP-IB and GPIB, and with extensions IEEE 488.2 / IEEE 60488.2:2004 and IEEE 488.1-2003, short HS-488, and as combined "Dual Logo" IEEE/ IEC standard IEC-60488-1 and IEC-60488-2).

For serial wired communication several standards of a variety of different types have been designed. Very often used in industry and embedded applications are the serial communication interface (SCI) standards RS-232, RS-422, and RS-485. The ANSI EIA/TIA-232 standard is referred to most often by "RS-232" with the current version "ANSI EIA/ TIA-232-F" issued in 1997. It is a point-to-point interface connecting two devices with up to 115,2~kbit/s. The bit rates are not defined in the standard but the given rate is the usual recommended maximum in current implementations. Higher bit rates can sometimes be chosen but may lead to high error rates or even be impractical.

The interfaces defined by standard RS-422 uses DS instead of SE compared to RS-232. This leads to a higher noise immunity and, therefore, can allow longer cable lengths. The second difference between RS-232 and RS-422 is that RS-422 can be used as a bus with up to ten devices instead of an one-to-one connection only. RS-485 extends RS-422 with

 ISBN 978-3-527-33606-7

up to 32 devices and is a superset of RS-422. Further field bus standards are the CAN-Bus, the LIN-Bus, and PROFIBUS.

On the standard computer systems (PC and embedded systems) as serial interface the Universal Serial Bus (USB) is one of the most often used interfaces. It uses one DP (up to USB 2.0) or three DPs (for full-duplex communication in SuperSpeed mode of USB 3.0) and allows transfer rates of up to 10~Gbit/s (USB 3.1).

A second group of communication interface standards are designed for short distance and simple chip-to-chip communication. These are, for example, synchronous serial communication interface standards $I^2C$ ($I^2S$), (Inter-Integrated Circuit, up to 5.0~Mbit/s), SPI (Serial Peripheral Interface, no formal standard, max. frequency device-dependent up to 100~MHz). Several microcontrollers and peripheral integrated circuits (IC) are available which support one or both of the interfaces.

For non-embedded (and especially not material-integrated) communication, Ethernet (100BASE-TX, IEEE 802.3 Clause 24, or GbE 1000BASE-T) is heavily used for communication between computer systems or between sensors and computer systems. However, especially for material-integrated applications the needed hardware is not suited and, thus, 100BASE-TX and 1000BASE-T will not be described here.

However, for use of Ethernet inside a vehicle special solutions needed to be found that might be applicable to material-integrated communication, too. The resulting standard for automotive Ethernet Open Alliance BroadR-Reach$^{TM}$ (OABR) **[OPA16]** focuses on the reliability of the data transmission, as well as on a cost-efficient transmission medium (two-wire line, see Fig. **6.2**). The standard for the physical layer was developed by the industry OPEN Alliance (One-Pair Ether-Net) special interest group (SIG) **[OPA16]** and became IEEE standard (IEEE 802.3bw, 100BASE-T1 with 100 Mbit/s based on BroadR-Reach, 1000BASE-T1 with 1 Gbit/s). So far, 100BASE-T1 Ethernet was designated for the transmission of image data inside the vehicle. The description of further applications (e.g., implementation of a backbone system) is new and subject of current research projects **[TIE16]**.

For the application in material-integrated intelligent systems\ the reliability and especially the transmission medium of a single unshielded twisted pair seems to be interesting.

 ISBN 978-3-527-33606-7

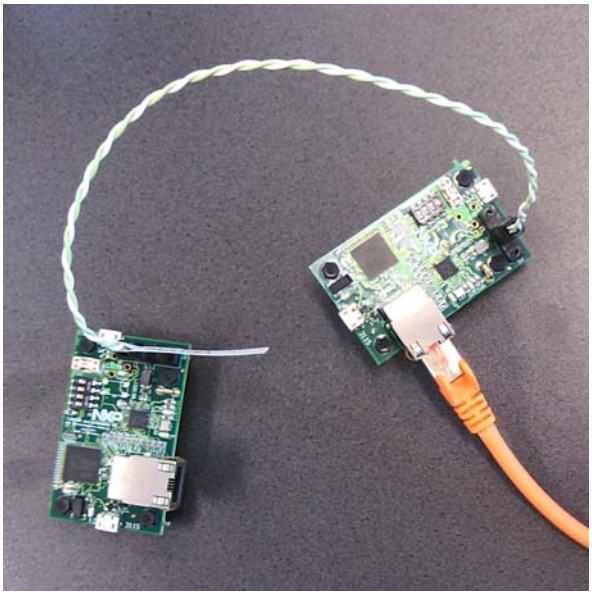


**Fig. 6.2** *Automotive Ethernet is designed for single unshielded twisted pair cables. The photo shows two converters to standard 100BASE-TX interfaces using NXP PHY devices.*

### *6.1.5 Examples of Wireless Communication Hardware*

*Standards*

For wireless (RF) communication several interface standards have been developed. One of the most often used group of standards are the WLAN (Wi-Fi) standards IEEE 802.11. In industry and private use the standards 802.11a, b, g, and n are very common. Especially for the connection of a vehicle to other vehicles (V2V, Car-2-Car) or to the infrastructure (V2I, V2X, Car-2-X), the IEEE 802.11p standard was designed **[IEE16]**. IEEE 802.11p describes the lower layers of dedicated short-range communications (DSRC) to enable wireless access in vehicular environments (WAVE). One main problem of DSRC in Car-2-X communication (addressed in IEEE 802.11p) are the short communication periods, especially when vehicles are passing each other in opposite directions. A higher layer based on IEEE 802.11p is IEEE 1609. One feature designated for the vehicle/traffic application (but interesting for other embedded applications as well) is the service announcement. On the so-called control channel Service Announcement Messages (SAM) can be sent to announce services and service channels. The receiver can switch to the service channel where the contents of the services are transmitted. This is part of the higher layer standard IEEE 1609. However, here are differences between different standards, e.g., between the IEEE 1609 and the European ETSI ITS-G5.

 ISBN 978-3-527-33606-7

IEEE 802.11 standards offer a high data throughput but the power consumption of available devices is relatively high. Therefore, especially for embedded applications other standards like Bluetooth, Bluetooth LE, and ZigBee are often better suited.

Bluetooth is a standard for wireless short-distance communication and was invented by the telecom company Ericsson. IEEE standardized Bluetooth as IEEE 802.15.1 but does not maintain the standard. It is managed by the Bluetooth Special Interest Group (SIG) **[BLU16]**.

ZigBee is a specification developed by the ZigBee Alliance **[ZIG16]** and based on the standard IEEE 802.15.4. It is intended to be simpler and less expensive compared to Bluetooth. The ZigBee Alliance proposes ZigBee as a foundation of the Internet of Things (IoT) **[ZIG16]**.

Bluetooth low energy (or Bluetooth LE, BLE, Version 4.0+, or Bluetooth Smart) was designed by the Bluetooth SIG **[BLU16]**. It is intended to provide a reduced power consumption and reduced cost compared to classic Bluetooth.

*Application examples*

Several research activities related to material-integrated systems and using RF communication were conducted and published.

Kennedy et al. present body-worn e-textiles for the application of extravehicular activity (EVA) suits for space missions. They propose (and focus on the) e-textile antennas and give some more information on the positioning of the body-worn antennas. **[KEN09]**. Hertleer et al. focus on the antenna only and present a flexible patch antenna entirely made out of textile material **[HER07]**. Both target on the communication of the body-worn system with a base station and not on intra-garment communication.

This is also the case for van Torre et al. who present a solution for off-body RF communication for firefighters and other rescue workers using textile antennas. Their proposal uses multiple input/multiple output (MIMO) communication with 2×2 or 4×4 links realized by two dual-polarized antennas, one at the front and one at the back of the firefighter jacket. As they show, the bit error rates (BER) improve with increasing diversity (while keeping the transmitted power equal) **[TOR11]**.

And finally, Magno et al. present a review of power minimization techniques. They target on wireless sensor networks but their results can clearly be relevant for many kinds of embedded and especially material-integrated applications, too. Besides well-established power optimization methods they discuss newer technologies with a focus on wake-up radio receivers **[MAG14]**.

### *6.1.6 Examples of Optical Communication Hardware*

The industry "Infrared Data Association" (IrDA) provides several specifications for wireless infrared communication. They cover different layers and data rates of up to 1~Gbit/s. The physical layer IrPHY (Infrared Physical Layer Specification) defines opti-

 ISBN 978-3-527-33606-7

cal link, modulation, and coding. Different data rates and modulation/coding schemes are defined in IrPHY ranging from SIR (Serial Infrared, up to 115.2 kbit/s) up to Giga-IR with up to 1~Gbit/s. Further layers are IrLAP (representing the data link layer of the OSI model), IrLMP (Infrared Link Management Protocol, implementation of logical channels and service registration), and further but non-mandatory protocols.

The Li-Fi Consortium (founded in 2011) is a non-profit organization, devoted to introducing optical wireless technology (www.lificonsortium.org). After academical exchange of knowledge in the first years they focus on technology and product development. The members of the Li-Fi Consortium developed scenarios and solutions that might be relevant for different embedded communication applications. One example is GigaDock (combination of 10 Gbit/s data transfer and parallel inductive charging for plug-free data and energy transfer), another one is GigaHotspots (ceiling-mounted device communicate with 1 to 5 Gbit/s over distances of 3 to 5~m). Finally, free-space optical communication (FSO) is usually used for data transfer over longer distances of tens of meters up to a few km. However, the devices used for FSO are too large and too heavy for most applications of material-integrated intelligent systems

### *6.1.7 Summary*

To realize communication and networking options for material-integrated intelligent systems one can avail oneself of many existing solutions, most of them fixed in standards. One first design decision to make is the question if a solid, tethered connection should be used (wires or optical fibers) or if a data transfer through air (or water) is preferred.

The second decision is to answer the question of using electrical (or electro-magnetical) transfer or optical data transmission (or in very specific cases acoustic transmission). This point might depend on the environmental conditions, induced noise, needed data rates, and available power. Therefore, a definition of the application's communication requirements (e.g., using the parameters and metrics listed above) is an important prerequisite.

Eventually, most options of off-the-shelf products are available for electrical or electromagnetic communication. Consequently, these standards and products are often the first choice in material-integrated intelligent systems, too. However, with the specific requirements in this field there might be adaptations of the communication hardware and/or software and, thus, violations of the existing standards needed.

 ISBN 978-3-527-33606-7

## 6.2 Networks and Communication Protocols [Bosse]

Material-integrated sensing systems like sensor networks embedded in technical structures entail new data processing and communication architectures. With increasing system complexity increased reliability and robustness of the entire commonly heterogeneous sensing system are required to maintain a specified quality-of-service. Failure is not an exception, it is treated as a part of the design process and the run-time behaviour. Reliable networking and communication is one major part contributing to safe and useful sensing system.

Cloud computing introduces dynamics and scalability of distributed computing with virtualized resources. Though cloud computing is concerned for big data problems, it can be attractive for large-scale sensor networks, especially considering the integration of sensor networks in sensor clouds and the Internet (of Things), discussed in Sec. **6.3**.

A communication network is basically represented with a network connectivity graph (NCG) with nodes (computers, sub-networks, mobile devices, sensors) and edges connecting the nodes, representing the technical communication links, discussed in the previous section. The edges can be directed or not directed (bi-directional communication), and the graph can have cycles resulting in closed paths. A path is a part is a linear list of nodes of the graph, starting with a source node, and ending on a destination node.

Communication is a central part of distributed data processing in sensor networks, influencing performance and operation significantly. A connection between two nodes is usually a uni- or bi-directional (serial) link which is capable to transfer data encapsulated in messages using protocols for coordination and coding. Message passing is the preferred communication method for large scale networks compared with switched networks requiring crossbar or multi-stage Banyan (butterfly) switches. Path finding and routing, the process of forwarding a message in the network with the goal to deliver messages from the sender to the receiver is required in message passing networks, and for economic reasons and the sake of simplicity there are usually no dedicated routers in sensor networks. Thus a sensor or computational node is a service end-point and a message router, too. Traditional computer networks use unique node identification not suitable for (especially loosely or ad-hoc coupled) sensor networks. Path finding algorithms can be classified in those using routing tables managed by each network node and storing information about the network topology (usually limited to the neighbourhood of the respective node), and those not relying on routing tables. Routing tables provide information for path planning in advance. Managing routing tables can be critical in resource constrained and real-time systems, because they allocate a fairly high amount of local storage and require computational and communication activities, which should be minimized in sensor networks. Routing tables can only reflect an outdated view of the network (the network configuration within a certain time interval) and hence dealing with network changes is complicated by using routing tables.

 ISBN 978-3-527-33606-7

### 6.2.1 *Network Topologies and Network of Networks*

The network topology is a specific structure of the network connectivity graph, introducing some regularity in the arrangement of nodes and their connectivity to neighbour nodes. Some common topologies are shown on the left of Fig. **6.3**, and the topology and technology classifications on the right side.

One of the simplest and still widely used topologies are master-slave 1:N star networks. These are mainly used in centralized processing architectures. There is only one connection from each client to a server node (connectivity degree 1, distance 1). The failure of the server is critical. A different situation is shown in the bottom row (b) using a broadcast medium (bus, Ethernet) which connects N individual nodes, with a connectivity degree (N-1) and a maximal distance of 1. Here the failure of the broadcast medium (the interconnect) is critical.

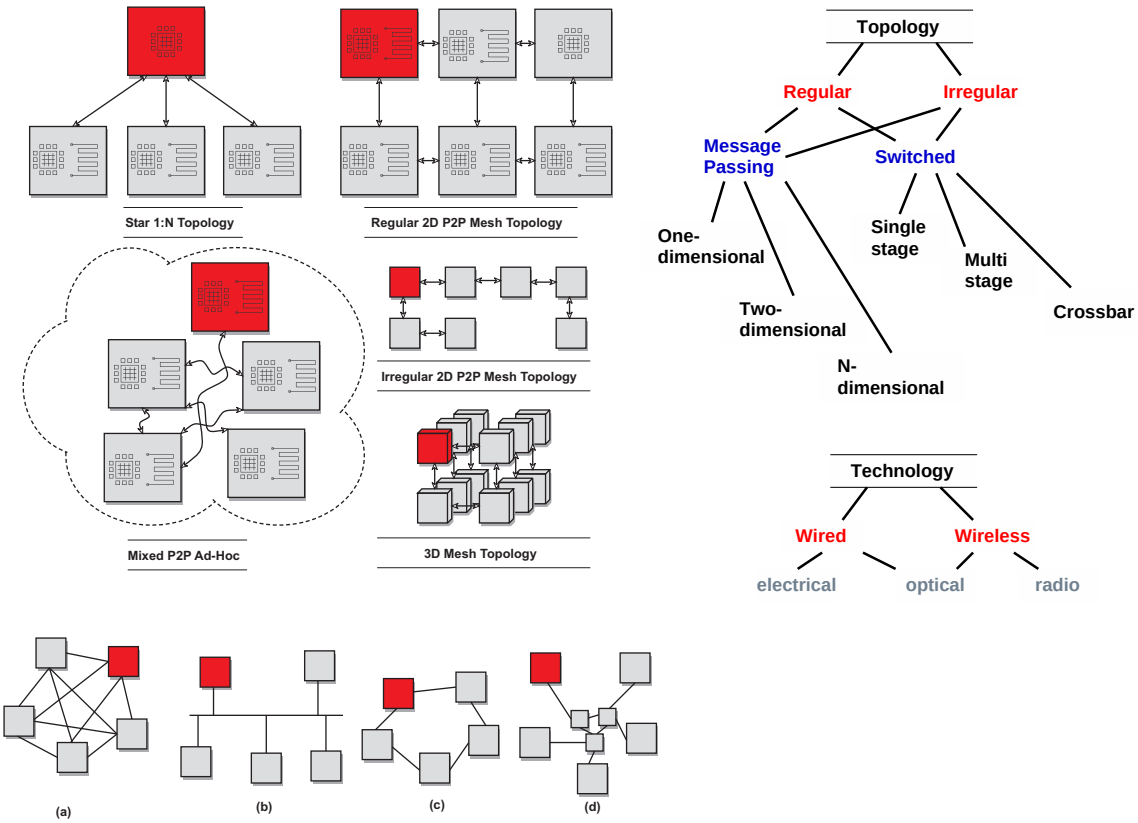



**Fig. 6.3** *(Left) Common Network Topologies (Right) Summary of network topology and technology classification*


ISBN 978-3-527-33606-7

Two-dimensional mesh-like networks, either fully (upper right corner) or partially (middle row, right side) connected and using point-to-point links, are the preferred topologies for wired planar material-integrated sensor networks having the lowest interconnect complexity but still providing robustness in the presence of missing or defective links due to path redundancy (there is more than one possible path from a source to a destination node). They have a connectivity degree of 4 and a maximal distance of $\log_2 N$. Furthermore the logical network topology corresponds to the geometrical placement order of nodes.

In fully connected networks (a) each node is connected with each other node, leading to a connectivity degree of N-1 and a distance of 1 with the highest degree of robustness and lowest message passing latency, but requiring the highest resource demands. Cube networks (three-dimensional topology, middle row, right side) provide a good compromise between the afore mentioned networks, having a connectivity degree of 6 and a distance of $\log_3 N$ (resulting in a lower message passing latency compared with two-dimensional networks). They are a special case of a generic hypercube networks (n-dimensional), with very limited practical use in material-integrated sensor networks due to the complex interconnect structure.

Network topologies (see right in Fig. **6.3**) can be basically classified in regular and irregular structures. Mobile Ad-hoc networks with peer-to-peer communication are prominent examples for networks with a massive irregular structure that changes over time. The connectivity in mobile networks is dynamic in the spatial and temporal dimension and is determined by the overlapping of (mostly) circular radio transmission ranges, different from wired connectivity.

In chain or ring networks (c) the connectivity is 2, the maximal distance N-1, providing no (chain) or low (ring) robustness. Here, failure of a single node can already be critical: The chain is only as strong as its weakest link.

Hierarchical networks (d) provide node partitioning in spatial bounded sub-networks connected with each other by using dedicated routers (for example, sub-star networks arranged in chain networks applied in **[GHE10]**).

### *6.2.2 Redundancy in Networks*

A chain network has no redundancy, but two- and three dimensional mesh and cube networks (with four and six neighbour node connections, respectively) provide an increasing number of possible path alternatives without introducing additional high connectivity complexity and costs (number of connections are in the order of nodes). The possible number of paths P connecting the (upper) left and (lower) right nodes and the connectivity costs C (number of all links in a regular configuration) of the network depend on the number of nodes ($n$, $m$, $o$) in each dimension respectively and can be calculated from (assuming a connecting path visits each node at most one time):

 ISBN 978-3-527-33606-7

$$
\begin{aligned}
&P_1(n) = 1,\ P_2(n,m) = \frac{(n+m)!}{n!m!},\ P_3(n,m,o) = \frac{(n+m+o)!}{n!m!o!} \\
&C_1(n) = n-1,\ C_2(n,m) = 2nm - m - n, \\
&C_3(n,m,o) = 3mno - m - n - o
\end{aligned}
\tag{6.1}
$$

Fig. **6.4** compares the redundancy of 1d, 2d, and 3d grid networks in relation of the number of nodes. Naturally, a linear chain has no redundancy (i.e., there is only one possible path between a source and destination node), but a ring has already a redundancy degree 2.

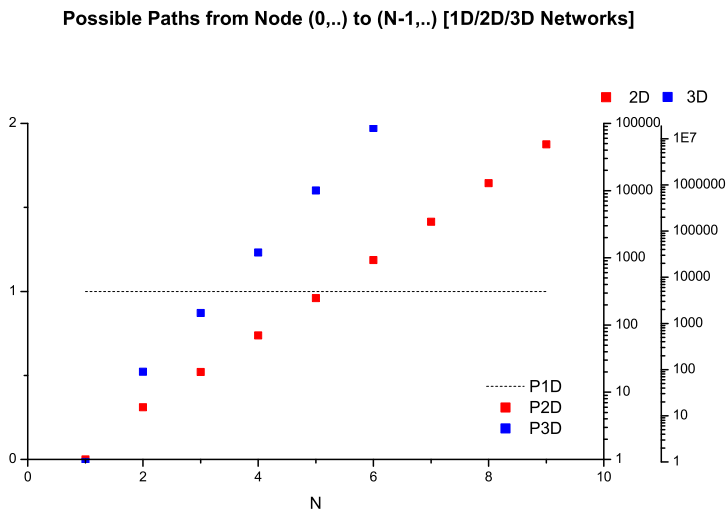



**Fig. 6.4** *Redundancy in mesh-like networks (chain, grid, cube): possible paths from node (0,..) to (n-1,..) depending on the network size (n nodes in each dimension).*

### *6.2.3 Protocols*

A communication or network protocol is basically a specification. A protocol defines:

- The message format and structure;
- A sequence flow between communication end-points or routers assigning states to communication steps, commonly defined by a finite state machine;
- The operational and synchronization semantic;


ISBN 978-3-527-33606-7

- Access control and Security;
- Structural organization of networks;
- Kind of node or communication end-point addressing (if any);

Communication start- and end-point can be computers, routers, programs (or more precisely spoken processes of a programs), other protocol layers, and so on, and depends on the particular communication protocol.

Therefore, there is commonly a set of protocols with different operational and organizational semantics invloved in end-to-end message communication, assigned to different layers of the Open Systems Interconnection (OSI) model, shown in Fig. **6.5**. The layers are:

1. Physical Layer (Technology-dependent raw data transfer)
2. Data Link Layer (Manages specific communication devices, message fragmentation)
3. Network Layer (Routing of messages, flow and congestion control)
4. Transport Layer (Manages end-to-end message delivery)
5. Session Layer (Control of logical/virtual links and sessions)
6. Presentation Layer (Device-independent data representation)
7. Application Layer (Interface to user processes)

Each protocol layer requires its own data and message encapsulation, adding each time a protocol specific header. For example, the application layer adds a process identifier, the transport layer adds computer addresses.

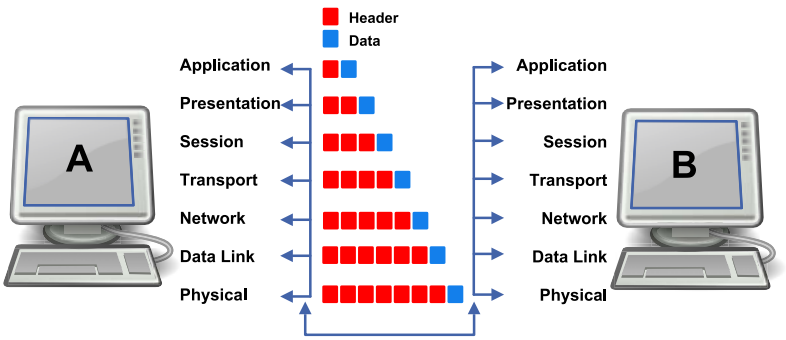



**Fig. 6.5** *OSI layer model*

There are still many sensor networks based on traditional computer networking approaches like TCP/IP and remote procedure call (RPC) communication systems

 ISBN 978-3-527-33606-7

**[BAG10]**, resulting in an increased overhead and requiring a large amount of computational, power and microchip resources, being contra indicated for energy-aware and microchip-level sensor network designs. But a case study showed that application-specific FPGA implementations using the RTL architecture of a reduced UDP/IP protocol stack requires about 200k logic gates, which is reasonable low and suitable for microchip scale **[LOE05]**.

Still most communication protocols require unique addressing of nodes (or service endpoints), commonly human-managed, able to compose an organizational structure of the network. With an increasing number of nodes beyond the trillion boundary this addressing scheme is questionable up to impossible. Why should two sensors communicate with each other?

Different path finding schemes propagating messages and data rely on data-based and event-based routing, i.e., data sources and data sinks publish or negotiate the interest of specific data and events, e.g., sensor data of a specific kind or originating from a specific bounded region of interest. The data-centric approach avoids the node addressing requirement, and, e.g., replacing unique node addresses by spatial coordinates or spatial neighbourhood connectivity, like in wireless networks. Bio-inspired networking focus on self-organizing and self-adaptive structures without any external organizer, e.g., implementing some kind of swarm intelligence.

### *6.2.4 Switched Networks versa Message Passing*

Switched networks are peer-to-peer networks connecting nodes directly. A switched network uses line multiplexer to connect a node with multiple other nodes. A full crossbar switch for a network of $N$ nodes require $N^2$ multiplexer resources, and it is conflict free (each node can be connected with any other node at any time). A multi-stage switch network using, e.g., binary switches, has lower resources requirements (usually $N\log(N)$ switches), but is not conflict free. A network switch commonly not provide synchronization between nodes, in contrast to message passing networks.

 ISBN 978-3-527-33606-7

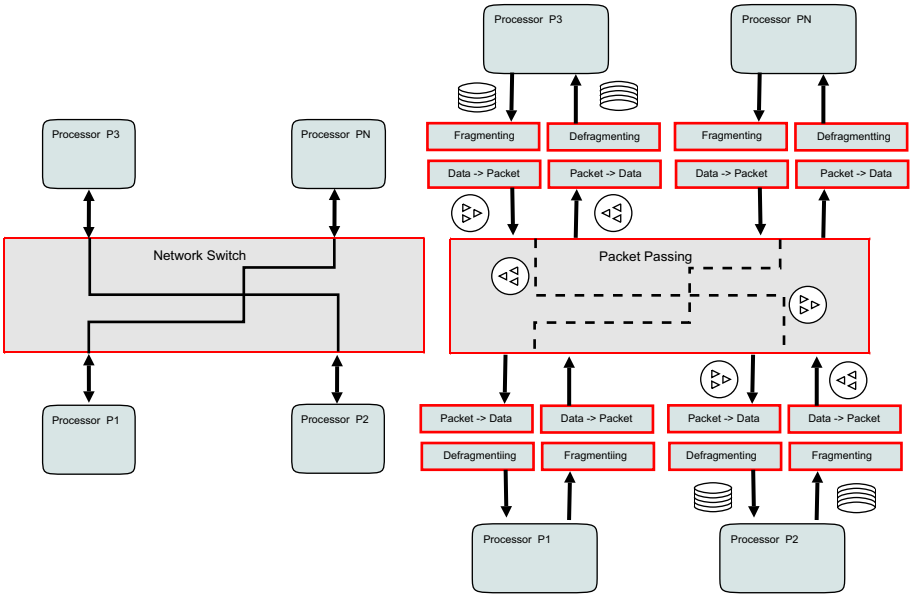



**Fig. 6.6** *(Left) Switched Circuit Network (Right) Message Passing Network*

### *6.2.5 Bus Systems*

Bus systems are very popular in network designs, especially their deployment in material-integrated systems. A bus system has linear hardware costs of the order Θ(N), if N is the number of network nodes connected by the bus. But they have the disadvantage of a total bandwidth of 1, i.e., only one connection between two nodes is possible at any time. Furthermore, one node failure can block the entire bus by a defective receiver, sender, or inter-connect wire.

A bus is a switched network without node and communication process synchronization. The access of the bus is hence exclsuive and requires a mutual-exclusive scheduling performed by a shared arbiter, shown in Fig **6.7**. A node wanting to access the bus, i.e., the data lines, will request the access by raising a request signal. If the access is granted by the arbiter an acknowledge is send to the requesting node. Higher level of node synchronization can be introduced by simple messages sent by the nodes on dedicated control lines or by using the data lines with a special coding. A minimal set of messages consists of (adapted from the IEEE 488 / GPIB protocol):

**LISTEN <dstaddr>**

The current owner of the bus sends a control command to notify the receiver of the following data messages.

**UNLISTEN**

 ISBN 978-3-527-33606-7

All nodes wait for control commands.

**TALK <srcaddr>**

The current owner of the bus publishes his address.

**UNTALK**

This control command indicates that no node should be a sender of data messages.

Each node, the inter-connect wires, and the arbiter can be considered as single point of failures.

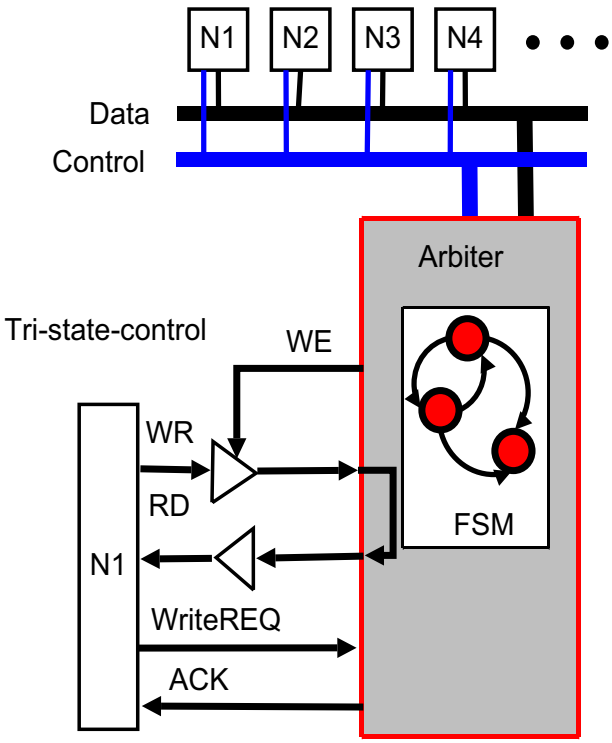



**Fig. 6.7** *Bus Request-Acknowledge Protocol handled by an Arbiter*

### *6.2.6 Message Passing and Message Formats*

A network message is used to transport data between network nodes and processes executed on the nodes. Usually processes are communication endpoints, rather than

 ISBN 978-3-527-33606-7

computers. Message passing can involve multiple protocols belonging to different communication layres (physical, logical link, or transport layers). Each protocol adds a header to the message, shown in Fig. **6.8**. The header contains protocol specific information and is used for the message routing in the network, discussed in the next section.

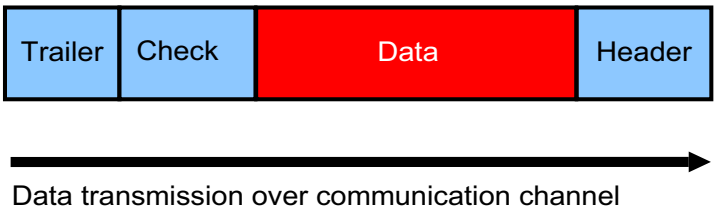



**Fig. 6.8** *A typical message format consisting of header with information required for routing, the data pay-load, and optional check fields and a trailer marking the end of a message.*

Message passing requires a significant amount of time, depending on the network topology (number of hops between a source and destination node), the communication hardware (bandwidth and latency), and the amount of data (N bytes):

$$T(N)_{S\to D} = T_{\text{Overhead}} + T_{\text{Routing Delay}} + T_{\text{Transfer}}(N) + T_{\text{Conflict}} \tag{6.2}$$

The message passing time adds an overhead to the data processing latency in distributed and parallel systems. Communication channels are usually shared by different "virtual" paths (connecting a set of source with a set of destination nodes). Hence, the conflict time $T_{\text{Conflict}}$ can be a significant overhead if there is a high channel utilization.

### *6.2.7 Routing*

A network node is connected to a network via communication ports. At elast one communication port is required. A network node can be a service-end point or a message passing unit (or both), the router. A routing service determines the path of a message from an incoming to an outgoing port. Usually routing bases on the source-destination parameters of a message.

In macro-scale sensor networks nodes are often connected by using wireless technologies with dynamic ad-hoc network topologies. Protocol-based routing in changing networks (regarding communication connections, node position, and node failures) is one main challenge in the design of robust and energy-aware sensor networks. Most traditional routing protocols rely on an address space uniquely identifying nodes as well as routing

 ISBN 978-3-527-33606-7

tables storing network information usually limited to a bounded spatial scope. Path finding is required for the process of forwarding of a message, and is the main goal of routing by minimizing the number of passed routers (e.g., nodes). Commonly there is a set of different paths connecting nodes (see the previous Section). In the past many routing algorithms were developed (for wireless and wired networks) finding 1. alternative paths in the case of missing or non-operating connectivity in parts of the networks, and 2. by finding the shortest path (concerning the distance in hop counts and the delivery latency). One examples is junction based routing (discussed in detail in **[BAD11]**). Routing in large-scale wireless sensor networks is discussed in detail in **[LI11]**.

Geographic routing and addressing bases on geometrical relations between nodes, for example, relative addressing specifying the relative distance between nodes, can be used to avoid absolute unique node addressing, which is not applicable in large scale miniaturized networks with an initially unknown configuration, like smart dust networks **[WAR01]**.

The scaling of networking down to material-integrated level requires simplified and robust approaches. Some of them capable for microchip level implementations satisfying low-resource constraints are summarized below. They all not rely on routing tables (saving memory and computation) as well as they are able to adapt to network changes immediately.

*Store-and-Forward Routing*

The Store-and-Forward (*SF*) algorithm divides message passing in three steps, shown on the left side of Fig. **6.9**:

1. Receive and store the entire message from an input port $CP_{\mathrm{I}}$
2. Compute the output port $CP_{\mathrm{O}}$ by commonly using look-up tables giving a destination-port mapping.
3. If the output port $CP_{\mathrm{O}}$ is not busy, send the entire message to the output port, else wait for the the output becoming ready and send message.

The *SF* algorithm requires $N*P$ full size message buffers, with $N$ being the maximal number of buffered messages per port and $P$ the number of communication ports per node.

The routing timea message (or packet) of size $N$ bytes is given by:

$$T_{\text{Routing SF}}(N,H) = H\left(\frac{N_{\text{Packet}}}{B} + \Delta\right) \tag{6.3}$$

and is the product of the message transfer time ($N$/B) with a given communication bandwidth $B$ and the number of hop nodes ($H$). An additional routing overhead (step 2) $\Delta$ applies on each hop, too.

 ISBN 978-3-527-33606-7

*Cut-through and Wormhole Routing*

Cut-through (*Ct*) and Wormhole (*Wh*) routing splits messages in parts (so called flits). A network message consists of a (small) header and a (large) data part. If the header fits in one flit packet, only the header flit must be received and stored, which can be immediately evaluated. Hence, *Ct*/*Wh* routing is divided in different steps, shown on the right side of Fig. **6.9**:

1. Receive and store the header of a message
2. Compute of the output port $CP_{\mathrm{O}}$ by commonly using look-up tables giving a destination-port mapping.
3. If the output port $CP_{\mathrm{O}}$ is not busy, send the header and all following flit packets to the output port.
4. *Ct*: If the output port is busy, store the entire message and send it if the output port is ready.
   *Wh*: If the output port is busy, notify the sending network node to stop sending further flit packages. The notification is propagated up to the source node.

*Ct* and *Wh* algorithms behave only differently if the output port is blocked (i.e., busy). The *Ct* algorithm requires *P* full size messages buffers, whereby the *Wh* algorithm requires only *P* flit size buffers.

The routing time for a message (or packet) of size *N* bytes is given by:

$$T_{\text{Routing CT}}(N,H) = \frac{N_{\text{Packet}}}{B} + H\Delta \ll T_{\text{Routing SF}}(N,H) \tag{6.4}$$

and is a sum of the transfer time for the entire message (or packet) *N*/*B* with the communication bandwidth *B*, and *H* times of the routing overhead Δ occurring on each hop node.

 ISBN 978-3-527-33606-7

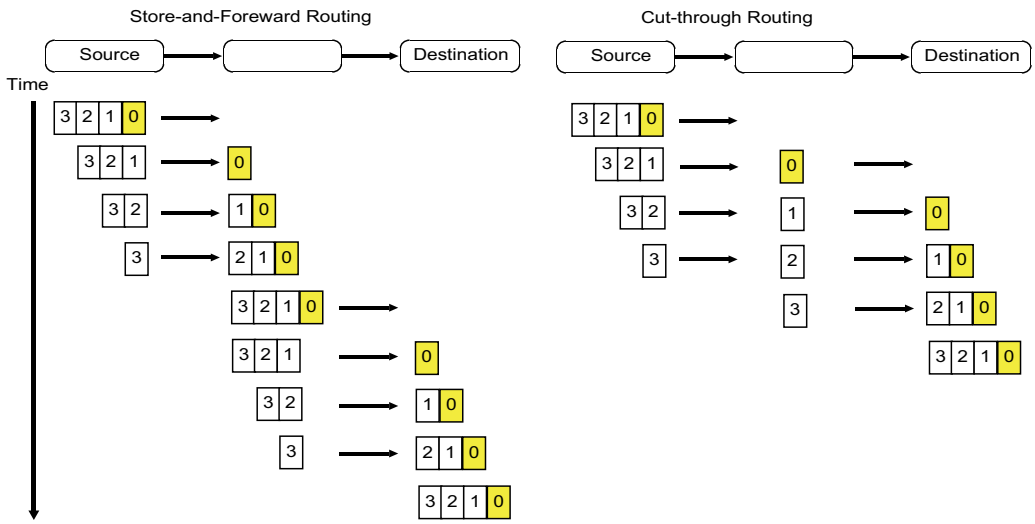



**Fig. 6.9** *Comparison of Store-and-Forward and Cut-through Message Routing*

*Delta-Distance Routing*

In this scheme, a path from a source to a destination node in a grid-like network (of arbitrary dimension) is given by its relative distance in each cartesian direction, the delta vector $\Delta=(\delta_1, \delta_2, .. , \delta_n)$. Though $\Delta$-distance addressing and routing is one of the simplest approaches it is very well suited for microchip level implementation.

Commonly a message is forwarded to the destination node by choosing a path with the goal to minimize one distance value after each other by passing the message along nodes in the path (hopping), each time decrementing the respective distance value $\Delta_i$ in the i-th direction, until the $\Delta$ vector is finally zero ($\Delta^0$ is the original distance vector and *M* is the message content). This is shown in Alg. **6.1**, considering a message as a mobile process. The route function tries to find a suitable direction to forward the message using the `moveto(DIR)` operation. After the message was moved to the new neighbour node, the routing function is applied again until the destination is reached.

**Alg. 6.1** *Standard and Simple Adpative Δ-Distance Routing (in Pseudo Code Notation)*

```
TYPE DIM = {1,2,..,m} Enumeration of network topology dimensions
TYPE DIR = {NORTH,SOUTH,EAST,WEST,UP,DOWN,..} Symbolic directions
DEF dir = fun (i) → match i with  Numerical mapping of directions
  ... -3 → UP | -2→NORTH | -1→WEST | 1→EAST | 2→SOUTH  3 → DOWN ...

DEF route_xy = fun (Δ,Δ⁰,M) →
```

 ISBN 978-3-527-33606-7

```
  if Δ ≠ 0 then
    for first i ∈ DIM with Δi ≠ 0 do:
      if Δi > 0 then
        moveto(dir(i));
        routexy(Δ with Δi:Δi-1,Δ0,M);
        return
      else if Δi < 0 then
        moveto(dir(-i));
        routexy(Δ with Δi:Δi+1,Δ0,M);
        return
    discard(M)
  else
    deliver(M)

DEF routexyvar = fun (Δ,Δ0,M) →
  if Δ ≠ 0 then
    for each i ∈ DIM with Δi ≠ 0 try:
      if Δi > 0 ∧ link?(dir(i)) then
        moveto(dir(i));
        routexyvar(Δ with Δi:Δi-1,Δ0,M);
        return
      else if Δi < 0 ∧ link?(dir(-i)) then
        moveto(dir(-i));
        routexyvar(Δ with Δi:Δi+1,Δ0,M);
        return
    discard(M)
  else
    deliver(M)
```

This trivial routing protocol requires no routing table and is suitable for low-resource hardware SoC implementations. There is only one possible path which can be traveled if Alg. **6.1** is applied strictly ordered ($\texttt{route}_{xy}$), limiting this simple Δ-routing protocol to complete and regular network topologies without missing or broken links. If the routing direction can be chosen based on actual detected node connectivity (using the `link?` operation and $\texttt{route}_{xyvar}$), there is a set of disjoint paths which can be chosen alternatively.

*Adaptive Delta-Distance Routing with Backtracking*

Robustness and support of incomplete mesh-like networks with missing and non-operating links and nodes can be achieved by extending the simple Δ-routing protocol with an selection mechanism of the routing scheme providing the ability to overcome missing network connectivity in a specific direction **[BOS12A]**. This approach bases on a sink attraction and backtracking behaviour. There are different routing behaviours selected by the available local link connectivity (tested by the function `link(DIR)`) and information stored in the message (Δ,Γ), shown in Alg. **6.2**, again considering a message as a mobile

 ISBN 978-3-527-33606-7

process. An optional preferred routing direction ω can be specified. The Γ vector marks back or opposite traveling.

**Alg. 6.2** *Advanced Adaptive Δ-Routing with Backtracking (in Pseudo Code Notation)*

```
DEF route_s = fun (Δ,Δ⁰,Γ,ω,M) →
 d^in := incoming-direction-of-message(M)
 if Δ=0 then deliver(M,Δ⁰)
  else if not route_normal(Δ,Δ⁰,Γ,ω,M) then
    if not route_opposite(Δ,Δ⁰,Γ,ω,M,d^in) then
      if not route_backward(Δ,Δ⁰,Γ,ω,M,d^in) then
        discard(M)
WITH
 route_normal = fun (Δ,Δ⁰,Γ,ω,M) →
  for each i ∈ DIM starting with ω and with Δ_i ≠ 0 try:
    rev := Γ ≠ 0 ∧ Δ_i ≠ Δ_i⁰
    if Δ_i > 0 ∧ link(dir(i)) & not rev then
      moveto(dir(i));
      route_s(Δ with Δ_i:Δ_i-1,Δ⁰,Γ,ω,M);
      return true
    else if Δ_i < 0 ∧ link(dir(-i)) ∧ Γ_i=0 then
      moveto(dir(-i));
      route_s(Δ with Δ_i:Δ_i+1,Δ⁰,Γ,ω,M);
      return true
  Reached here: No success! All failed!
  return false

 route_opposite = fun (Δ,Δ⁰,Γ,ω,M,d^in) →
  for each i ∈ DIM starting with ω try:
    if Γ_i = 0 ∧ d^in ≠ dir(-i) ∧ link(dir(-i)) then
      moveto(dir(-i));
      route_s(Δ with Δ_i:Δ_i+1,Δ⁰,Γ with Γ_i :-1,ω,M);
      return true
    else if Γ_i = 0 ∧ d ≠ dir(i) ∧ link(dir(i)) then
      moveto(dir(i));
      route_s(Δ with Δ_i:Δ_i-1,Δ⁰,Γ with Γ_i :-1,ω,M);
      return true
  Reached here: No success! All failed!
  return false

 route_backward = fun (Δ,Δ⁰,Γ,ω,M,d^in) →
  for each i ∈ DIM starting with ω try:
    if Γ_i = 0 ∧ d^in < 0 ∧ link(dir(-i)) then
      moveto(dir(-i));
      route_s(Δ with Δ_i:Δ_i+1,Δ⁰,Γ with Γ_i :-1,ω,M);
      return true
    else if Γ_i = 0 ∧ d^in > 0 ∧ link(dir(i)) then
      moveto(dir(i));
      route_s(Δ with Δ_i:Δ_i-1,Δ⁰,Γ with Γ_i :-1,ω,M);
      return true
```

 ISBN 978-3-527-33606-7

```
    else if Γi = -1 ∧ link(dir(-i)) then
      moveto(dir(-i));
      route_s(Δ with Δi:Δi+1,Δ0,Γ,ω,M);
      return true
    else if Γi = 1 ∧ link(dir(i)) then
      moveto(dir(i));
      route_s(Δ with Δi:Δi-1,Δ0,Γ,ω,M);
      return true
  Reached here: No success! All failed!
  return false
```

For a first communication attempt, the normal routing strategy is tried with the goal to minimize the Δ-vector in each dimension in any order. If this is not possible due to a lack of required connectivity the message is tried to send in one opposite direction increasing the Δ-vector temporarily. An opposite travel is marked in the Γ entry of the message. Finally, if this strategy fails also, the message is send backward, finally reaching the source node again if there are no other routing alternatives found on the back-path. Some care must be taken to avoid sending a message back to the direction from which it originally comes from (resulting in a ping-pong forwarding with a live lock). To summarize, this simple routing algorithm uses decision making based on information stored in the message, and neighbourhood connectivity retrieved by the current node. It is suitable for peer-to-peer networks with a limited number of neighbours, as found in wired material-integrated sensor networks and benefits from low computational requirements and complexity (a minimized hardware implementation requires less than 100k logic gates resources **[BOS11A]**).

The smart backtracking-based routing algorithm is well suited for path migration of mobile agents in mesh-like networks (of any dimension), requiring no host routing tables and no path planning in advance.

 ISBN 978-3-527-33606-7

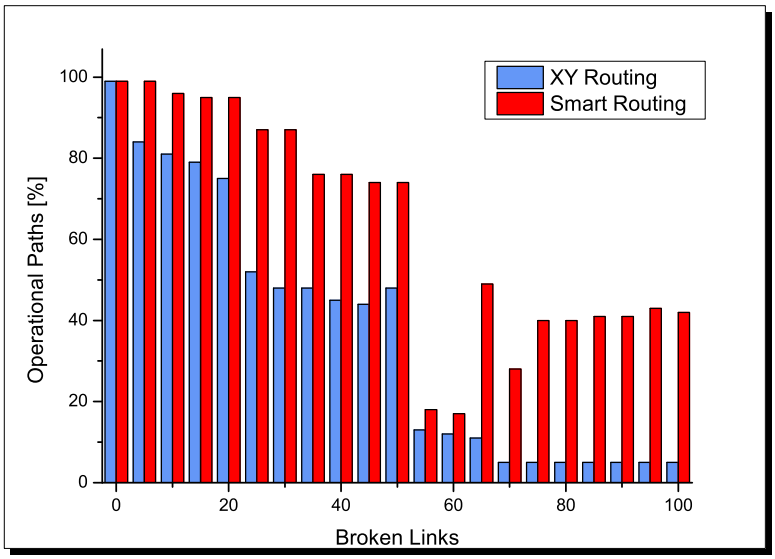



**Fig. 6.10** *Simulation results from an reachability analysis in a two-dimensional mesh sensor network arranged in a 10 by 10 matrix with up to 180 links. Each node was connected with up to four neighbour nodes. The percentage of the operational paths in dependency of the number of broken node-to-node links was compared for the traditional non-adaptive XY and the adaptive smart routing strategy. Beyond 50 broken links the smart routing outperforms, and up to 40 percent of paths are still reachable.* ***[BOS12A]***

*Data-centric and Event based Routing*

Commonly stream-based data communication is used in (distributed) signal processing systems, transferring data from sensors (information sources) to computational nodes (information sinks) continuously, mostly based on a service request-reply architecture with sensors as a service (SaaS). Directed diffusion **[INT00][ZHA07]** is a data-centric routing and path finding approach suitable for sensor networks, which promises energy efficient delivery of messages based on events and the interpretation of their content and the demand of the data, in contrast to peer-to-peer approaches only considering geographical ontologies of the network. There are two different event-based sensor data distributions strategies which can be applied: 1. the sensor node (the data producer) detects an event and decides to notify the neighbourhood (the data consumer) or the entire network of the availability of new data by using message flooding (i.e., Rumor routing in **[BRA02]**, a compromise between flooding queries and flooding event notifications), and 2: the data consumer (recipient) notifies all neighbourhood nodes in a bounded region for the interest

 ISBN 978-3-527-33606-7

of specific data, and the sensor node detecting an event can use this information for delivering the sensor data to a specific node on a shortest path (i.e., Directed diffusion routing).

*Agent based Routing*

The simplest forwarding strategy in a network is flooding the network by using multi- or broadcasting strategies performed on the link layer. An incoming message is replicated and sent to all neighbour nodes. Wireless networks have the advantage that a message can be received by many nodes around the neighbourhood of the sending node efficiently implementing 1:N multi- and broad-casting immediately on the link layer, but still requiring computational activity by all N nodes, decreasing the energy efficiency significantly. In wired networks 1:N multi-casting requires the composition of N uni-cast messages, resulting in the worst routing algorithm in terms of energy efficiency.

Instead of replicating and forwarding a message to all neighbours of a node, the message can be forwarded to a randomly chosen node in the neighbourhood, resulting in a random walk. Additionally, gossiping algorithms operate within a limited spatial region, only reaching a small number of nodes, discussed in **[HAA06]**. The basic ideas of random walks are derived from biologically inspired algorithms, for example, based on ant or bee swarm behaviours (discussed in detail in **[FAR09]**).

Usually messages can be treated as passive units. The routing strategy depends entirely on the algorithm and knowledge provided by each forwarding node along the path. Instead treating messages as active units - like mobile agents can do - leads to a decoupling from the capabilities and knowledge of each node and provides increased autonomy. Routing decisions are now made by the agent itself carrying the message. The goal of the agent or a collection of agents is to deliver the message carried by the agent to a destination node. Replication can be used to increase the delivery probability and decrease the latency. Learning agents can improve path finding by considering their travel history. Data-centric directed diffusion algorithms can be easily implemented with autonomous mobile agents.

The aforementioned backtracking routing protocol was developed with an agent-based system and implemented entirely on microchip level, using neighbourhood hopping and sink attraction behaviour, described in detail in **[BOS13A]**.

In **[BRA02]** an event notification algorithm is proposed using notification agents installing routing information of an event, for example, a sensor value is exceeding a threshold, in all the nodes they are visiting along a path (rumor routing). This approach does not flood the network. Search agents are used to query and find an event installed by the notification agents. The root location of an event can be found by following the path the notification agents have marked, offering backtracking routing.

Directed diffusion can also be implemented with different agents. Agent-based routing is used in several works to improve communication robustness, reliability, latency, and energy efficiency, but not generally using autonomous agents for message delivery. Such

 ISBN 978-3-527-33606-7

examples for agent-based design methodologies can be found in **[YAN02][SHA07] [EBR11]**.

Biologically inspired routing in conjunction with agents and path marking is another approach to find optimal paths in ad-hoc and wireless networks, for example, ant colony algorithms described in **[ZAH12B]**, which is used to find out optimal and efficient migration paths for mobile agents.

### *6.2.8 Failures, Robustness, and Reliability*

Communication failures can originate from failures of the technological interconnect, the link receiver or sender unit, and finally from process failures performing communication on protocol level. The run-time behaviour of communication (with failures) and communication networks can be distinguished by different properties and behaviours, all having influence of the overall system stability of the sensor network (based on **[GUE06][WU99]**):

- *Fair-loss and Finite duplication*: a message from a sender process $p_a$ directed to a destination process $p_b$ is eventually delivered with a probability P<1 if none of the both processes crashes and the link is capable of retransmission (compensating lost messages). Furthermore, if a message was sent a finite number of times, it can only be delivered a finite number of times. This link behaviour can be assumed to be always present in any technical system.
- *No creation/Causality*: If a message was delivered at least one time to a process $p_b$ then there was a process $p_a$ sending the message to $p_j$ at least one time. If this property is violated, either by link or protocol failures, ambiguities and data corruption can occur.
- *Stubborn delivery*: A process $p_a$ sends a message to a destination process $p_b$ at most a finite number of times. It is possible that the message is delivered an infinite number of times, which can be system-critical in energy autonomous networks. A retransmit-forever behaviour of the sending process or node can be one cause of this event.
- *No delivery/Live-lock*: A message was sent by a process $p_a$ and requires routing through a finite number of network nodes to reach the destination process $p_b$. There is at least one path connecting the nodes of both processes. It is possible that a message never reaches the destination node, but circulates in the network and is routed an infinite number of times. This situation mainly has its origin in (simplified and incomplete) protocol design, rarely caused by technical failures. For example, the simple but smart routing based communication protocol SLIP **[BOS12A]** used in microchip-level sensor networks suffered from livelocks occurring under rare certain network topology situations caused by (over) simplification of the protocol implementation. Live-locks are system-critical.

 ISBN 978-3-527-33606-7

- *Disorder*: The order of messages is not preserved and is critical for proper communication.
- *Data corruption*: a message gets corrupted during its transmission or during routing. Non detected data corruption can result in process or total system failures and is system-critical.

*Perfect links* guarantee a reliable delivery of a message, this includes that if a message is send from a process $p_a$ to a process $p_b$ it is eventually delivered, there are no duplicated messages (each message is delivered only once), and if a message was delivered it was send by a process. Reliability can be achieved on different networking layers: link layer, protocol layer, process layer. Though there are technological communication links with low data loss and high noise immunity, the interconnect structure can be broken or defective, resulting in a broken connection between two nodes. This can be always assumed in material-integrated sensor networks. Traditional reliability on protocol layer leads to an increase of computational complexity, communication complexity, and resource demands, which is contra indicated for microchip level and energy-optimized implementations.

Reliability in distributed systems can be achieved by **[WU99]**:

1. Programming fault tolerance
2. Communication fault tolerance
3. Self organization and replication of autonomous and mobile program (process) units, which is discussed in Section **6.4**

### *6.2.9 Distributed Sensor Networks*

A distributed sensor network as the central part of a sensing system consists of multiple active sensor nodes, a set $N=\{n_1,n_2,...\}$, connected and arranged in a network forming a graph G(N,C) with edges C, a set $C=\{c_1 ,c_2,..\}$, connecting nodes and providing communication (and eventually energy transfer) between nodes. Each sensor node provides signal and data processing for a set of sensors $S=\{s_1,s_2,..\}$, communication, some kind of energy supply and management, and sensor interfaces. The connectivity of the sensor nodes can be used for data and power exchange, too. Each sensor node should provide a minimal degree of autonomy and independence from other nodes in the network.

From the computer science point of view a sensor network can be composed of a set of processes P={p1,p2,...} performing parallel data processing and interaction using messages M={m}, shown in Fig. **6.11**. The processes communicate by exchanging data by using messages, and messages are exchanges by the nodes hosting the processes by using communication links, discussed in Sec. **6.1**. Message-based communication, which takes place between processes, requires protocols for source and destination process synchronization and message routing (forwarding). Thus in this sense and assuming the assumption of the above definition a distributed sensor network can be considered as a distributed

 ISBN 978-3-527-33606-7

computer performing distributed data processing. Interaction between nodes is required to manage and distribute data and to compute information.

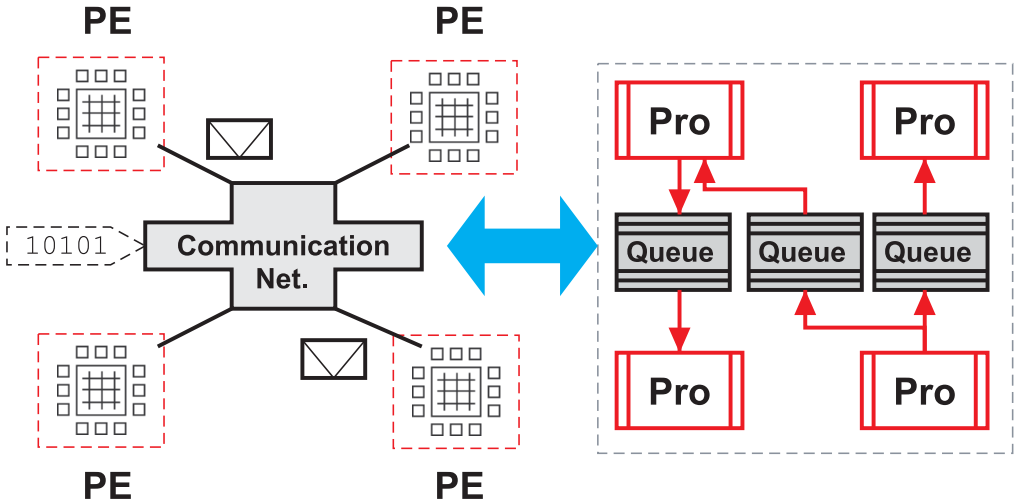



**Fig. 6.11** *Abstraction of distributed computing in sensor networks with processes and message passing using queues*

The properties and operational behaviour of a distributed system can be summarized as follows:

1. Processing elements (PE) represent the physical computational resources and are part of the nodes of the sensor network
2. Processes (P) are logical resources which perform the data processing for a specific task
3. Cooperative communication and inter-process communication handled by message passing (instead of a centralized master-slave model)
4. Communication links (the inter-connection network) are physical resources and connects PEs
5. Processes are cooperative by interaction
6. Communication delay does not affect the overall distributed program behaviour (immutability)
7. Robustness in the presence of resource failures (failures of single logical or physical resources do not affect overall system behaviour)
8. Run-time reconfiguration: adaptation of single resources or the whole system in the presence of resource failures

Of these physical and logical elements, specifically 1, 2 and 4 are prone to fail.


ISBN 978-3-527-33606-7

In **[AKY02]** a general discussion of the architectures, communication, and data processing issues in sensor networks can be found.

Sensor network applications introduces application-specific network domains with specific requirements and constraints. These are primarily (see **[GRA14]** for details):

**Personal Area Networks (PAN)**

Personal area networks connect primarily mobile devices or laptops with sensor networks, for example, body area networks (BAN) and wireless body area networks (WBAN) that connect smart sensors with data acquisition devices like mobile phones.

A (W)BAN can incorporate different classes of sensor types, for example, heart beat and EEG monitors, temperature, moisture, pressure, and many more.

The fields of application for BANs are medical diagnostics and smart clothing, but not limited to these main fields.

Sensor networks as a basic part of PANs and the below discussed APNs have different communication requirements. They are equipped with nodes having limited computational resources and storag capacity, limiting the communication capabilities significantly. Energy consumption can be a major challenge in the design of sensor networks.

**Ambient and Pervasive Networks (APN)**

Ambient and pervasive networks (APN) are strong heterogeneous networks consisting of networks nodes ranging from sensors over mobile devices to home computer. They equip buildings inside and ambient environments outside with sensors and computation with the goal to adapt these environments personally and to meet personal needs. Data inference, data mining, and data modeling are the major challenges in pervasive networks. Ontologies as a tool for a common understanding between different platforms, messages, users, and data are required to provide interoperability and a high degree of adaptivity to environmental changes. Personal area networks can be considered as part of these APNs.

**Wide Area Networks (WAN)**

Wide area networks connect different networks on a long spatial distance scale, basically providing the conenctivity for the Internet and all its public and private sub-networks. In contrast to APNs and PANs, they are equipped with network nodes and routers having enough computational power and storage capacity to support high traffic and high bandwidths. The communication over WANs is mostly client-server driven, in contrast to APNs and PANs with data and event driven communication. WANs rely on a unique node and process addressing scheme (IP-4/IP-6) and communication devices with a unique identifier (e.g., MAC address).

WANs are hierarchical organized and there exist usually a lot of paths from a source to a destination node (process) crossing different sub-networks, offering increased reliability based on redundancy and adaptive path finding.

 ISBN 978-3-527-33606-7

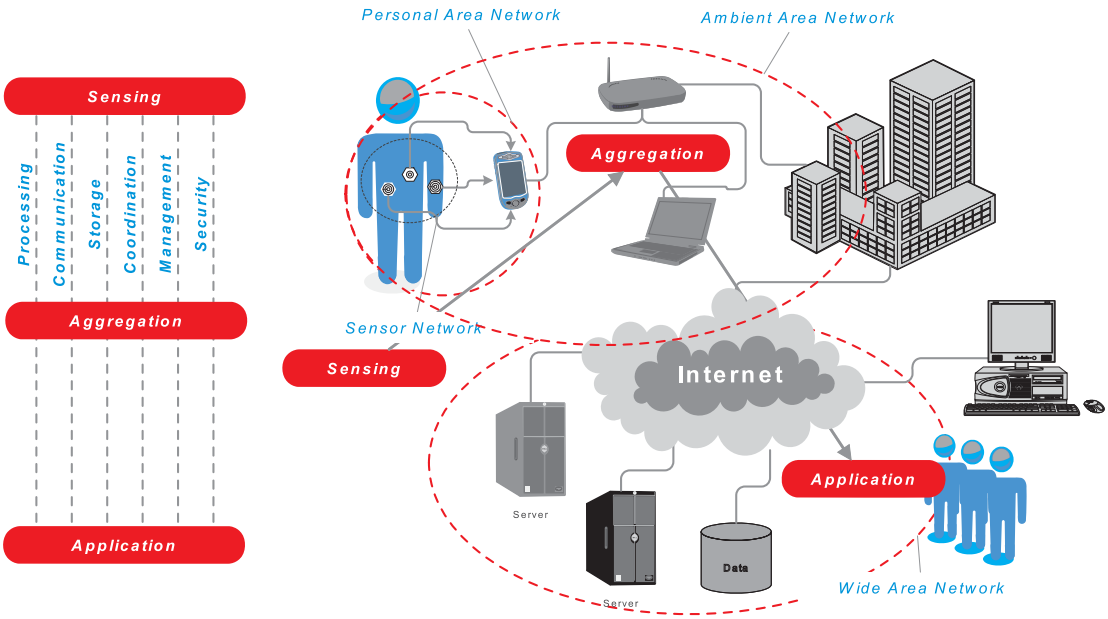



**Fig. 6.12** *A heterogeneous network environment with PAN, APN, and WAN domains (on network level) and sensing, aggregation, and application domains on the use-case level.*

There are basically three different layers of data processing in sensing applications, which require processing platforms with different computational power and storage capacity:

**Sensing**

Localized Acquisition and Pre-processing of Sensor Data, Sensor Data Fusion

**Aggregation**

Distribution and Collection of Sensor Data, Distributed Sensor Fusion, and globalized Sensor Information Mapping (using Map&Reduce methods from Big Data Mining).

**Application**

Presentation of condensed Sensor Information, Storage, Visualization, Interaction.

Theses different layers can be scattered in different network domains. The sensing layer is usually located in sensor networks, for example, body area networks, the aggregation layer can be found in personal and ambient area networks, and finally the applicaiton layer can be found in ambient area and wide area networks.


ISBN 978-3-527-33606-7

The characterization of sensor network features and their operational capabilities can be further divided into the following classes and terms, handled by all three layers of the sensing application, shown in Fig. **6.12**:

- Processing
- Communication
- Messaging
- Storage
- Ontologies and Data Models
- Manageability
- Security

### *6.2.10 Active Messaging and Agents*

There are several works using agents to implement active and robust messaging like the directed diffusion behaviour, discussed in **[MAL07]**. The Multi-Agent model, a collection of individual agents acting locally but following a global goal to be satisfied, is discussed in the next Section. Commonly communication tasks use passive messages that are forwarded from a source to destination node, or more precisely spoken processes as communication endpoints, by active processing units (routers, computers). In the agent model world, a message is active an the routing and transport of the message content is performed by active agents.

 ISBN 978-3-527-33606-7

## 6.3 Distributed and Cloud Computing: The Big Machine [Bosse]

Cloud computing is a new computation paradigm used to process a large amount of data (Big Data), commonly in a distributed environment. A Cloud offers the abstraction of the underlying data storage and computing platforms commonly with a central control instance. Cloud computing pursue multiple goals:

- Virtualisation of resources in heterogeneous environments, e.g., storage and processors
- Offering a service-orientated data processing approach, rather than a data-centric
- Computing of meaningful information from raw data, known as data mining
- Scalability of data processing
- Load balancing of processors in a distributed computing environment
- Storage balancing of data

A cloud in terms of storage 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
- A dynamic reconfiguration capability of the virtualized resources (computing, storage, connectivity and networks)

The taxonomy of data in large-scale sensor network applications can be splitted in three categories (based on the WEB data taxonomy by **[MOR12]**), each having their own complexity and characteristics:

1. Bags: Unordered collections of data with items consisting of scalar, textual, vectors, and multi-set values.
2. Graphs: Graph ordered collections of data consisting of nodes associated with data and vertexes defining the relation, that means the structure, of data. Graphs can be used to compose similarity or query (search) graphs.
3. Streams: Unbounded sequences of data items ordered by time. In contrast to low-level signal data streams the streams in sensor and WEB clouds are string heterogeneous.

 ISBN 978-3-527-33606-7

Originally, Cloud-based Computing was primarily used in business processes and consumer-producer scenarios offering distributed databases (storage) and services (computation). In the meantime, a Cloud can be any decentralized computing architecture, recently extended by Sensor Clouds integrating and connecting smart sensors in distributed services, data processing, and data storage. The Internet-of-Things (IoT) will heavily rely on Cloud-based services and infrastructures, and hence material-integrated sensing systems (MISS). The IoT can be seen as the bridge technology between the high-density and low-resource MISS and Computing and Storage Clouds, illustrated in Fig. **6.13**.

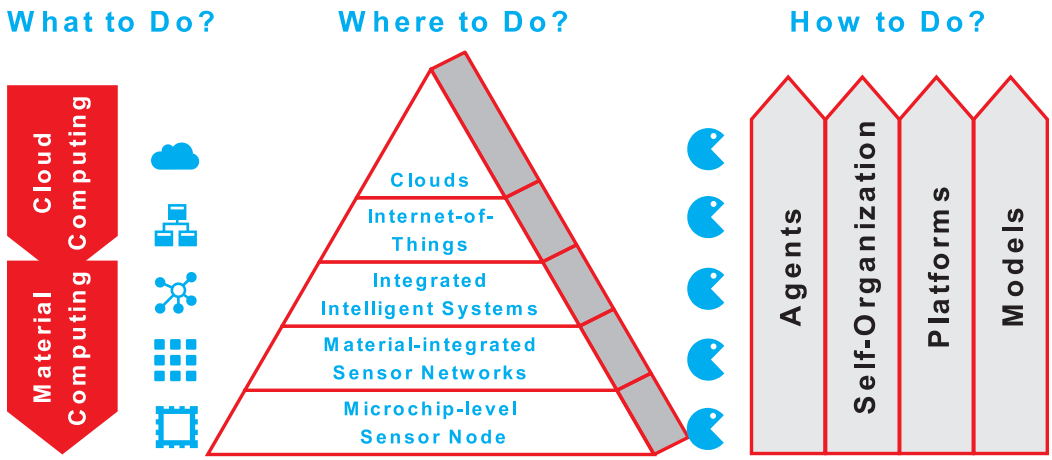



**Fig. 6.13** *From Material-integrated Sensing Systems (MISS) to Clouds*

Programming consists of computation and communication. This is still the case in cloud-based environments. Mobile Agent-based computation and communication can be used to satisfy requirements in such a distributed and loosely coupled machine. Furthermore, the deployment of agents can enable a paradigm shift from central control instances towards self-organizing systems not requiring a central control instance. Mobile agents can be considered as an enabling technology for the seamless information processing ranging from materials to clouds, and cover both Material Computing (distributed computing in materials) and Cloud Computing (distributed computing in clouds).

 ISBN 978-3-527-33606-7

## 6.4 The Mobile Agent and Multi-Agent Systems [Bosse]

One major question to be answered in large-scale and wide-area sensing networks is the strategy of decentralized data processing, data delivery, and information reduction. It is a similar issue arising in the Internet-of-Things (IoT) domain that can be extended by sensorial materials integrated in the IoT domain. New unified data processing and communication methodologies are required to overcome different computer architecture and network barriers. One common distributed and co-ordinated data processing model is the mobile agent, a self-contained and autonomous virtual processing unit, which can be organized in groups and societies of agents. The mobile agent represent a mobile computational process that can migrate in networks, e.g., the Internet domain and as well in sensor networks. Multi-agent systems (MAS) represent self-organizing societies consisting of individuals following local and global tasks and goals including the coordination of information exchange in the design and manufacturing process. A MAS is a collection of autonomous or semi-autonomous problem solvers, often solving problems pro-actively. The Agent's action and behaviour depends on an abstract model of the environment founded by ontologies. A MAS rely on communication, negotiation, collaboration, and delegation with specialization including instantiation of new agents.

Multi-Agent systems introduce a paradigm shift from passive messaging performed by active processes to active messages with mobile processes.

Summary of the most important agent behaviours and features which can be used for data processing in distributed sensor networks:

**Autonomy**

Each agent is an autonomous operation unit, independent from other agents, which either uses a virtual or a physical execution platform for data processing.

**Mobility**

An agent is capable of migration of its data and processing state (eventually including its program code) from one processing platform to a spatial or logical different platform by using containers (containing a snapshot of the agent state). After migration the agent process continues operation on the new platform at (or after) the breakpoint of the control flow at which the agent was interrupted.

**Creation**

An agent can be created at run-time. Agent creation is related with resource allocation. This includes usually a data container for private agent storage and a control container handled by the platform (code) for agent processing.

**Inheritance and Replication**

An agent can be created by another (parent) agent inheriting the control and data state of the parent agent, called forking. This feature enables replication behaviour in a group of agents.

 ISBN 978-3-527-33606-7

**Reconfiguration**

Reconfiguration can be used to adapt the agent behaviour based on learning and environmental perception to improve the system response to agent actions.

Reconfiguration of agents can be performed down to microchip level, for example, proposed in **[BOS13A]** or **[MEN05]**. But commonly adaptation of agents is limited to conditional behaviour, activity, or action selection at run-time (**[BOS13A][SAN08]**). In **[BOS12B]** code morphing (ability of a program to modify code at run-time) was applied by the agent itself to directly modify its behaviour. Code morphing with code executed on a virtual machine is attractive and suitable for creating new behaviours of agents by other agents. Code morphing techniques can be implemented efficiently on hardware level.

**Agent Interaction**

Communication of agents is required to implement cooperating and synchronized multi-agent systems. Interaction can be performed by exchanging simple messages, which requires an understanding of data types and structures of the participating agents. A tuple-space database approach with synchronized access based on pattern matching is both a simple and powerful technique providing advanced fault tolerance at run-time **[QIN10]**, especially suited for heterogeneous agent systems.

The autonomous behaviour model of mobile agents provide the capability of performing and unifiying:

- voting and negotiation,
- M-of-N and N-version systems,
- parallel operation,
- neighbourhood computations,
- replication and self organization,
- adaptability based on learning,

all contributing to the design of fault tolerant systems.

Multi-Agent systems with mobile autonomous agents provide an advanced and more intuitive methodology to model and implement distributed data processing systems including communication compared to traditional immobile program and messaging related approaches, suitable for sensor networks, proposed in **[GUI08]**.

**Distribution and Parallelization**

Parallelized image processing were successfully implemented with agent-based systems relying on the inherent computational independence of autonomous agents, the factorization capabilities of images, and agent cooperation with learning to improve the sub-task scheduling and planning **[LUE97]**. In **[YUA06]** a distributed and parallel

SHM system was implemented with agents consisting of different agents (agent behaviour classes) performing different computations in parallel. The agents operate in a wireless sensor network, those sensors were embedded in composite materials. Castendo et al. **[CAS10]** used a multi-agent architecture based on the Belief-Desire-Intention (BDI) agent model for processing the data in a distributed visual sensor network by using data fusion and local information processing instead exchanging raw image data.

There are basically three different layers of data processing in sensing applications, which require processing platforms with different computational power and storage capacity:

**Sensing**
Localized Acquisition and Preprocessing of Sensor Data, Sensor Data Fusion

**Aggregation**
Distribution and Collection of Sensor Data, globalized Sensor Information Mapping and Sensor Data Fusion

**Application**
Presentation of condensed Sensor Information, Computation, Storage, Visualization, and Interaction.

Theses different layers can be scattered in different network domains. The sensing layer is usually located in sensor networks, for example, body area networks, the aggregation layer can be found in personal and ambient area networks, and finally the application layer can be found in ambient area and wide area networks.

The characterization of sensor network features and their operational capabilities can be further divided into the following classes and terms, handled by all three layers of the sensing application, shown in Fig. **6.12**:

- Processing
- Communication
- Messaging
- Storage
- Ontologies and Data Models
- Manageability
- Security

### *6.4.1 The Agent Computation and Interaction Model*

Traditional software is composed of objects and functions specified at design time. Though there are concepts of extending programs at run-time by dynamically loading

 ISBN 978-3-527-33606-7

libraries, these programs are mainly static. Usually, the execution of software is strongly coupled to a specific execution environments and operating systems. Interpreted script languages, e.g., *JavaScript*, are more loosely coupled and can be executed on a wider variety of host architectures, operating systems, and specific computers. Finally, a multi-program system requires communication regulated by protocols. Two programs participating in communication must have significant knowledge of the understanding, meaning, and most of all the encoding of information encapsulated in communication protocols and messages.

Software agent technologies can overcome the limiting barriers in heterogeneous systems that can be affected by technical unreliability. An agent is an operational unit, usually a software process, that satisfies the following properties:

- *Autonomy*, i.e., the execution of agents do not require continuously intervention of humans and machines, and agents have the control over their inner state and the actions they perform;
- *Reactivity*, i.e., agents respond to a an environmental perception, that can be the information of the state of a communication network and the connectivity, sensor input, or platform related properties (like available storage, architecture, other agents);
- *Pro-activeness*, i.e., the actions agents can perform do not depend only on the perception, they are planned based on goals an agent should reach, e.g., the delivery of sensor data from a source to a sink node in a network;
- *Social ability*, i.e, different agents can interact with each other to reach a group goal, based on the progress and actions of other agents.

A Multi-agent system (MAS) is a collection of individual agents capable to communicate and interact with each other, mainly by exchanging messages, and acting following a collaborative goal. Self-organizing is a common approach for a MAS to operate in unreliable environments with missing or incomplete world models.

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.

Agents record information about an environment state $e \in E$ and history $h{:}e_0 \rightarrow e_1 \rightarrow e_2 \rightarrow \ldots$ Let $I = S \times D$ be the set of all internal states of the agent consisting of the set of control states $S$ related with activities and internal data $D$. An agent's decision-making pro-

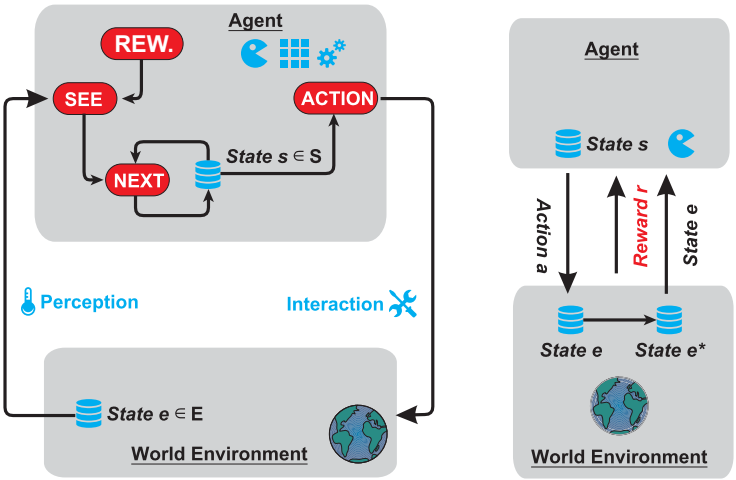
 ISBN 978-3-527-33606-7

cess is based on this information. There is a perception function ***see*** mapping environment states to perceptions, a function ***next*** mapping internal states and perceptions $p \in Per$ to internal states (state transition), and the action-selection function ***action*** which maps internal states to actions $a \in Act$, shown in Def. **6.1**.

Actions of agents modify the environment, which is seen by the agent, thus the agent is part of the environment. Software agents modify data of the environment, hardware agents like robots or Cyber-Physical Systems (CPS) modify the physical world, too. Learning agents can improve their performance to solve a given task if they analyse the effect of their action on the environment. After a performed action the agent gets a feedback in form of a reward $r(t)=r(e_t, a_t)$. There are strategies $\pi:E \rightarrow A$ which map environment states on actions. The goal of learning is to find optimal strategies $\pi^*$ that is a subset of $\pi$. The strategies can be used to modify the agent behaviour. Rewarded behaviour learning was addressed in **[JUN12]**, for example, based on Q-learning.

**Def. 6.1** *Agent processing and state change by applying three basic functions in a service loop*

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




*Agent Behaviour* models basically consisting of perception, reasoning, computation, and acting by performing discrete actions **[BUS04]**, shown in Fig. **6.14**. Agent computation models are attractive in the context of sensorial and reactive systems found in manufacturing processes due to the inherent processing of sensing data that causes actions. Actions modify the environment, i.e., commonly data stored and computations performed outside of the agent (including data and activities of other agents), but they can affect the physical world, too, regarding CPS and machine interaction.

There are basically three major behaviour model classes suitable for industrial agents **[BUS04]**:

**Reactive Agents.**


ISBN 978-3-527-33606-7

This agent model maps sensory input directly on a set of actions, i.e., the agents react immediately to perceptive data. This (*sensor,action*) mapping function can be ambiguous and conflicts and misbehavior can result. To overcome the incomplete sensor data reactivity and to provide run-time adaptability, *subsumption* architectures were proposed (Brooks, 1986), shown in in Fig. **6.14** on the upper right side. The agent behaviour is composed of different behaviours that can inhibit (block) lower prioritized behaviours, resulting in the selection of different (*sensor,action*) mapping functions. In this work, the agent behaviour can be transformed at run-time, which is a similar but more efficient and dynamic approach than inhibition used in the subsumption architecture.

**Deliberative Agents.**

Uses explicitly the more natural formulation of beliefs, desires, and intentions to select plans finally performing actions by a reasoner merging all these parts of a goal-orientated behaviour, shown in Fig. **6.14** on the lower left side. In the most common *Belief-Desire-Intention* (BDI) architecture **[WOO99]** the beliefs relies on the sensory input accumulated over time, in contrast to pure immediate reactive behaviour. The desires represent the goals of the agent, and the intentions influence future plans and actions.

**Hybrid Agents.**

They combine methods and methodologies of reactive and deliberative agents with stacked layer architectures. For example, the *InteRRap* architecture (Müller, 1996) is composed of a world interface, behaviour, plan, and co-operation *layers* propagating sensory input upwards and actions downwards.

Actions of agents modify the environment, which is seen by the agent, thus the agent is part of the environment. The change of the environment due to agent actions can be immediately or delayed, evaluating the history and past. The actions of pure reactive agents base only on the current perception, and usually an action has an effect on the environment immediately.

An agent behaviour model can be partitioned into the following tasks, which must be reflected by an agent programming language model by providing suitable statements, types, and structures:

**Computation**

One of the main tasks and the basic action is computation of output data from input data and stored data (history). Principally functional and procedural (or object-orientated) programming models are suitable, but history incorporating computed data and storage is handled only by the procedural programming model consequently.

**Communication**

Communication as the main action serves two canonical goal tuples: (data exchange and synchronization), (interaction with the environment and with other agents). The

 ISBN 978-3-527-33606-7

latter goal tuple can be reduce to agent interaction only if the environment is handled by an agent, too. Communication between agents can link single agents to a Multi-agent system, by using peer-to-peer or group communication paradigms.

**Mobility**

Mobility of agents increases the perception and interaction environment significantly. Mobile agents can migrate from one computing environment to another finally continuing there their processing. The state of an agent, consisting of the control and data state, must be preserved on migration.

**Reconfiguration**

Traditional computing systems get a fixed behaviour and operational set at design time. Adaption in the sense of behavioural reconfiguration of a system at run-time can significantly increase the reliability and efficiency of the tasks performed.

**Replication**

These are the methods to create new agents, either created from templates or by forking child agents, which inherit the behaviour and state of the parent agent, finally executing in parallel. Replication is one of the major agent behaviours to compose distributed computational and reactive systems.

**Agents and Objects**

Modern data processing is often modelled based on object-orientated programming paradigms. But there is a significant difference between agents and objects. Objects are computational units encapsulating some state and are able to perform actions (by applying methods) and communicate commonly by message passing. Objects are related with object-orientated programming and are not (or less) autonomous in contrast to agents, and they are commonly not mobile. The common object model has nothing common with proactive and social behaviours. But agents can be implemented on top of the object model with methods acting on objects. The modification of the behaviour engine is basically not supported by the object programming model. Agents decide for themselves, in contrast objects require external computational units, like operating systems or users, for the decision making process. Though object-orientated programming can be extended by parallel and concurrent processing (multi-threading), multi-agent systems are inherently multi-threaded.

To summarize, agents are characterized by their autonomy (without or less intervention of users) , reactivity, social ability, and pro-activeness, combined with behavioural adaptivity based on learning or sub-classing. Sub-classing selects a particular behaviour from a larger set of behaviours, for example, to fulfil a specific task only. This is basically reflected by the well known divide-conquer approach that splits a large problem and its solving in smaller sub-tasks.

 ISBN 978-3-527-33606-7

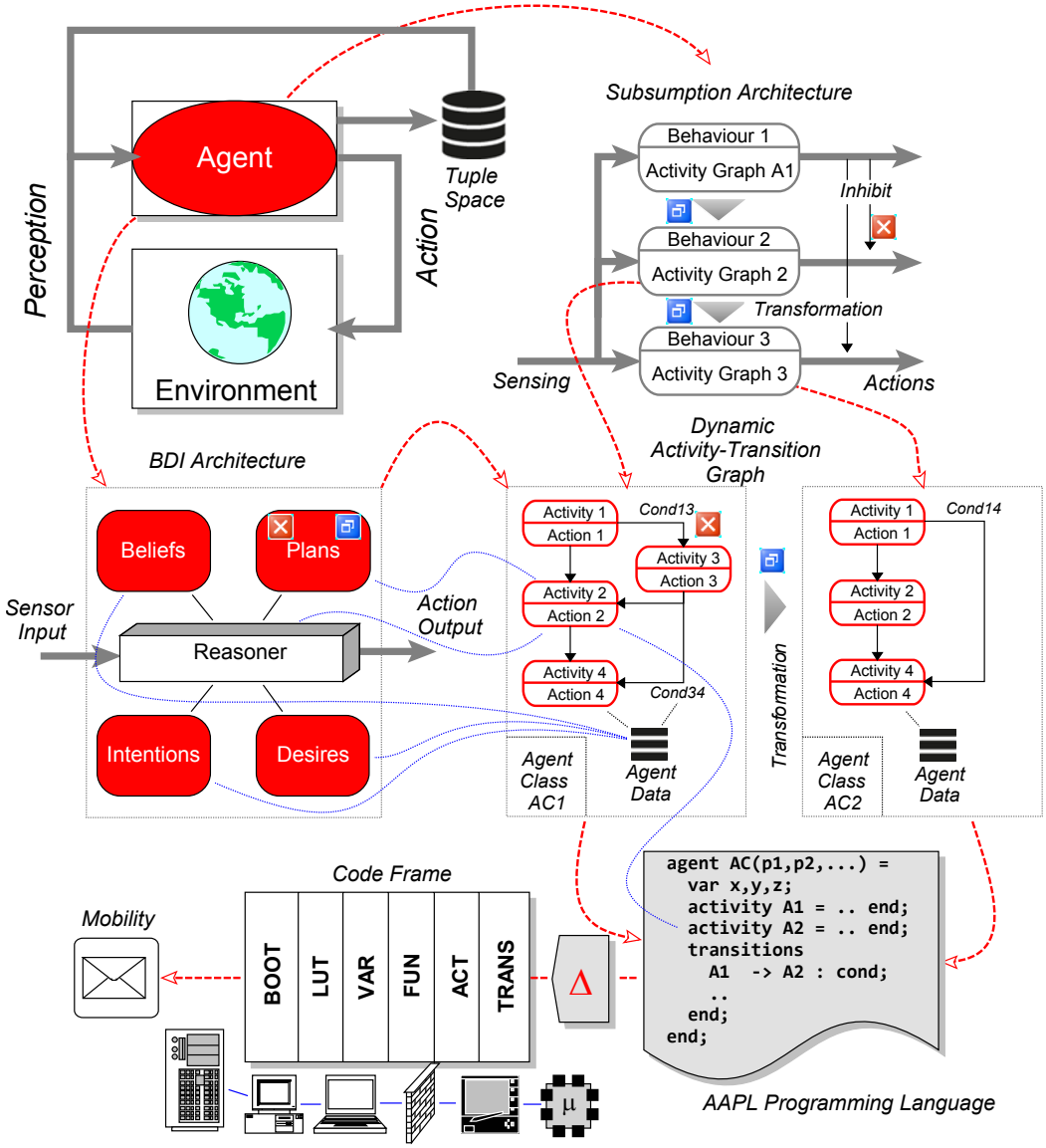



**Fig. 6.14** *Agent Models: (Left, top) Basic Agent Model, (Right, top) Subsumption Architecture composed of different behaviour, (Right, middle) Reconfigurable (Dynamic) Activity-Transition Graphs (DATG) used in this work, (Left, middle) Relation to the Belief-Desire-Intention (BDI) Architecture, (Bottom) Programming model and code frame.*

 ISBN 978-3-527-33606-7

### 6.4.2 *Dynamic 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), suitable for the behavioural modelling of reactive agents. The transitions start activities commonly depending on the evaluation of agent data (body variables), representing the data state of the agent, as shown in Fig. **6.15**.

An activity-transition graph, related to an agent behaviour class, discussed later, consists of a set of activities $\mathcal{A}=\{A_1,A_2,..\}$, and a set of transitions $\mathcal{T}=\{T_1(C_1),T_2\ (C_2),..\}$, 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 is simpler ones, based on the composition of MAS. 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.

The ATG behaviour model is closely related to the interaction of agents with the environemnt, here mainly by exchanging data by using a tuple space database, and migration. Message passing between agents is available by passing signals that execute signal handler on the destination agent asynchronously. The execution of signal handlers changes the agent data and hence has an impact on activity transitions.

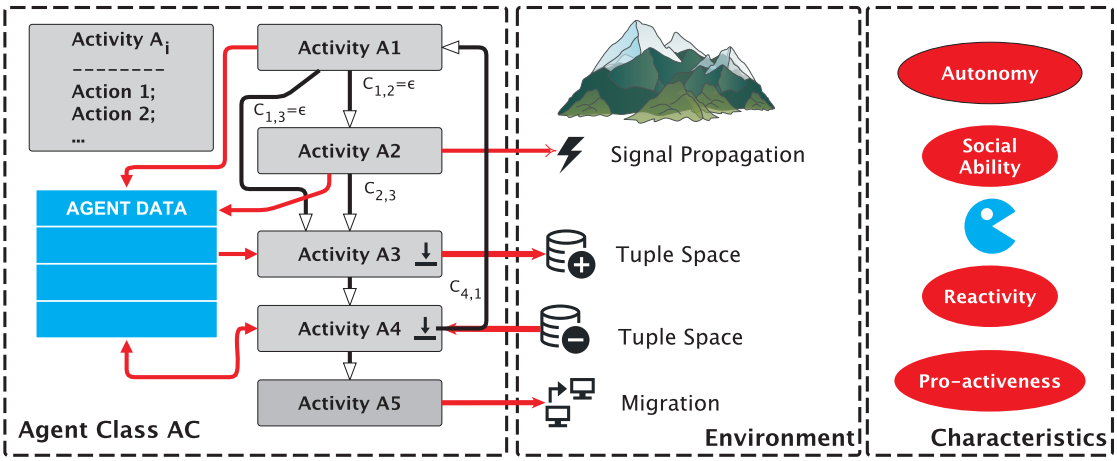



**Fig. 6.15** *(Left) Agent behaviour given by an Activity-Transition Graph and the interaction with the environment performed by actions executed within activities (Right) Agent Characteristics*


ISBN 978-3-527-33606-7

The characteristics of agents can be classified in autonomy, social ability and social interaction, reactivity with respect to changes of the environment and learning based on history and rewards, and finally pro-activeness by making assumptions about the estimated change of the environment resulting from actions performed by the agent.

An ATG describes the complete agent behaviour. Any sub-graph and part of the ATG can be assigned to a sub-class behaviour of an agent. Therefore modifying the set of activities $\mathcal{A}$ and transitions $\mathcal{T}$ of the original ATG introduces several sub-behaviours implementing algorithms to satisfy a diversity of different goals.

The reconfiguration of activities $\mathbb{A}=\{\mathcal{A}_1 \subseteq \mathcal{A}, \mathcal{A}_2 \subseteq \mathcal{A}, ..\}$ derived from the original set $\mathcal{A}$ and the modification or reconfiguration of transitions $\mathbb{T}=\{T_1, T_2, ..\}$ enables dynamic ATGs and agent sub-classing at run-time, shown in Fig. **6.16**.

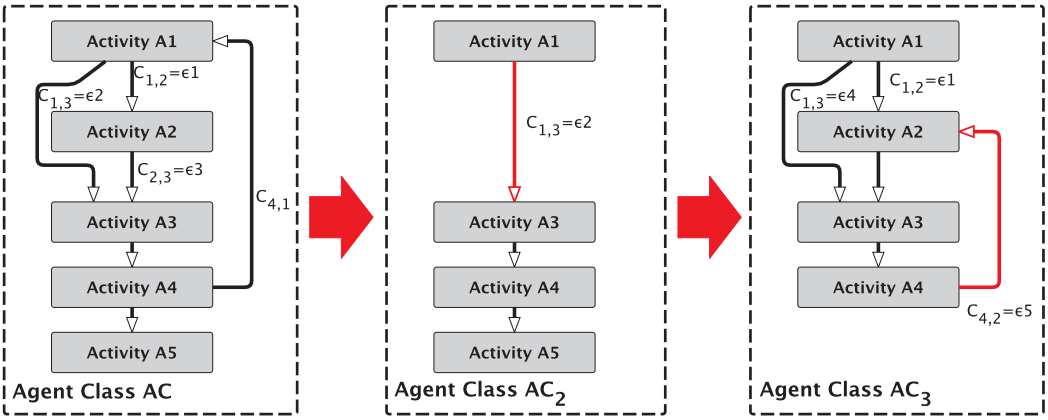



**Fig. 6.16** *Dynamic ATGs: Transformation and Composition*

### *6.4.3 The Agent Behaviour Class*

**Behaviour**. A particular agent class $\mathcal{AC}_i$ is related with the previously introduced ATG that defines the run-time behaviour and the computational action

**Perception**. An agent interacts with its environment by performing data transfer using a unified tuple-space with a coordinated database-like interface. Data from the environment influences the following behaviour and action of an agent. Data passed to the environment (e.g., the database) influences the behaviour of other agents.

**Memory**. State-based agents perform computation by modifying data. Since agents can be considered as autonomous data processing units, they will primarily modify private data, and a computational outcome using this data will be transferred to the environment. Therefore, each agent and agent class will include a set of body variables $\mathbb{V}=\{v_1{:}DT, v_2{:}DT, ..\}$, which are modified by actions in activities and read in activities and transitional expressions ( with $DT$: set of supported data types).

 ISBN 978-3-527-33606-7

**Parameter**. Agents can be instantiated at run-time from a specific agent class creating agents with equal initial control- and data states. To distinguish individual agents (creating individuals), an external visible parameter set $\mathbb{P}=\{p_1{:}DT, p_2{:}DT, ..\}$ is added, which get argument values on instantiation, enabling the creation of different agents regarding the data state. Inside an agent class, parameters are handled like variables.

To summarize, an agent class is fully defined by the *ATVPFH*-tuple:

$$
\begin{aligned}
&AC_i = \langle \mathbb{A}, \mathbb{T}, \mathbb{V}, \mathbb{P}, \mathbb{F}, \mathbb{H} \rangle \\
&\mathbb{A} = \{a_1, a_2, .., a_n\} \\
&\mathbb{T} = \{t_{ij} = t_{ij}(a_i, a_j, cond) \mid a_i \xrightarrow{cond} a_j; i, j \in \{1, 2, .., n\}\} \\
&a_i = \{i_1, i_2, .. \mid i_u \in ST\} \\
&\mathbb{V} = \{v_1, v_2, .., v_m\} \\
&\mathbb{P} = \{p_1, p_2, .., p_i\} \\
&\mathbb{F} = \{f_1, f_2, .., f_j\}, \text{ with } f_i : (x_1, x_2, ..) \rightarrow y \\
&\mathbb{H} = \{h_1, h_2, .., h_k\}, \text{ with } h_i : (v?) \rightarrow ()
\end{aligned}
\tag{6.5}
$$

with $\mathbb{F}$ as a set of computational or procedural functions, and $\mathbb{H}$ a set of signal handlers, discussed in the next section.

### *6.4.4 Communication and Interaction of Agents*

Communication and interaction is one major paradigm in distributed computing systems and enable synchronization and negotiation between parallel and concurrent processes. One suitable approach used in the agent behaviour model is the interaction of agents by exchanging data using a tuple database as a shared object supporting synchronized and atomic read, test, remove, and write operations. Agents can communicate and synchronize peer-to-peer by using signals, which can be delivered to remote execution nodes, too.

A tuple space is basically a shared memory database used for synchronized data exchange among a collection of individual agents as a suitable MAS interaction paradigm. 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 independence, vital in a heterogeneous network environment.

A tuple database stores a set of n-ary data tuples, $tp_n=(v_1, v_2,.., v_n)$, a n-dimensional value tuple. The tuple space is organized and partitioned in sets of n-ary tuple sets $\nabla=\{TS_1,TS_2,..,TS_n\}$. A tuple is identified by its dimension and the data type signature. Commonly the first data element of a tuple is treated as a key. Agents can add new tuples (output operation) and read or remove tuples (input operations) based on tuple pattern tem-

 ISBN 978-3-527-33606-7

plates and pattern matching, $pat_n=(v_1, x_2?, .., v_j,.., x_j?,., v_n)$, a n-dimensional tuple with actual and formal parameters. Formal parameters are wild-card place holders, which are replaced with values from a matching tuple and assigned to agent variables. The input operations can suspend the agent processing if there is actually no matching tuple is available. After a matching tuple was stored, blocked agents are resumed and can continue processing. Therefore tuple databases provide inter-.agent synchronization, too. This tuple-space approach can be used to build distributed data structures and the atomicity of tuple operations provides data structure locking. The tuple spaces represents the knowledge of agents.

One of the simplest interaction models are signals. They are send from one source agent and delivered to one or more destination agents. An agent receiving a signal, which can carry simple data or being pure symbolic, will process a signal by a signal handler, i.e., a function. Signal propagation is asynchronous, i.e., the sender of a signal is not aware of the successfull receiving of a signal. In contrast to tuple-space interaction signals are not reliable, which must be considered by the agent behaviour design.

### *6.4.5 Agent Programming Models*

There are already mutiple agent programming languages and processing architectures, like APRIL **[MCC95]** providing tuple-space like agent communication, and widely used FIPA ACL, and KQGML **[KON00]** focusing on high-level knowledge representations and exchange by speech acts, or model-driven engineering (e.g. INGENIAS **[SAN08]**). But required fine-grained resource and processing control is missing, preventing the deployment of agents in large-scale heterogeneous applications, which is addressed properly by the reactive *AAPL* programming language, discussed below. Furthermore, most computing algorithms are given in a conventional programming language model hard to implement with knowledge or plan based models.

#### AAPL Programming Language

The ATG-based Agent Programming langauge (*AAPL*) was designed

1. To Aid and simplify the programming of multi-agent systems based on a reactive agent behaviour model;
2. To enable direct synthesis of optimized hardware platforms from the AAPL programming level;
3. To enable software and simulation model synthesis without additional effort, too.

 ISBN 978-3-527-33606-7

*AAPL* statements are related directly to the Activity-Transition Graph model and offer

1. The definition of an agent classes containing the agent data and behaviour;
2. The instantiation (creation), management, and migration of agents;
3. The inheritance of agent behaviours and state by child agents using forking;
4. The modification of the ATG agent behaviour at run-time, based, for example, on learning;
5. The Agent interaction using signals (simple messages), basically used in coupled parent-child groups;
6. The Agent interaction using a tuple-space database providing synchronized data exchange with n-ary tuples *T*(*v1*,*v2*,...) and patterns *P*(*v1*,*p2*?,*v2*,..), basically used in loosely coupled and heterogeneous MAS;
7. The computational and control flow statements including modern exception handling.

The activity-graph based agent model introduced in the previous section is attractive due to the proximity to the finite-state machine model, which simplifies the hardware implementation of the agent processing platform.

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). An 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.

Plans are related to *AAPL* activities and transitions close to conditional triggering of plans. Table **6.1** summarizes the available language statements.

**Instantiation**: New agents of a specific class can be created at runtime by other agents using the `new AC(`$v_1$`,`$v_2$`,..)` statement returning a node unique agent identifier. An agent can create multiple living copies of itself with a fork mechanism, creating child agents of the same class with inherited data and control state but with different parameter initialization, done by using the `fork(`$v_1$`,`$v_2$`,..)` statement. Agents can be destroyed by using the `kill(`*ID*`)` statement.

Each agent has *private data* (body variables), defined by the `var` and `var*` statements. Variables in the latter statement will not be inherited or migrated! Agent body variables in conjunction with transition conditions represent the mobile data part of the agents beliefs database.

Statements inside an activity are processed sequentially and consist of data assignments (`x := ε`) operating on agent's private data, control flow statements (conditional branches and loops), and special agent control and interaction statements.

 ISBN 978-3-527-33606-7

**Agent interaction** and synchronization is provided by a tuple-space database server available on each node (based on McaCabe, 1995). An agent can store an n-dimensional data tuple ($v_1$,$v_2$,..) in the database by using the `out(`$v_1$`,`$v_2$`,..)` statement (commonly the first value is treated as a key). A data tuple can be removed or read from the database by using the `in(`$v_1$`,`$p_2$`?,`$v_3$`,..)` or `rd(`$v_1$`,`$p_2$`?,`$v_3$`,..)` statements with a pattern template based on a set of formal (variable,`?`) and actual (constant) parameters. These operations block the agent processing until a matching tuple was found/stored in the database. These simple operations solve the mutual exclusion problem in concurrent systems easily. Only agents processed on the same network node can exchange data in this way. Simplified the expression of beliefs of agents is strongly based on *AAPL* tuple database model. Tuple values have their origin in environmental perception and processing bound to a specific node location.

The existence of a tuple can be checked by using the `exist?` function or with atomic test-and-read behaviour using the `try_in/rd` functions. Tuple spaces interaction is generative, i.e., the lifetime of a tuple created by a process (agent) can exceed the lifetime of the process. A tuple with a limited lifetime (a marking) can be stored in the database by using the `mark` statement. Tuples with exhausted lifetime are removed automatically (by a garbage collector). Tuples matching a specific pattern can be removed with the `rm` statement.

**Remote interaction** between agents is provided by signals carrying optional parameters (data). Signals can be used locally, too. A signal can be raised by an agent using the `send(`*ID*`,`*S*`,`*V*`)` statement specifying the ID of the target agent, the signal name S, and an optional argument value V propagated with the signal. The receiving agent must provide a signal handler (like an activity) to handle signals asynchronously. Alternatively, a signal can be sent to a group of agents belonging to the same class AC within a bounded region using the `broadcast(`*AC*`,`*DX*`,`*DY*`,`*S*`,`*V*`)` statement. Signals implement remote procedure calls. Within a signal handler a reply can be sent back to the initial sender by using the `reply(`*S*`,`*V*`)` statement.

**Timers** can be installed for temporal agent control using (private) signal handlers, too. Agent processing can be suspended with the `sleep` and resumed with the `wakeup` statements.

**Migration** of agents (preserving the local data and processing state) to a neighbour node is performed by using the `moveto(`*DIR*`)` statement, assuming the arrangement of network nodes in a graph-like networks. To test if a neighbour node is reachable (testing connection liveliness), the `link?(`*DIR*`)` statement returning a Boolean result can be used.

**Reconfiguration**: Agents are capable to *change their transitional network* (initially specified in the transition section) by changing, deleting, or adding (conditional) transitions using the `transitionX(`$S_1$`,`$S_2$`,`*cond*`)` statements (with `X`='+':add, '-': remove, and '*':

 ISBN 978-3-527-33606-7

change transition). *This behaviour allows the modification of the activity graph, i. e., based on learning or environmental changes, which can be inherited by child agents.*

Furthermore, the set of activities can be changed at run-time using `activity`$X(A_1,A_2,..)$ statements (with $X$='+':add, '-': remove activities), creating, e.g., sub-classes on the fly. New sub-class templates can also be derived from existing class templates by using the *AC'* `:=` `new class` *AC* statement and the ATG modification statements. Note that class activities and transitions reference class variables and other functions. Therefore, only sub-classes can be composed (regarding the set of activities). The new sub-class can be initially empty by applying an additional `empty` keyword, but still referencing the super-class. The set of class parameters are identical in the super and sub-class.

**Tab. 6.1** *Summary of AAPL Statements*

| | |
|---|---|
| **Agent Class Definition**<br>agent AC($p_1$,$p_2$,..) =<br>*variables*<br>*activities*<br>*transitions*<br>*functions*<br>*handler*<br>end; | **Agent Interaction and Control**<br>out($v_1$,$v_2$,..);<br>in($v_1$,$x_2$?,$x_3$?,$v_4$,..); try_in..<br>mark(*TMO*,$v_1$,..);<br>exist?($v_1$,?,..); rm($v_1$,?,..);<br>send(*ID*,*SIG*,*arg*);<br>broadcast(*AC*,*DX*,*DY*,*SIG*,*arg*);<br>timer+(*TMO*,*HANDLER*); timer-.. |
| **Agent Instantiation and Management**<br>var *ID* := new AC($v_1$,$v_2$,..);<br>var *ID* := fork($v_1$,$v_2$,..);<br>kill(*ID*);<br>link?(*DIR*) moveto(*DIR*); | **Procedural Statements**<br>x := ε;<br>if *cond* then *stm1* else *stm2*;<br>f($v_1$,$v_2$,..);<br>for i = 1 to 100 do ..; |
| **Agent Activities and ATG Modification1**<br>activity $A_i$ =<br>*statements*<br>end;<br>activity+($A_1$,$A_2$,..);[1]<br>activity-($A_1$,$A_2$,..);[1] | **Transitions and ATG Modification**<br>transitions =<br>$A_i$->$A_j$:ε; .. end;<br>transition+($A_i$,$A_j$,ε);<br>transition-($A_i$,$A_j$);<br>transition*($A_i$,$A_j$,ε); |
| **Static ATG Sub-classing**<br>subclass AC'$_i$ =<br>*variables*<br>*activities*<br>*transitions*<br>*functions*<br>*handler*<br>end; | **Dynamic ATG Sub-classing**<br>class AC' := new [empty] class AC;<br>activity+ AC'($A_1$,$A_2$,..);[1]<br>activity- AC'($A_1$,$A_2$,..);[1]<br>transition+ AC'($A_i$,$A_j$,ε);<br>transition- AC'($A_i$,$A_j$); |

[1] Not supported/available on the PCSP Platform Architecture


ISBN 978-3-527-33606-7

An example for an AAPL agent class implementation is shown in Ex. **6.1**. The explorer agent has the goal to collect sensor data in a network region of interest and to recognize a significant stimulus using a strategy basing on an divide-and-conquer approach with child explorer agents.

This agent class uses dynamic ATG reconfiguration for the creation of child explorer agents at run-time, mainly performed in the activities `percept` and `percept_neighbour`. Child and parent agents communicate via the tuple data base and signals.

**Ex. 6.1** *AAPL Sensor Explorer Agent Class Template*

```
type direction = {NORTH,SOUTH,EAST,WEST,ORIGIN};

type keys = {ADC,FEATURE,H,MARK};
val TMO := 1 sec;
val MAXLIVE := 15;
val RADIUS := 4;
val ETAMIN := 4;
val ETAMAX := 8;
val DELTA := 10;

signal WAKEUP,TIMEOUT;
agent explore(dir: direction, radius: integer[1..16]) =
  var dx,dy:integer[-100..100];  long term data
      live:integer[0..15];
      h:integer[0..50];
      backdir: direction;
      s0: integer[0..1023];
      group: integer[0..255];
      b:boolean;

  var* s: integer[0..1023]; short term data
       enoughinput: integer[0..15];
       again: boolean;

  activity start =
    dx := 0; dy := 0; h:= 0;
    if dir <> ORIGIN then
      moveto(dir);
      case dir of
        | NORTH -> backdir := SOUTH;
        | SOUTH -> backdir := NORTH;
        | WEST ->  backdir := EAST;
        | EAST ->  backdir := WEST;
      end;
    else
      live := MAXLIVE;
      backdir := ORIGIN;
    end;
    group := random(integer[0..1023]);
```

 ISBN 978-3-527-33606-7

```
    transition*(move,percept_neighbour);
    out(H,id(self),0);
    rd(ADC,s0?);
  end;

  activity move =
    case dir of
      | NORTH -> backdir := SOUTH; incr(dy);
      | SOUTH -> backdir := NORTH; decr(dy);
      | WEST ->  backdir := EAST;  decr(dx);
      | EAST ->  backdir := WEST;  incr(dx);
    end;
    moveto(dir);
  end;

  activity diffuse =
    decr(live);
    rm(H,id(self),?);
    if live > 0 then
      case backdir of
        | NORTH -> dir := random({SOUTH,EAST,WEST});
        | SOUTH -> dir := random({NORTH,EAST,WEST});
        | WEST -> dir := random({NORTH,SOUTH,EAST});
        | EAST -> dir := random({NORTH,SOUTH,WEST});
        | ORIGIN -> dir := random({NORTH,SOUTH,EAST,WEST});
      end;
    else kill(ME); end;
  end;

  activity reproduce =
    var n:integer;
    stay here, increase feature counter
    rm(H,id(self),?);
    if exist?(FEATURE,?) then in(FEATURE,n?); else n := 0; end;
    out(FEATURE,n+1);
    decr(live);
    after some time perform perception again...
    if live > 0 then
      for nextdir in direction do
        if nextdir <> backdir and link?(nextdir) then
          eval(fork(nextdir,radius));
        end;
      end;
    end;
    transition*(reproduce,stay);
  end;

  Master perception
  activity percept =
    Send out child agents to explore
    the neighbourhood and calculate partial H values
```

 ISBN 978-3-527-33606-7

```
    enoughinput := 0;
    transition*(percept,move);
    for nextdir in direction do
      if nextdir <> backdir and link?(nextdir) then
        incr enoughinput;
        eval(fork(nextdir,radius));
      end;
    end;
    transition*(percept,diffuse,
                (h<ETAMIN or h > ETAMAX) and enoughinput < 1);
    transition+(percept,reproduce,
                h>=ETAMIN and h <= ETAMAX and enoughinput < 1);
    Wait for child agents delivering h values or timeout
    timer+(TMO,TIMEOUT);
  end;

  Child perception
  activity percept_neighbour =
    if not exist?(MARK,group) then
      mark(TMO,MARK,group);
      enoughinput := 0;
      rd(ADC,s?);
      out(H,id(self), calc());
      transition*(percept_neighbour,move);
      for nextdir in direction do
        if nextdir <> backdir and inbound(nextdir) and link?(nextdir) then
          incr enoughinput;
          eval(fork(nextdir,radius));
        end;
      end;
      transition*(percept_neighbour,goback, enoughinput < 1);
      Wait for child agents delivering h values or timeout
      timer+(TMO,TIMEOUT);
    else
      transition*(percept_neighbour,goback);
    end;
  end;

  activity goback =
    if exist?(H,id(self),?) then in(H,id(self),h?); else h := 0; end;
    moveto(backdir);
  end;

  activity deliver =
    var v:integer;
    in(H,id(parent),v?);
    out(H,id(parent),h+v);
    send(id(parent),WAKEUP);
    kill(ME);
  end;

```

 ISBN 978-3-527-33606-7

```
  activity stay =
    Stay here and wait
  end;

  handler WAKEUP =
    decr enoughinput;
    eval(try_rd(0,H,id(self),h?));
    if enoughinput < 1 then timer-(TIMEOUT); end;
  end;

  handler TIMEOUT =
    enoughinput := 0;
    again := true;
  end;

  function calc():integer =
    if abs(s-s0) <= DELTA then return 1; else return 0; end;
  end;

  function inbound(nextdir:direction):bool =
   case nextdir of
      | NORTH -> return (dy < RADIUS);
      | SOUTH -> return (dy > -RADIUS);
      | WEST ->  return (dx > -RADIUS);
      | EAST ->  return (dx < RADIUS);
    end;
  end;

  transitions = Some transitions are changed or added at run-time
    start -> percept;
    percept -> move;
    move -> percept_neighbour;
    percept_neighbour -> move;
    diffuse -> start;
    reproduce -> start;
    goback -> deliver;
    diffuse -> start;
  end;
end;
```

### 6.4.6 *Agent Processing Platforms and Technologies*

An agent processing platform must provided the following capabilities and services:

1. Execution of agents;
2. Instantiation of new agents at run-time;
3. Support for migration of mobile agents;
4. Provision of agent interaction services.

 ISBN 978-3-527-33606-7

One common agent platform is JADE that bases on the Java VM bytecode platform, and using AgentSpeak/Jason as a programming framework, discussed in **[BOR06][CHU02]**. Though this platform architecture is publicly available and attractive because programmers can reuse a well known programming language, this approach is limited to full equipped computers offering a sufficient amount of resources (Main Memory > 100 MB, Computational Power > 100 MIPS), and is not suitable for low-resource embedded systems. Low-resource and efficient agent platforms play a key role in distributed sensor networks that are embedded in materials. To enable the design of material-integrated sensing systems, a new low-resource agent platform is introduced. This platform can be entirely implemented in hardware (HW platform technology class). Alternatively, there are also software implementations (SW platform technology class). All technology classes (HW/SW) are fully compatible on operation level supporting agent migration between hardware and software platforms.

Agents are already deployed successfully for scheduling tasks in production and manufacturing processes **[CAR00]**, and newer trends poses the suitability of distributed agent-based systems for the control of manufacturing processes **[LEI15]**, 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 **[LEI15]**. Currently, many traditional processing platforms cannot yet handle big numbers with the robustness and efficiency required by industry **[MAR05][PEC08]**. In the past decade the capabilities and the scalability of agent-based systems have increased substantially, especially addressing efficient processing of mobile agents.

Multi-agent systems are successfully deployed in sensing applications, for example, structural load and health monitoring, with a partition in off- and online computations **[BOS15B]**. Distributed data mining and Map-Reduce algorithms are well suited for self-organizing MAS. Cloud-based computing, for example, as a base for cloud-based manufacturing, means the virtualization of resources, i.e., storage, processing platforms, sensing data or generic information. Hence, agent processing platform are logical virtualized processing environments, hiding technical details of the underlying physical execution platforms (microprocessors, WEB browser, digital logic,..).

**Agent-On-Chip Architectures.** Traditionally agent programs are interpreted, leading to a significant decrease in performance. Agent-on-chip processing platform can be directly implemented in standalone hardware nodes without intermediate processing levels and without the necessity of an operating system, but still enabling software

 ISBN 978-3-527-33606-7

implementations that can be embedded in applications (e.g., **[BOS14B]**). The common agent behaviour and programming ATG/*AAPL* model enables the processing of large-scale multi-agent systems on hardware and software platforms deployed in heterogeneous networks. Agents are mobile units and can migrate between processing nodes within the network, which must be supported by the agent processing platform. The agent behaviour "program" can be changed at run-time by agents, for example, based on learning or sub-classification. Very low-resource and high-efficiency platforms must implement the agent behaviour application-specific by using high-level synthesis to map the agent behaviour of multi-agent systems entirely on microchip-level.

There are programmable agent processing platforms (basing on machine virtualization) that can be deployed in strong heterogeneous network environments **[BOS15A]**, too, ranging from 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. A programmable platform is more flexible than the application-specific, but requires more resources (chip area, power, storage).

*Support for Heterogeneous Networks*

Commonly different Hardware and Software Agent Platforms are connected in one network or network graphs. Logical networks of nodes servicing a specific task are mapped on physical platforms differing in computational capabilities, size and power requirements as shown in Fig. **6.17**.

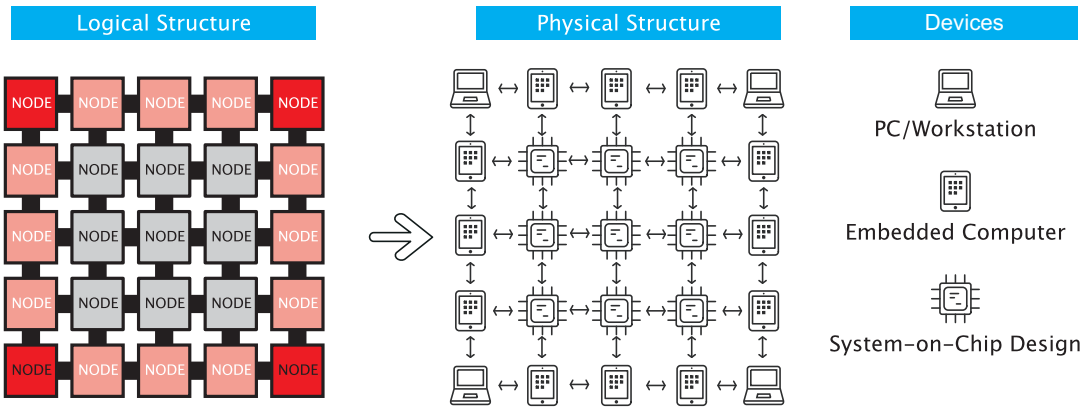



**Fig. 6.17** *Mapping of logical on physical node structures*

Agents must be capable to migrate between different physical platforms! Furthermore, the agents goals and behaviour define the task to be performed requiring generic agent processing platforms suitable for a wide range of different agent classes. Material-integrated sensor networks as part of a global sensing systems will only perform a reduced set


ISBN 978-3-527-33606-7

of tasks, mainly related to the sensor level. Therefore, the deployment of the application-specific platform implementing the agent behaviour directly can be considered, offering low-resource platforms capable for microchip scaling.

*Programmable versa application-specific platforms*

There basically two different platform architectures that can be used for the processing of large-scale MAS: programmable platforms based on mobile program code that contains the complete agent behaviour and agent state, and non-programmable platforms implementing the agent behaviour application-specific in the platform directly, and only the agent state is mobile.

In **[BOS14A]** an application-specific (*PCSP*), and in **[BOS15A]** a programmable (*PAVM*) agent processing platform were proposed. Both are capable of supporting multiple agent behaviour classes with a large number of instantiated agents and bases on a token-based agent scheduling offering advanced resource sharing and low-resource requirements.

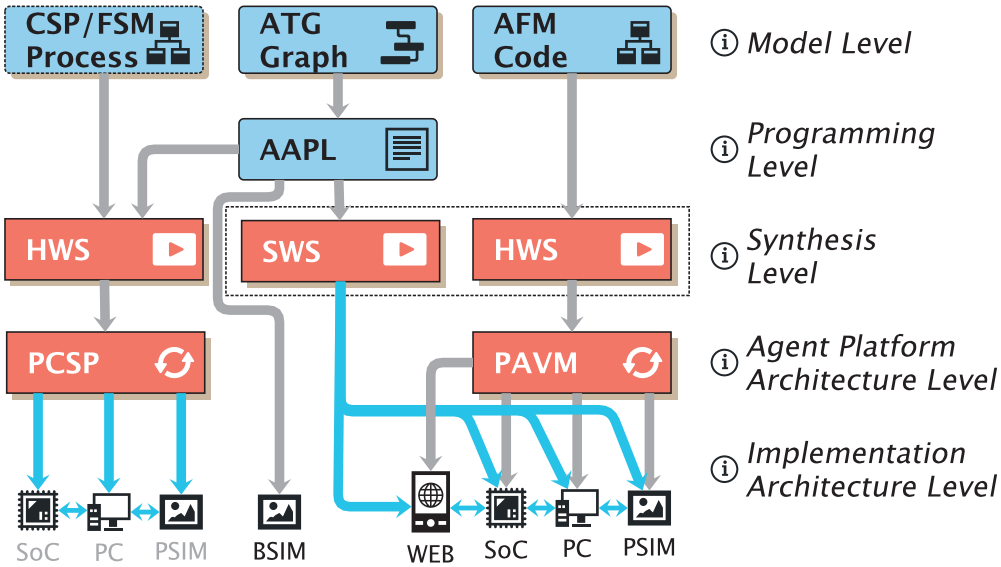



**Fig. 6.18** *PCSP and PAVM Agent Processing platforms designed from a common programming level*
*[CSP: Communicating Sequential Processes, FSM: Finite-State Machine, ATG: Activity-Transition Graph, AFM: Agent Forth Machine, AAPL: ATG Agent Programming Language, HWS: Hardware Synthesis, SWS: Software Synthesis, PCSP: Pipelined Communicating Sequential Processes, PAVM: Pipelined Agent Virtual Machine]*

 ISBN 978-3-527-33606-7

*PCSP Platform*

Towards the implementation of the (hardware) agent processing platform and for ATG analysis the ATG is transformed in a State-Transition (ST) Petri-Net (PN). Activities are mapped on states of the PN, conditional transition expressions and transition scheduling are merged with the activity states! Agents are represented by tokens passed by transitions between states of the PN. Only one token can be consumed by an activity state.

The PN is finally mapped on a Communicating Sequential Process Model architecture. The original activities are mapped on these sequential processes and the individual agents are represented with the aforementioned tokens, giving an unique agent handlers and identifiers, passed by queues between activity processes offering a Pipelined CSP architecture.

In some cases process transformations must be applied to implement the correct behaviour of *AAPL* statements. There are differences in pure computational and I/O related activities. I/O statements can block (suspend) the agent processing until an event occurs.

Timed Petri-Net analysis can be used to optimize the execution of multiple agents in the same activity state and belonging to one agent class AC. Processes of computational activities with high computation time can be re-factored and split into smaller units and processes with intermediate queues , and/or replicated to enable parallel agent processing to improve the overall pipeline throughput.

*Hardware Platform and High-level Synthesis*. The HW platform design is application specific with static resources implementing all supported agent classes for a maximal number of agents, which must be fixed at design-time. All parts are integrated in one System-on-Chip (SoC) design on RT level. The HW platform contains all components required to create, process, and migrate agents:

- Agent Manager
- Agent Management and Configuration Tables
- Communication Modules
- Agent Data
- Agent Processing Pipelines (one for each supported agent class)

A multi-stage High-level Synthesis approach is used to map the *AAPL* source code specification to micro-chip SoC level. Creation of agents at run-time is performed by creating an agent handler (token) Migration of agents is performed by transferring the state of the agent (data state of all body variables and control state à next activity after migration).

The processing architecture consists of pipelined communicating processes. Individual agents are represented by tokens (natural numbers) that are transferred by queues from an outgoing activity to another in-coming activity depending on the activity transitions specified in the *AAPL* model. An activity is implemented with a finite-state machine consuming

 ISBN 978-3-527-33606-7

an agent token from the input queue, processing the activity statements, and finally passing the token to an outgoing queue. There is only one in-coming transition queue for each process. There are two different kinds of activities: pure computational non-blocking activities, and event based activities which can block the agent processing until an event happened, for example the availability of external data or a time delay. These two activity processing kinds require different implementations.

Agent interaction is provided (locally on network node level) by a tuple-space dictionary (look-up table) and globally by using signals. Pending agent signals are checked each time an agent token was removed from a queue. The tuple dictionary operations are event-based statements that can block the agent processing. Thus *IO*/event-based processes remove an agent token from the input queue, check the availability of requested data or the happening of an event, and in the case the request can not be satisfied, the agent token is returned to the input queue, and the next agent token (if any) can be processed.

Local agent data is stored in a region of a memory module assigned to each agent and addressed by the agent token number. Each agent class is assigned to its own memory module. An agent is started by an agent manager that transfers the agent token to the start queue or in the case of a migrated agent in a different queue respective to the last agent control state. Terminating or migrating agents are handled by a manager queue.

 ISBN 978-3-527-33606-7

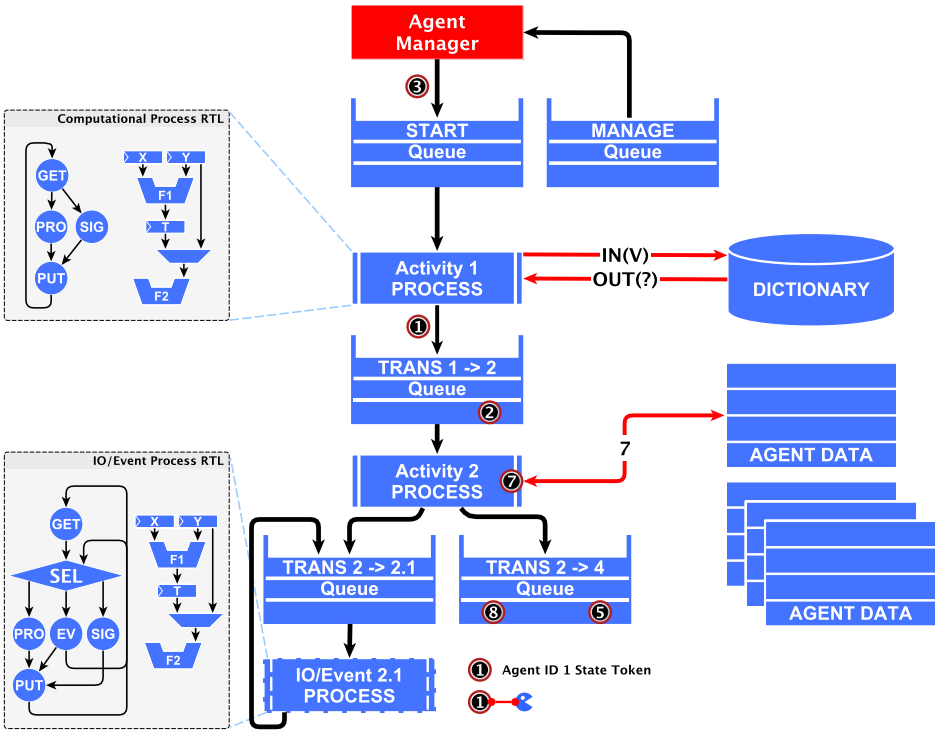



**Fig. 6.19** *Agent implementation with a pipelined multi-process architecture allowing operational unit sharing. Sequential processes are assigned to agent activities (representing states), queues implement state transitions. Inter-agent communication is performed by using a dictionary. Local agent data is stored in a region of a memory module assigned to each agent.*

*Software Platform.* For performance reasons and the lack of fine-grained parallelism the ATG of an agent class is implemented differently:

- Each activity of an agent class and transitions with cond. expessions are implemented with functions:
  $A_i$ ➠ `FA`$_i$`(){I`$_1$`;I`$_2$`;..;return;}`
  $T_{x,y}$ `| Cond` ➠ `FC`$_{x,y}$`{return cond;}`
- Transitions are stored in dynamic lists:
  ```
  struct tl {void (*from)(); void (*to)(); int (*cond)();
              struct tl *next;};
  ```
- Modification of the transitional network modifies the transition lists

 ISBN 978-3-527-33606-7

- Transitions between activities are handled by a transition scheduler which calls the appropriate activity functions:
  `while(!die) {next=schedule(curr,tl); curr=next; curr()}`
- Multi-threading: At run-time each agent is assigned to a separate thread executing the transition scheduler
- Operating system or multi-threading primitives are used to synchronize agents (event, Mutex, Semaphore, timer).
- Creation of agents at run-time is performed by creating an agent handler thread
- Migration of agents is performed by transferring the state of the agent (data state of all body variables and control state à next activity after migration)

Blocking of agent processing is handled by the thread implementation itself or by using a dedicated activity block scheduler. Further software modules implement the agent manager, tuple space databases, and networking.

*Simulation Platform.* The SeSAm [Klügel et al.] simulator environment is used to perform functional testing and analysis of multi-agent systems operating in distributed sensor networks, shown in Fig. **6.20**. The SeSAm simulator uses an agent behaviour model based on ATGs, similar to the model already introduced. But: 1. the SeSAm ATG model is static and fixed at simulation-time, 2. activities may not block, 3. there are no signal handlers. For this reason a transition and a signal scheduler are required that handle the agent processing. Activities with blocking statements (I/O) must be split according to the PCSP model. Migration of agents is performed by changing the spatial location!

 ISBN 978-3-527-33606-7

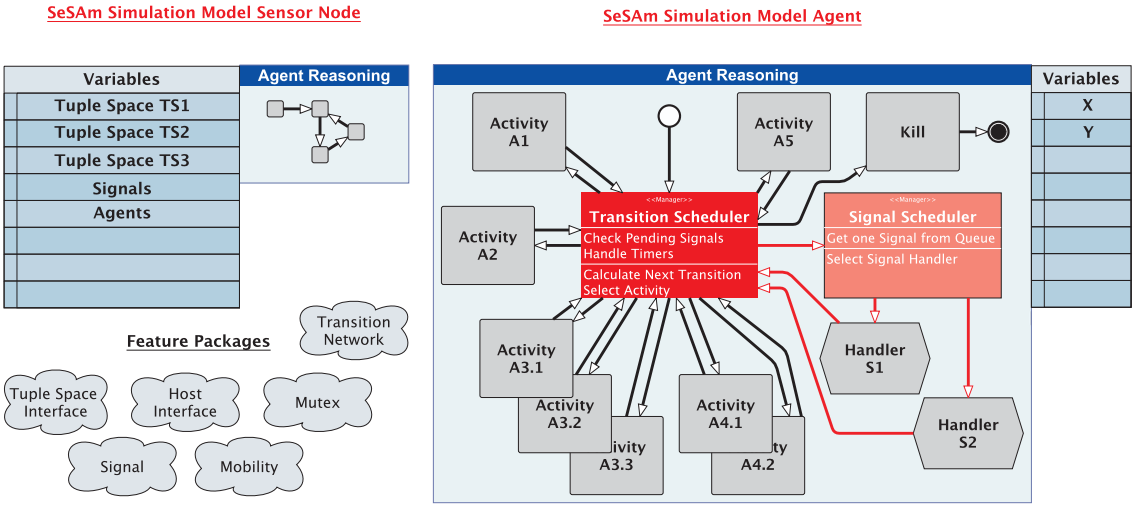



**Fig. 6.20** *Simulation of AAPL based agents in the SeSAm Simulator uisng a transition and signal scheduler*

*PAVM Platform*

This ia a Pipelined Agent Forth Virtual Machine with multiple stack-based program code processors. Agent processing is performed again with tokens passed by queues to the Forth processor offering fine grained agent scheduling. There are also different platform implementations: Hardware, Software, Simulation, WEB JavaScript App. They support compatibility on interface and operational level offering agent mobility crossing different platform implementations. The *AAPL* is the common and unified agent behaviour model and programming source for all implementations as well architectures, shown in Fig. **6.18**!

*PAVM Virtual Machine and Code Architecture*

- The virtual machine (VM) executing tasks is based on a traditional FORTH architecture, shown in Fig. **6.21**.
- It uses an extended zero-operand word instruction set (αFORTH) commonly operating on stacks rather on RAM
- Memory architecture of a VM:
  - Data (*TS*) stack used for data operations (computation)
  - Control (*RS*, return) stack used for program flow control
  - Code segment (*CS*) storing the program code with embedded data.


ISBN 978-3-527-33606-7

- The program is mainly organized by a composition of words (functions) stored in a code frame that embeds the agent data and control state, too.
- 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).
- 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 passed back to the processing queue, and another agent task can be processed.

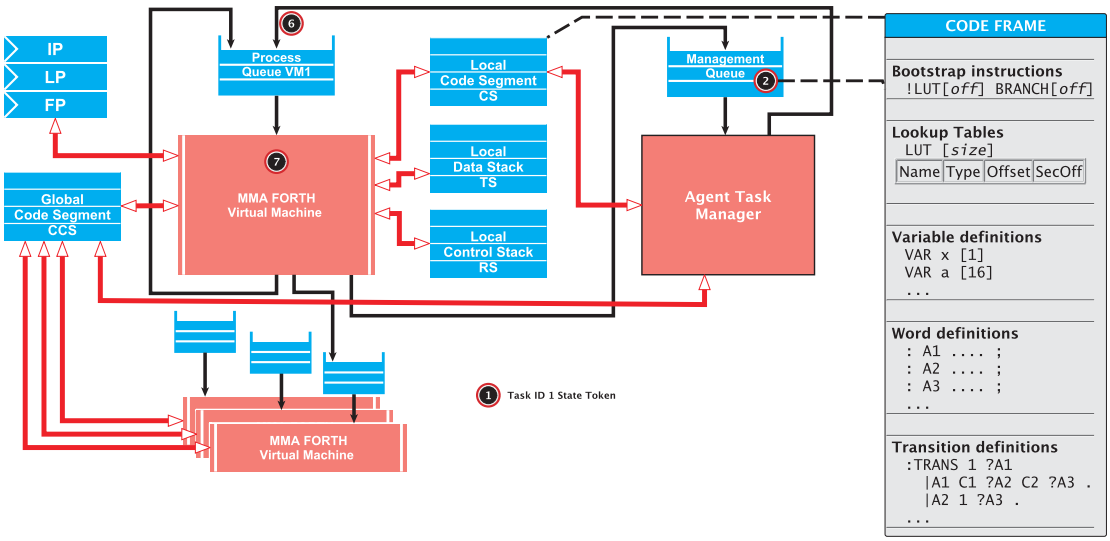



**Fig. 6.21** *Pipelined Multi-VM Architecture and Agent Program Code Frame*

*Comparison.* Both platform architectures are compared in the following Table **6.2**. Deploying of both architectures in one network is non trivial due to the completely different representations of agents, basically code with embedded behaviour, data and state versa data and state only in the *PCSP* architecture. The mixing of both architectures require agent wrapper modules transforming the data/control state of PCSP agents to an equivalent code frame that can be processed by the *PAVM* architecture and vice versa.

 ISBN 978-3-527-33606-7

| | Programmable PAVM | Non-programmable PCSP |
|---|---|---|
| **Approach** | • Program code based approach<br>• Platform is generic<br>• Code embeds instructions, configuration (control state), and data<br>• Migration: code transfer<br>• Zero-operand instruction format<br>• Stack-based Data processing model | • Application-specific approach<br>• Platform is application-specific and implements agent behaviour<br>• Activities of the ATG are mapped to processes<br>• Token-based agent processing<br>• Migration: data and control state transfer |
| **Hardware** | • Optimized Multi Stack Machine<br>• Each stack processor has a local code segment and two stacks shared by all agents. There is no data segment!<br>• Single SoC Design<br>• Multiprocessor architecture with distributed and global shared code memory<br>• Multi-FSM RTL hardware architecture<br>• Automatic Token-based agent process scheduling and processing<br>• Code morphing capability to modify agent behaviour and program code (ATG modification)<br>• Data- and code word sizes can be parameterized | • Pipelined Communicating Processes Architecture composition implementing ATG and token-based agent processing<br>• Single SoC Design<br>• Optimised resource sharing - only one PCSP for each agent class implementation required<br>• Activity process replication for enhanced parallel agent processing<br>• For each agent class there is one PCSP with attached data memory (agent data).<br>• Single SoC Design<br>• LUT configuration matrix approach for ATG reconfiguration |
| **Software** | • Multi-Threading or Multi-Process software architecture<br>• Multi word size support<br>• Inter-process-communication: queues<br>• Software model independent from programming language<br>• VM sources for various programming languages: C, ML, JavaScript, ...<br>• Can be embedded in existing software<br>• JavaScript platform offers WEB application integration | • Multi-Threading software architecture<br>• Optimization: Functional composition and implementation of ATG behaviour instead PCSP<br>• Inter-process-communication: queues<br>• Software model independent from programming language<br>• Source code for various programming languages: C, ML, ...<br>• Can be embedded in existing software |
| **Simulation** | • Agent-based Platform simulation<br>• Generic simulation model - can execute machine code directly<br>• Processor components and managers are simulated with agents | • Agent-based platform simulation<br>• Application-specific simulation model<br>• ATG activity processes are simulated with agents |

**Tab. 6.2** *Comparison of the PAVM and PCSP platform architecture and their implementations*


ISBN 978-3-527-33606-7

*JAM Platform*

This ia a *JavaScript* (*JS*) implementation of the Agent Virtual Machine based on the *PAVM* architecture but supporting the direct processing of *AgentJS* agents. *AgentJS* is the direct mapping of *AAPL* statements and the underlying programming model to JavaScript (*JS*). In contrast to the *PAVM* platform requires *JAM* a host platform for operation, commonly a generic computer, a mobile device (smart phone), an embedded computer, or a server. *JAM* (details in **[BOS15C]**) was intended for the deployment of mobile agents in large-scale and large-domain networks and environments, i.e., the Internet, Clouds, and the Internet-of-Things, shown in Fig. **6.22**. *JAM* completes the set of agent processing architectures, and as it bases on the *AAPL* model, it is partially inter-operable with the *PAVM* and *PCSP* platform architecture by using code or state transformation modules.

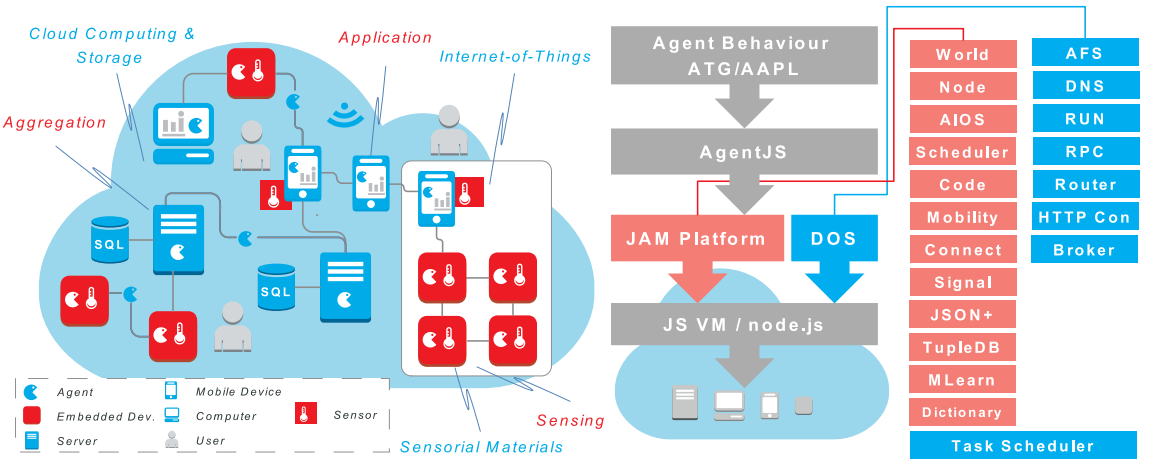



**Fig. 6.22** *JAM AgentJS processing platform: (Left) Deployment in the Internet and embedded networks, (Right) JAM and DOS modules*

*JAM* consists of different modules, shown in Fig. **6.22**, entirely implemented in *JS* that can be executed by any standalone *JS* VM or within WEB browsers. The deployment in Internet and client-side applications like browsers and hidden in virtual networks require a Distributed Co-ordination and Operation System layer (DOS) with a broker service. The main advantage of *JAM* is the unified deployment in the Internet domain.

The *AIOS* is the main execution layer of *JAM*. It consists of the sandbox execution environment encapsulating an agent process, with different privileged sub-sets depending of an agent role (level 0,1,2). Furthermore, the *AIOS* module implements the agent process scheduler and provides the API for the logical (virtual) world and node composition. The sandbox environment provides restricted access to a code dictionary based on the privilege level, enabling code exchange between agents Level 0 agents are not privileged to replicate, create, or kill other agents and to modify their code.

 ISBN 978-3-527-33606-7

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

In contrast to the *AAPL* model based on the Activity-Transition-Graph (ATG) model that supports multiple blocking statements (e.g., IO/tuple-space access) inside activities, *JS* is not capable of handling any kind of process blocking (there is no process and blocking concept). For this reason, scheduling blocks can be used in *AgentJS* activity functions handled by the *AIOS* scheduler. Blocking *AgentJS* functions returning a value use common callback functions to handle function results, e.g., `inp(pat,function(tup){..})`.

Agent mobility, provided by the *AIOS* `moveto(dst)` statement, requires a process snapshot and the transfer of the data and control state of the agent process. The control state of an agent is stored in a reserved agent body variable `next`, pointing to the next activity to be executed. The data state of an *AgentJS* agent consists only of the body variables. Thus, the migration starts with a code-to-text transformation to the extended *JSON+* representation of the agent object, transportation of the text code to another logical or physical node, and a back text-to-code conversion with a new sandbox environment. The agent object is finally passed to the new node scheduler and can continue execution.

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

*SEJAM: JAM Simulator World*

Commonly, execution and simulation platforms are completely different environments, and simulators are significantly slower in the agent execution compared to real-world agent processing on optimized processing platforms. *SEJAM* is a *JS-APPL* simulator implemented on top of the *JAM* platform layer, executing agents with the same VM as a standalone agent platform would do.

This capability leads to a high-speed simulator, only slowed down by visualization tasks and user interaction. Furthermore, multiple simulators can be connected via a stream link (sockets, IP network connection, etc.), improving the simulation performance by supporting parallel agent processing. Furthermore, the simulator can be directly connected to any other *JAM* node.

 ISBN 978-3-527-33606-7

The GUI of the simulator and the simulation world is shown for an example setup in Fig. **6.23**. The GUI consists of the simulation world, composed of 64 logical nodes connected with virtual circuit links. Each node shape provides information about the node name in the first row, the number of agents and tuples in the database in the second row, and some flag indicators in the last row, e.g., flags signaling the existence of specific agents or sensor values.

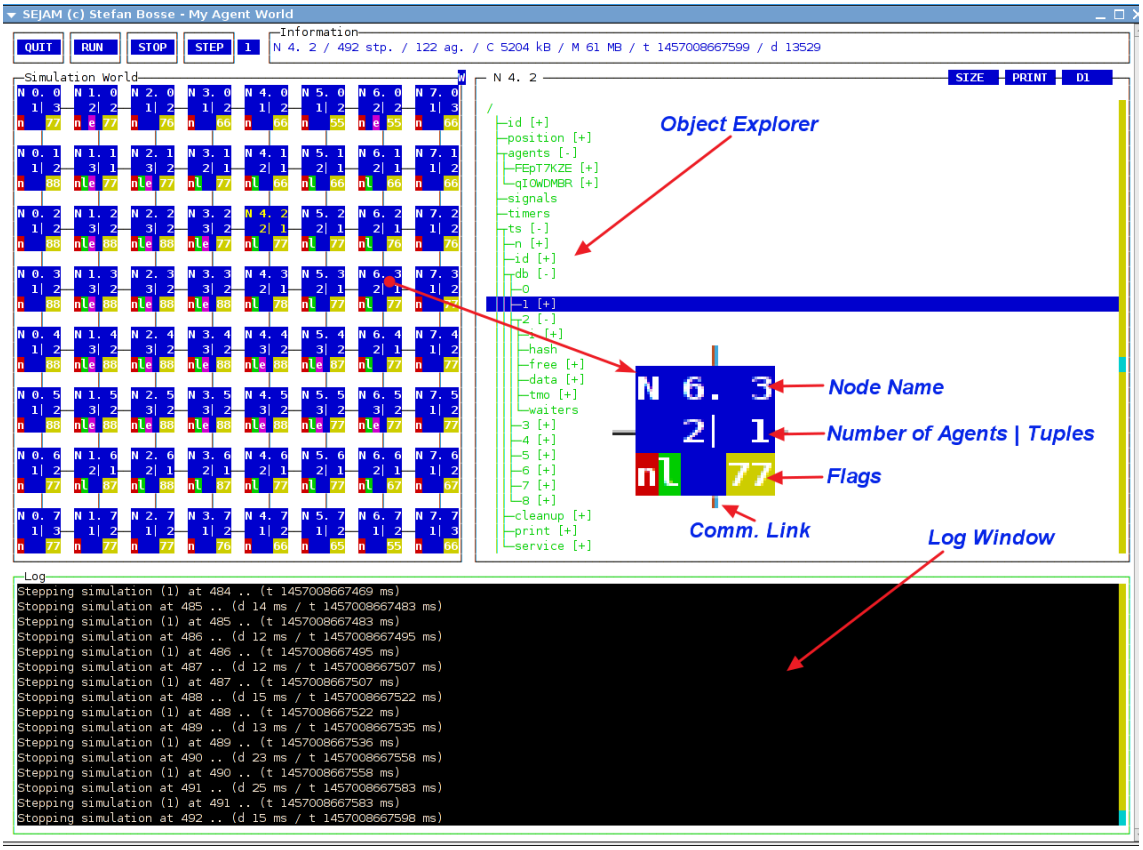



**Fig. 6.23** *SEJAM simulation demonstration with a simulation world consisting of 8x8 logical nodes populated with mobile and non-mobile agents, indicated by markings on the bottom of the node shape (blue rectangle).*

On the right side there is a code and data navigator. Each node can be selected including the world object. The code navigator can be used to explore node and agent information in a *JSON*-like tree presentation. The bottom part of the simulator contains a logging and message window. Agents can write messages to this window, and a compacted *JSON* can be printed from selected items in the object navigator tree. Furthermore, agents executed in the simulator world inherit a special simulation object, which can be used to get specific

 ISBN 978-3-527-33606-7

simulation and world information, e.g., the current simulation step, or support for creation of agents on a specific node, e.g., used by the world agent, that is the only agent created and started at the beginning of the simulation. Multiple simulator worlds (*SEJAM* instances) can be connected enabling the composition of complex simulation worlds.

### 6.4.7 *Agent-based Learning*

Traditional Machine Learning is partitioned in two separated actions: (1) Collecting of data; (2) Learning or Classification using this data. Supervised Machine Learning require additional labeld (tagged) data, assigning a specific class symbol to a dataset, whereas unsupervised learning require a feedback from the environment to evaluate a given hypothesis. It is well known that unsupervised learning can be integrated in the reactive agent model (Agent Learning), but Multi-agent systems can create a bridge between traditional Agent and Machine Learning, shown in Fig. **6.24**. A distributed MAS can be used to implement a multi-instance ML task, which is originally only implemented by a single ML instance. An example for a distributed multi-instance ML using MAS is outlined in the following section.

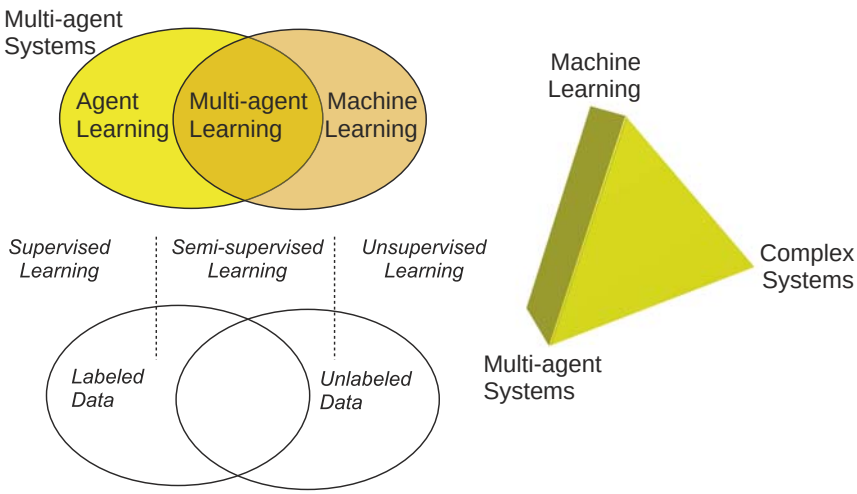



**Fig. 6.24** *Classification of agent-based Machine Learning and the synergy triangle of classical Machine Learning, Multi-agent Systems, and complex systems.*

### 6.4.8 *Event and Distributed Agent-based Learning of noisy Sensor Data*

In **[BOS16A]** it was shown that three different data processing and distribution approaches can be used and implemented with agents, leading to a significant decrease in

 ISBN 978-3-527-33606-7

network communication activity and a significant increase of the reliability and Quality-of-Service:

1. An event-based sensor distribution behaviour is used to deliver sensor information from source sensor to computation nodes based on local decision and sensor change predication.
2. Adaptive path finding (routing) supports agent migration in unreliable networks with missing links or nodes by using a hybrid approach of random and attractive walk behaviour
3. Self-organizing agent systems with exploration, distribution, replication, and interval voting behaviours based on feature marking are used to identify a region of interest (ROI, a collection of stimulated sensors) and to distinguish sensor failures (noise) from correlated sensor activity within this ROI.

In Structural Monitoring applications, sensor nodes are commonly arranged in some kind of a two-dimensional grid network (as shown in Fig. **6.25**) and they provide spatially resolved and distributed sensing information of the surrounding technical structure, for example, a metal plate or a composite material equipped with strain sensors. Usually a single sensor cannot provide any meaningful information of the mechanical structure. One example for an information mapping discussed in Sec.**5.3** uses inverse numerical computation to determine load situations. An alternative approach is Machine Learning used to classify different learned load situations, for example, to distinguish between normal load and overload conditions. In the following use case distributed learning is applied to spatially bounded regions in the network, the Regions of Interest (ROI), that provide an event-based prediction and classification of the load case situation using supervised Machine Learning. The entire learning problem depending on the data from all sensor nodes is divided into multiple local ROI segment learners capable of predicting the local load situation based on the sensor data from the ROI, finally using a majority decision from a set of activated learners.

 ISBN 978-3-527-33606-7

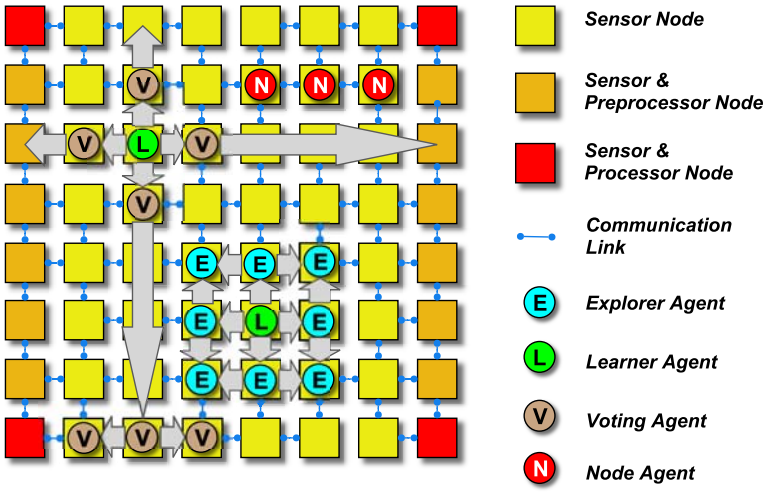



**Fig. 6.25** *The logical view of a sensor network with a two-dimensional mesh-grid topology (left) and examples of the population with different mobile and immobile agents (right): node, learner, explorer, and voting agents. The sensor network can contain missing or broken links between neighbour nodes. Non-mobile node agents are present on each node. Pure sensor nodes (yellow nodes in the inner square) create learner agents performing regional learning and classification. Each sensor node has a set of sensors attached to the node, e.g., two orthogonal placed strain gauge sensors measuring the strain of a mechanical structure..*

Again, mobile agents are used to collect (percept) and deliver sensor data to learner agents, limited to the ROI, shown in Fig. **6.25**, by using a divide-and-conquer approach. The advanced ε-Interval and NN Learning algorithm from Sec. **5.4.5** is used on-line with ROI data sets to compute a decision tree used for load case prediction and recognition.

Distributed and event-based machine learning is evaluated with simulated load data from **[BOS16A]**. Different load situations were applied to a metal plate, and the strain of the plate was computed at particular points using FEM simulation, finally mapped on artificial sensor data processed by a MAS in the *SEJAM* MAS simulator. The structure of the sensor network was already shown in Fig. **6.25**. A load-strain matrix $\boldsymbol{T}$ is required to compute artificial strain values.

Originally, the simulated loads used to compute the load-strain matrix $\boldsymbol{T}$ were cylindrical weights placed at $N$=400 weight positions from an equidistant rectangular grid for $N_x = N_y = 20$. The force on the upper surface of the upper horizontal plate due to the loading hence vanishes outside the circle covered by the weight; inside this circle the force points in direction $-\boldsymbol{z}$ and equals 1 N/cm$^2$. Since the used deformation model is linear, the actual value assigned to this force is unimportant if it re-scale the load-strain matrix such that the magnitude of reconstructed loads matches those of the true load for noise-free data. After

 ISBN 978-3-527-33606-7

computing the deformation field, the surface strain in *x* and *y* direction at the sensor points was extracted by computing the deformation field and the extracted surface strain for a sequence of refined meshes.

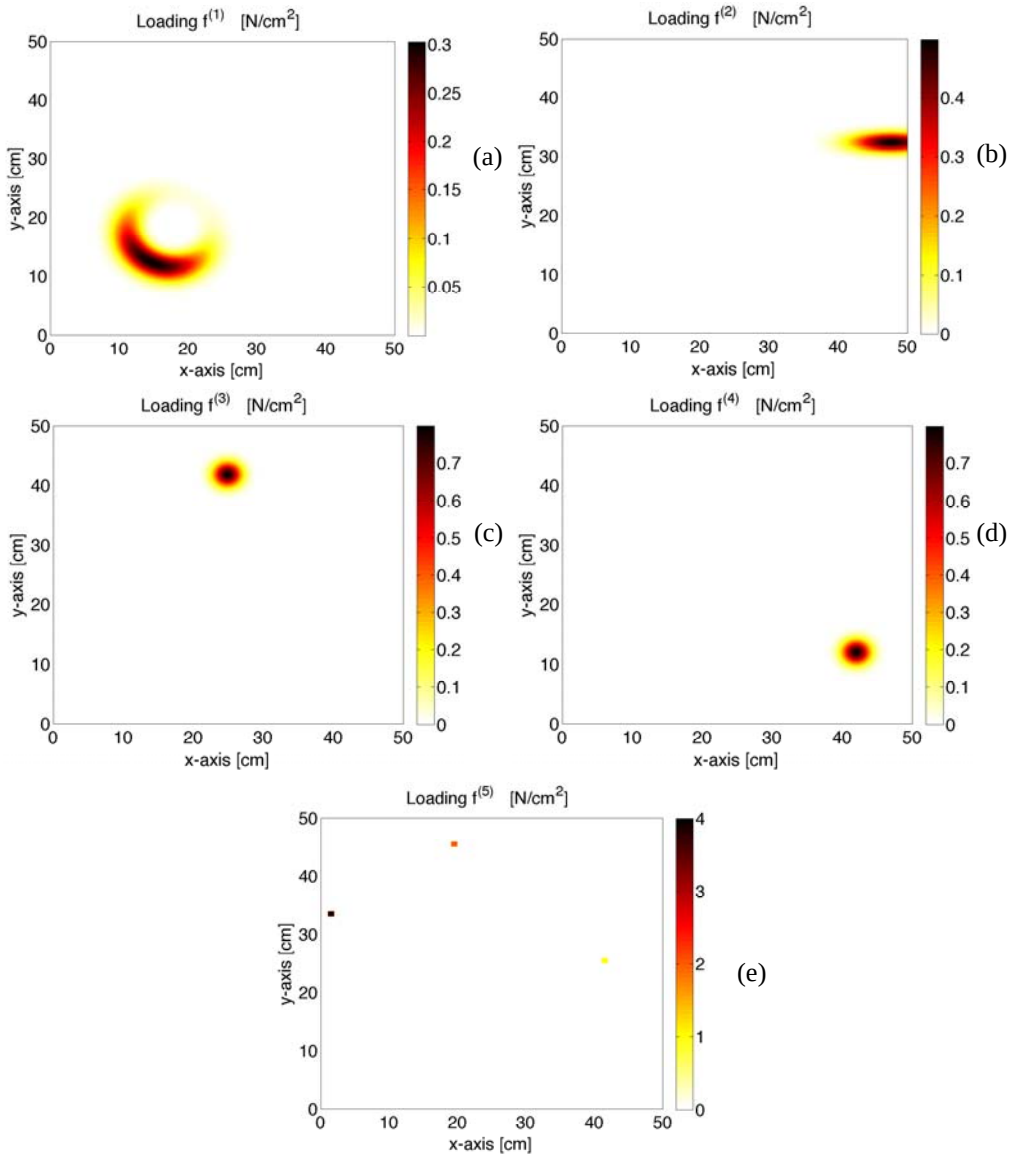



**Fig. 6.26** *From (a) to (e): The five artificial loads $1^{(1)}, .. , 1^{(5)}$. (f)* ***[BOS16A]****.*

 ISBN 978-3-527-33606-7

Apart from the simple cylindrical loads to compute the load-strain matrix $\boldsymbol{T}$, five spatially extended and different loads $\boldsymbol{l}^{(1)}, .. , \boldsymbol{l}^{(5)}$ with different characteristics and acting on different parts of the steel plate were simulated, shown in Fig. **6.26**.

The simulated strain values are then converted to integer values in the range between 1 and 1024, with a no-load signal corresponds to the value 512. If $\sigma_i$ denotes a simulated strain value, this conversion is done using the formula $s_i = \lfloor 512 + 10000*\sigma_i \rfloor$, $1 \leq i \leq 2M$. ($\lfloor a \rfloor$ denotes the largest integer smaller than or equal to $a \in \mathbb{R}$.).

Thus, five strain measurement vectors $\boldsymbol{s}^{(1)}, \ldots , \boldsymbol{s}^{(5)}$ are computed., and these five data sets were feed consecutively as sensor values into the simulation framework for the sensor network shown in Fig. **6.25**. After a randomly chosen load situation was applied, the response of the LM is evaluated. Between two load cases there is always the null-load case that is applied to relax the system. For the learning an ε=5 setting was used. Monte-carlo simulation of sensor noise was applied with ε=5, too, adding equally distributed noise intervals [-ε,ε] to the sensor values.

Simulation results are shown in Fig. **6.27**. The top figure shows the temporal agent population for a long-time run with a large set of single training and classification runs, with a zoom shown in the middle two figures. Each peak represents a particular training or classification run. In this experiment, the learner agents are non-mobile, and hence the population do not change in time. Side and edge nodes are not populated with learner agents. A learner agent covers the ROI containing its host node and eight surrounding nodes. If the host node detects a sensor event (change), it notifies the waiting learner agents by storing a `TODO` tuple in the database, consumed by the learner agent. Either a training (learning) or classification (prediction) request is send. In both operational cases the learner agent will send out explorer agents to collect sensor data in the neighbourhood, which is back delivered to the learner. In the classification modus the learner will use the already trained and learned model to predict the load case situation. The result is send out in the network by using voting agents, finally accumulated by the four edge nodes of the network by the major vote election.

The bottom figure shows global classification results obtained by major voting of all event-activated regional learner agents. The load sequence was randomly chosen, but always with a idle load situation ($l^0$) inserted between two different load cases. The learner must predict this load after a change in the load situation, too, meaning there is no load applied to the structure. The major election results show a very accurate prediction of significant distinguishable loads, i.e., $l^1$, $l^2$, $l^2$, and $l^4$. The last load case $l^5$ is hard to distinguish from the zero load case.

 ISBN 978-3-527-33606-7

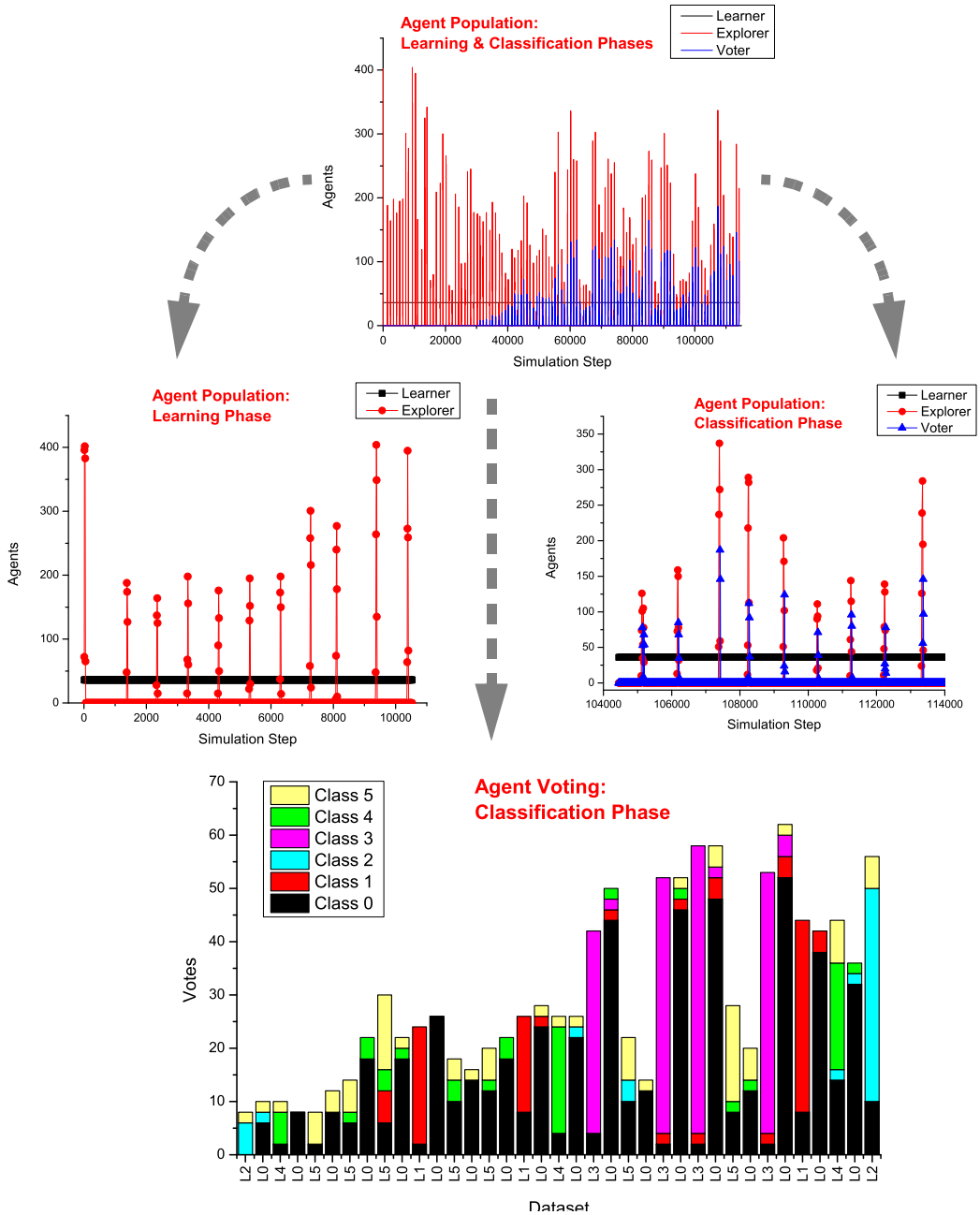



**Fig. 6.27** *Simulation Results. The top figure shows the event-based system behaviour by the temporal agent population for a long-time run with a large set of single training and classification runs, with a zoom shown in the middle two figures. The bottom figure shows global classification results obtained by major voting of all event-activated regional learner agents.*

 ISBN 978-3-527-33606-7

## 6.5 Bibliography

[AKY02] Akyildiz, I. F., Su, W. S. W., Sankarasubramaniam, Y., & Cayirci, E. (2002). A survey on sensor networks. IEEE Communications Magazine, 40(8). doi:10.1109/MCOM.2002.102442

[BAD11] S. Badri, "JUNCTION BASED ROUTING : A NOVEL TECHNIQUE FOR LARGE Shabnam Badri THESIS WORK 2011 ELECTRONICS JUNCTION BASED ROUTING : A NOVEL TECHNIQUE FOR LARGE," 2011

[BAG10] Baglini, E., Cannata, G., & Mastrogiovanni, F. (2010). *Design of an embedded networking infrastructure for whole-body tactile sensing in humanoid robots*. 2010 10th IEEE-RAS International Conference on Humanoid Robots, 671–676. doi:10.1109/ICHR.2010.5686834

[BLU16] Bluetooth Special Interest Group, http://www.bluetooth.com, accessed: 2016-06-27

[BOR06] R. H. Bordini and J. F. Hübner, *BDI agent programming in AgentSpeak using Jason*, Computational Logic in Multi-Agent Systems, Volume 3900 of the series Lecture Notes in Computer Science, Springer, 2006, pp. 143-164.

[BOS11A] S. Bosse, H*ardware-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

[BOS12A] S. Bosse, F. Pantke, *Distributed computing and reliable communication in sensor networks using multi-agent systems*, Prod. Eng. Res. Devel., 2012, DOI 10.1007/s11740-012-0420-8

[BOS12B] S. Bosse, F. Pantke, and F. Kirchner, *Distributed Computing in Sensor Networks Using Multi-Agent Systems and Code Morphing*, ICAISC Conference, Poland, Zakapone, 2012

[BOS13A] S. Bosse, *Intelligent Microchip Networks: An Agent-on-Chip Synthesis Framework for the Design of Smart and Robust Sensor Networks*, Proceedings of the SPIE 2013, Microtechnologie Conference, Session EMT 102 VLSI Circuits and Systems, 24-26 April 2013, Alpexpo/Grenoble, France, SPIE, 2013, DOI:10.1117/12.2017224012

[BOS14A] S. Bosse, Design of Material-integrated Distributed Data Processing Platforms with Mobile Multi-Agent Systems in Heterogeneous Networks (Conference), Proc. of the 6\u2019th International Conference on Agents and Artificial Intelligence ICAART 2014, 2014, DOI:10.5220/0004817500690080.

[BOS14B] S. Bosse, *Distributed Agent-based Computing in Material-Embedded Sensor Network Systems with the Agent-on-Chip Architecture*, IEEE Sensors Journal, Special Issue MIS, 2014, DOI:10.1109/JSEN.2014.2301938.

[BOS15A] 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, DOI:10.3390/s150204513.

[BOS15B] S. Bosse, 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, DOI: 10.1016/j.protcy.2014.09.039

[BOS15C] S. Bosse, *Unified Distributed Computing and Co-ordination in Pervasive/Ubiquitous Networks with Mobile Multi-Agent Systems using a Modular and Portable Agent Code Processing Platform*, in The 6th International Conference on Emerging

Ubiquitous Systems and Pervasive Networks (EUSPN 2015), Procedia Computer Science, 2015

[BOS16A] S. Bosse, A. Lechleiter, *A hybrid approach for Structural Monitoring with self-organizing multi-agent systems and inverse numerical methods in material-embedded sensor networks*, Mechatronics, 2015, doi:10.1016/j.mechatronics.2015.08.005, in press

[BRA02] Braginsky, D., & Estrin, D. (2002). Rumor routing algorthim for sensor networks. Proc. of the Workshop on Wireless sensor networks and applications, 22–31

[BUS04] S. Bussmann, N, R. Jennings, M. Wooldridge, *Multiagent Systems for Manufacturing Control*, Springer, 2004

[CAR00] 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

[CAS10] Castanedo, F., García, J., Patricio, M. a., & Molina, J. M. (2010). *A Multi-agent Architecture Based on the BDI Model for Data Fusion in Visual Sensor Networks*. Journal of Intelligent & Robotic Systems, 62(3-4), 299–328. doi:10.1007/s10846-010-9448-1

[CHU02] 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

[EBR11] M. Ebrahimi, M. Daneshtalab, P. Liljeberg, J. Plosila, H. Tenhunen, Agent-based on-chip network using efficient selection method, 2011 IEEEIFIP 19th International Conference on VLSI and SystemonChip (pp. 284-289). IEEE. doi:10.1109/VLSISoC.2011.6081593

[FAR09] Muddassar Farooq, Bee-inspried protocol enginnering, Springer, 2009

[GHE10] Ghezzo, F., Starr, A. F., & Smith, D. R. (2010). *Integration of Networks of Sensors and Electronics for Structural Health Monitoring of Composite Materials*. Advances in Civil Engineering, 2010, 1–13. doi:10.1155/2010/598458

[GUE06] R. Guerraoui, L. Rodigues, *Introduction to Reliabe Distributed Programming*, Springer, 2006

[GUI08] M. Guijarro, R. Fuentes-fernández, and G. Pajares, *A Multi-Agent System Architecture for Sensor Networks*, Multi-Agent Systems - Modeling, Control, Programming, Simulations and Applications, 2008.

[GRA14] M. J. McGrath, C. N. Scanaill, *Sensor Technologies*, Apress Open, 2014, ISBN 9781430160134

[HAA06] Haas, Z., Halpern, J., & Li;, L. (2006). Gossip-based ad hoc routing. Networking, IEEE/ACM Transactions on, 14(3), 479–491. doi:10.1109/TNET.2006.876186

[HER07] Carla Hertleer, Anneleen Tronquo, Hendrik Rogier, Luigi Vallozzi, Lieva Van Langenhove, *Aperture-coupled patch antenna for integration into wearable textile systems*, IEEE antennas and wireless propagation letters, (6) 2007, IEEE, pp 392-395

[HU13] F. Hu and Q. Hao (Eds.), *Intelligent Sensor Networks*. CRC Press, 2013, ISBN 9781420062212.

[IEE16] IEEE 802.11p, https://standards.ieee.org, accessed: 2016-05-28

[ITT16] IT+Textiles, http://dru.tii.se/reform/projects/itextile/, accessed: 2016-06-27

[INT00] Intanagonwiwat, C.; Govindan, R.; Estrin, D.: Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks. In: Proceedings of the 6th

annual international conference on Mobile computing and networking. 11(2000) 3, S. 56-67

[JAC05] Margot Jacobs, Linda Worbin, *Reach: dynamic textile patterns for communication and social expression*, CHI'05 Extended Abstracts on Human Factors in Computing Systems, 2005, ACM, pp 1493-1496

[JUN12] R. Junges, F. Klügel, *How to design agent-based simulation models using agent learning*, Proc. of the Simulation Conference (WSC) 2012.

[KEN09] Timothy Kennedy, Patrick Fink, Andrew Chu, Nathan Champagne, Gregory Lin, Michael Khayat, *Body-worn E-textile antennas: the good, the low-mass, and the conformal*, IEEE Transactions on Antennas and Propagation, (57) 4, 2009, pp. 910-918

[KON00] Kone, M. T., Shimazu, A., & Nakajima, T. (2000). The State of the Art in Agent Communication Languages. Knowledge and Information Systems, 2(3), 259–284. doi:10.1007/PL00013712

[KOR07] I. Koren, C. M. Krishna, *Fault tolerant Systems*, Morgen Kaufmann, 2007

[LEI15] P. Leitão and S. Karnouskos (ed.), in Industrial Agents Emerging Applications of Software Agents in Industry. Elsevier, 2015

[LI11] C. Li, H. Zhang, B. Hao, and J. Li, "A survey on routing protocols for large-scale wireless sensor networks.," Sensors (Basel, Switzerland), vol. 11, no. 4, pp. 3498–526, Jan. 2011

[LOE05] Andreas Löfgren, Lucas Lodesten, Stefan Sjöholm, *An analysis of FPGA-based UDP/IP stack parallelism for embedded Ethernet connectivity*, IEEE, NORCHIP Conference, 2005. 23rd

[LUE97] M. Lückenhaus and W. Eckstein, *A Multi-Agent Based System for Parallel Image Processing*, Proceedings of the International Conference on Parallel and Distributed Methods for Image Processing at SPIE's Annual Meeting, Proc. SPIE 3166, 1997

[LYM08]
A .Lymberis, R. Paradiso, *Smart fabrics and interactive textile enabling wearable personal applications: R&D state of the art and future challenges*, 30th Annual International Conference of the IEEE Engineering in Medicine and Biology Society, 2008, pp. 5270--5273

[MAG14] M. Magno, S. Marinkovic, B. Srbinovski, E.M.Popovici, *Wake-up radio receiver based power minimization techniques for wireless sensor networks: A review*, Microelectronics Journal, (45) 12, 2014, pp. 1627-1633

[MAL07] Malik, H., Shakshuki, E., Dewolf, T., & Denko, M. K. (2007). *Multi-Agent System for Directed Diffusion in Wireless Sensor Networks*. 21st International Conference on Advanced Information Networking and Applications Workshops (AINAW'07), 2. doi:10.1109/AINAW.2007.260

[MAR05] Marík, V., McFarlane, D.C., 2005. Industrial adoption of agent-based technologies. IEEE Intell. Syst. 20 (1), 27–35

[MAR11] Peter Marwedel, Embedded System Design, Springer, 2011, ISBN 978-94-007-0256-1

[MCC95] F. G. McCabe, K. L. Clark, APRIL - Agent Process Interaction Language, 1995, (M. Wooldridge & N. R. Jennings, Eds.) Intelligent Agents Theories Architectures and Languages LNAI volume 890. Springer-Verlag

[MEN05] Y. Meng, *An Agent-based Reconfigurable System-on-Chip Architecture for Real-time Systems*, in Proceeding ICESS '05 Proceedings of the Second International

Conference on Embedded Software and Systems, 2005, pp. 166-173.

[MOR12] G. D. F. Morales, “Big Data and the Web: Algorithms for Data Intensive Scalable Computing,” IMT Institute for Advanced Studies, Lucca,, 2012.

[OPA16] OPEN Alliance SIG website, http://www.opensig.org, Accessed: 2016-06-25

[PEC08] Pechoucek, M., Marík, V., 2008. Industrial deployment of multi-agent technologies: review and selected case studies. Auton. Agent. Multi-Agent Syst. 17 (3), 397–431.

[QIN10] Qin, Z., Xing, J., & Zhang, J. (2010). *A Replication-Based Distribution Approach for Tuple Space-Based Collaboration of Heterogeneous Agents*. Research Journal of Information Technology. doi:10.3923/rjit.2010.201.214

[SAN08] Sansores, C., & Pavón, J. (2008). *An Adaptive Agent Model for Self-Organizing MAS* (Short Paper). Proc. of 7th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2008), Padgham, Parkes, Müller and Parsons (eds.), May, 12-16., 2008, Estoril, Portugal (pp. 1639–1642).

[SHA07] Shakshuki, E., Malik, H., & Xing, X. (2007). Agent-Based Routing for Wireless Sensor Network. Lecture Notes in Computer Science,Advanced Intelligent Computing Theories and Applications. With Aspects of Theoretical and Methodological Issues, 4681, 68–79. doi:10.1007/978-3-540-74171-8_8

[TIE16] T. Tiedemann, C. Backe, T. Vögele, P. Conradi, *An Automotive Distributed Mobile Sensor Data Collection with Machine Learning Based Data Fusion and Analysis on a Central Backend System*, Proceedings of the 3rd Int. Conf. on System-integrated Intelligence SysInt, 2016, Procedia Technology, Elsevier

[TOR11] Patrick Van Torre, Luigi Vallozzi, Carla Hertleer, Hendrik Rogier, Marc Moeneclaey, Jo Verhaevert, Indoor off-body wireless MIMO communication with dual polarized textile antennas, IEEE Transactions on Antennas and Propagation, (59) 2, 2011, pp 631-542

[WAR01] B. Warneke, M. Last, and B. Liebowitz, “Smart dust: Communicating with a cubic-millimeter computer,” Computer, 2001

[WOO99] M. Wooldrige, *Intelligent Agents, in Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*, G. Weiss (Ed), MIT Press, 1999

[WU99] J. Wu, *Distributed System Design*, CRC Press, 1999

[YAN02] Yang, Y., Zincir-Heywood, A. N., Heywood, M. I., & Srinivas, S. (2002). Agent-based routing algorithms on a LAN. IEEE CCECE2002. Canadian Conference on Electrical and Computer Engineering. Conference Proceedings (Cat. No.02CH37373), 3, 1442–1447

[YUA06] Yuan, S., Lai, X., Zhao, X., Xu, X., & Zhang, L. (2006). *Distributed structural health monitoring system based on smart wireless sensor and multi-agent technology*. Smart Materials and Structures, 15(1), 1–8. doi:10.1088/0964-1726/15/1/029

[ZHA07] Zhao, S., Yu, F., & Zhao, B. (2007). An Energy Efficient Directed Diffusion Routing Protocol. 2007 International Conference on Computational Intelligence and Security (CIS 2007). doi:10.1109/CIS.2007.70

[ZAH12B] S. Zhang, Z. He, and H. Yang, “Mobile Agent Routing Algorithm in Wireless Sensor,” in D. Jin and S. Lin (Eds.): Advances in CSIE, Vol. 2, AISC 169, vol. 2, 2012, pp. 105–113

[ZIG16] ZigBee Alliance, http://www.zigbee.org, accessed: 2016-06-27

ISBN 978-3-527-33606-7