SpyGlass: A Wireless Sensor Network Visualizer

Carsten Buschmann
Institute for Telematics,
University of Lübeck
D-23538 Lübeck, Germany,
+49 451 500 5384
buschmann@itm.uni-luebeck.de

Dennis Pfisterer
Institute for Telematics,
University of Lübeck
D-23538 Lübeck, Germany,
+49 451 500 5384
pfisterer@itm.uni-luebeck.de

Stefan Fischer
Institute for Telematics,
University of Lübeck
D-23538 Lübeck, Germany,
+49 451 500 5380
fischer@itm.uni-luebeck.de

Sándor P. Fekete
Department of Mathematical Optimization
Technical University at Brunswick
D-38106 Braunschweig, Germany
+49 531 391-7551
s.fekete@tu-bs.de

Alexander Kröller
Department of Mathematical Optimization
Technical University at Brunswick
D-38106 Braunschweig, Germany
+49 531 391-7410
a.kroeller@tu-bs.de

ABSTRACT
In this paper we present a modular and extensible visualization framework for wireless sensor networks. These networks have typically no means of visualizing their internal state, sensor readings or computational results. Visualization is therefore a key issue to develop and operate these networks. Data emitted by individual sensor nodes is collected by gateway software running on a machine in the sensor network. It is then passed on via TCP/IP to the visualization software on a potentially remote machine. Visualization plug-ins can register to different data types, and visualize the information using a flexible multi-layer mechanism that renders the information on a canvas. Developers can easily adapt existing or develop new custom tailored plug-ins for their specific visualization needs and applications.

Categories and Subject Descriptors
C.2.3 [**Network Operations**]: *Network monitoring.*

General Terms
Measurement, Performance, Experimentation.

Keywords
Sensor networks, smart dust, visualization, debugging, embedded computing.

1. INTRODUCTION
Recently, the study of wireless sensor networks [4] has become a rapidly developing research area that offers fascinating perspectives for combining technical progress with new applications of distributed computing. These networks consist of tiny battery powered devices that can be seen as a combination of a micro computer and a sensor board. They feature a number of sensors (maybe custom tailored for certain applications) as well as wireless communication and limited computational capabilities. Limitations arise from power, form factor and price constraints. Examples are the Mica Motes developed at the UC Berkeley [6] or the Embedded Sensor Board ESB 430/2 from the FU Berlin [1]. As can be seen in Figure 1, these devices are usually only a few centimetres big and don't have any user interfaces like displays or keyboards.

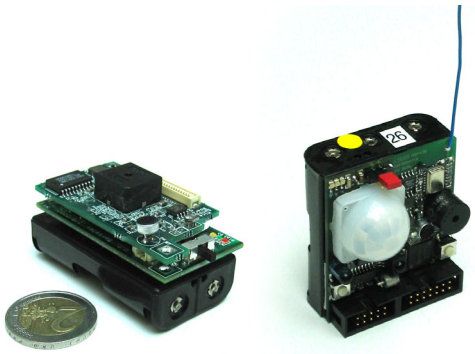

Figure 1: The Mica2 (left) and ESB 430/2 (right) wireless sensor motes

Deployed in large numbers by for example dropping the wireless sensor motes from a plane flying over the area of interest, sample applications comprise the monitoring of physical structure integrity, biological habitats, battle fields or environmental parameters. They can help for example in disaster recovery, by helping rescue personnel finding victims, guiding them through the terrain or giving them a more comprehensive view on the situation as a whole. In these application domains, future scenarios anticipate networks consisting of several thousand nodes [7], [8]. Nevertheless, the development of algorithms, protocols and applications is extremely difficult and progresses only slowly. The inherent

lack of user interfaces leads to a tedious and error-prone developing and debugging cycle for which embedded systems are ill-reputed. In addition, unpredictable hardware and communication behaviors aggravate application development. This might be one of the reasons why real-world systems today consist of rather a few dozens nodes [1], [10], [11]. While there is tool-chain support for embedded systems, development software for sensor networks is still in the early beginnings.

2. RELATED WORK

With the SpyGlass sensor network visualizer we aim at easing the life for sensor network debugging, evaluation and deeper understanding of the software by visualizing the sensor network, its topology, the state and the sensed data. A few tools exist which cover some aspects of sensor network data display and visualization.

The *Surge Network Viewer* [13] and the *Mote-VIEW Monitoring Software* [12] are Crossbow's products to visualize wireless sensor networks. The *Surge Network Viewer* features topology and network statistics visualization as well as logging of sensor readings and the viewing of the logged data. The statistics function includes the end-to-end data packet yield, a prediction for the future and the RF link quality, but is limited to these features. The system is not extensible; hence custom-made visualizations are not feasible. The *Mote-VIEW Monitoring Software* covers essentially the same topics but presents a much cleaner user interface and more features. It is also capable of logging wireless sensor data to a database and to analyze and plot sensor readings. It allows querying the sensor network for collected data in a database-like manner, hiding the distribution of the data collection software on the sensor nodes. Apart from lacking independency from the Mica sensor network hardware [6] and easy extensibility, this tool differs from our software in that it targets the operation and data-retrieval phase while we aim at easing the setup and debugging of applications.

The *TinyViz* visualization framework is the most generic among the related visualizers. It is part of the TOSSIM package of TinyOS [5]. It visualizes Sensor readings, LED states, radio links and allows direct interaction with running TOSSIM simulations. The architecture of *TinyViz* allows adding application specific visualization functionality. This functionality includes specialized drawing operations, subscription and reaction to events and providing feedback to the TOSSIM simulator. It is very tightly coupled to the TinyOS software, the TOSSIM simulator and the Mica sensor network hardware [6].

3. SPYGLASS ARCHITECTURE

The visualization framework consists of three major functional entities: The sensor network, the gateway nodes located in the sensor network and the visualization software. Figure 2 shows how these three work together with a TCP/IP based transit network.

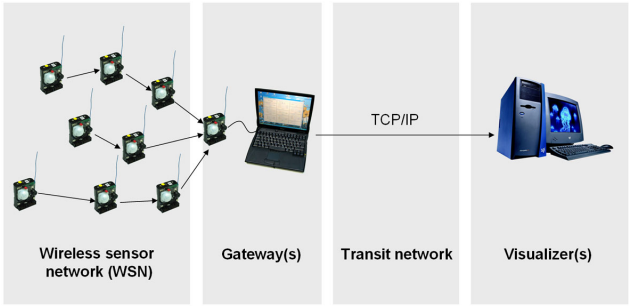

Figure 2: Information from the sensor network is forwarded to the gateway and then transferred to the visualizer

3.1 Data flow

In the sensor network, each individual sensor node collects data using its sensors, derives new information from calculations or communicates with neighbors. If new information is generated, it is forwarded to one or more so called gateway nodes. Our current implementation of the sensor network software uses simple flooding mechanisms to achieve maximum reliability by redundant transmission. However, any technique may be used to route the data to the gateway nodes. This may range from the rather simple current approach up to a fully sophisticated geographic routing protocol in order to minimize the generated in-network traffic.

Gateway nodes connect the sensor network to some other TCP/IP based network. Depending on the setup they may be implemented in very different manifestations. In our demo installation gateway nodes are regular sensor nodes that are connected to an off-the-shelf PC via a serial connection. In real world deployments it may not be feasible to use standard PC equipment, but specialized hardware or mobile phones to provide a connection to another network.

On boot, each sensor node checks its serial port for a connection to a PC, and hence can determine whether it functions as a gateway node or not. If so, it queries the PC for the current time and date, and forwards this information into the wireless sensor networks. Like this, all nodes in the networks become loosely time-synchronized, so all readings can be augmented with time stamps within a common, consistent time system. This enables consistent visualization of sensor readings etc. regardless from which particular sensor node the data originates. The gateway node forwards all the data received from other nodes to the gateway PC via the serial connection.

The software component running on a gateway PC features a ring buffer for storing a number of data packets from the sensor network. Its size can be set using a startup parameter. A gateway PC listens on a certain port (also set using a parameter) for incoming connections from visualization PCs. On connection, it sends over the contents of it ring buffer. When new data arrives via the serial link, it is both stored into

the buffer and forwarded to all connected visualizers. The circular buffer enables the system to bridge the time gap of transit network failures or to provide data to visualization stations which connect at a later point in time.

For the network connection between the gateway PC and the machine running the visualization component, all kinds of TCP/IP based networks including LAN, WLAN or GPRS can be used. Obviously the gateway and the visualization component can also run on the same machine to enable in-situ monitoring of a sensor network. All that needs to be done is that the visualization software component opens a connection to the local host.

All data packets flowing through the sensor network, from the gateway node to the attached PC, and through the transit networks to the visualization PC have the same payload format. Independent of the information they carry (e.g. sensor readings, calculated data, internal sensor state, etc.), they consist of a data type indicator, length information and the data. Using this simple format, developers can come up with new data types which will be immediately supported by the sensor network and the gateway software. Up to this point data has only been forwarded, all data processing and display tasks are performed by the visualization software. Using this architecture and data format makes it possible to replace each of the three components individually, since the communication between them follows a well-defined packet format. Currently we provide three sensor network configurations: a real life sensor network using the embedded sensor board ESB 430/2 [1] and data originating from the ns-2 [2] network simulator and a replay implementation for debugging purposes.

3.2 The visualization component

The graphical user interface of the visualization component (see Figure 3) consists of three major components: a graphical display canvas (on the upper left), a sidebar for tree-structured textual information on the network as a whole (on the upper right), and a display for line-based output e.g. for debugging purposes (at the bottom).

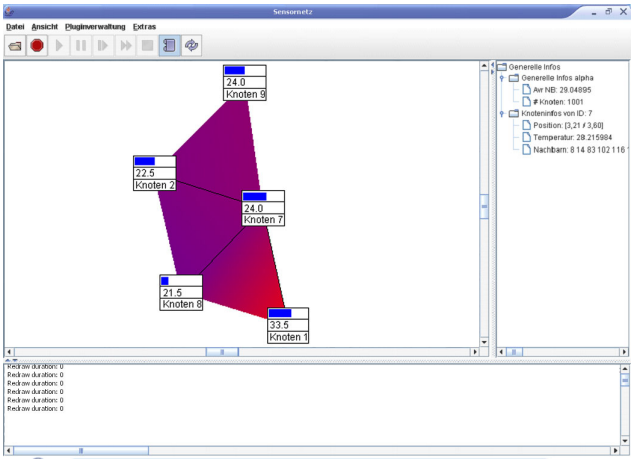

Figure 3: The graphical user interface of the visualization component

The graphical display canvas consists of three layers:

- The *background layer* is used for painting the background of the visualization. In Figure 3 it is used for displaying both the white background and the reddish/bluish temperature gradient field.
- The *relation layer* is used for displaying all kinds of relations between nodes e.g. by connecting nodes that can communicate with a "can communicate" relation as in Figure 3.
- The *node layer* is used for displaying the actual nodes. In Figure 3, the box representing each node, as well as the textual information and the blue battery indicator are painted on this layer.

The actual visualization is done by user-written plug-ins, one for each visualization demand. When a number of visualization plug-ins independent of each other shall cooperate, one important issue is to make sure that painting operations don't collide. To solve this problem while keeping configuration simple, we have decided to create different plug-in types, each type corresponding to one of the three display layers and being only allowed to paint on "their" layer (see also Figure 4).

- *Background Painter plug-ins* draw the background of the visualization canvas. They can also be used to illustrate spatial phenomena which can be inferred from the received sensor data and positions. Examples are temperature maps [3], the display of coordinate systems or terrain visualizations.
- *Node Relation Painter plug-ins* display arbitrary relations between sensor nodes onto the canvas, e.g. by using lines to connect related nodes or drawing polygons around them. Such relations might be communication links, group membership or routing paths.
- *Node Painter plug-ins* draw the actual nodes and additional information onto the canvas. The depiction may be dependent on the node type (e.g. gateway node, cluster head, etc.), and may comprise a symbol representing the node, as well as additional textual or graphical information arranged around it.

Apart from these plug-ins painting on the canvas, there are two other types of plug-ins.

- *Node Positioner plug-ins* are used by *Node Painter* and *Node Relation Painter plug-ins* to determine where to paint the nodes and relation end points on the canvas. Placement decisions can be either based on location estimates/measurements received from the sensor network or on strategies based on graph theoretical calculations optimizing screen representation. Like this, the actual depiction of

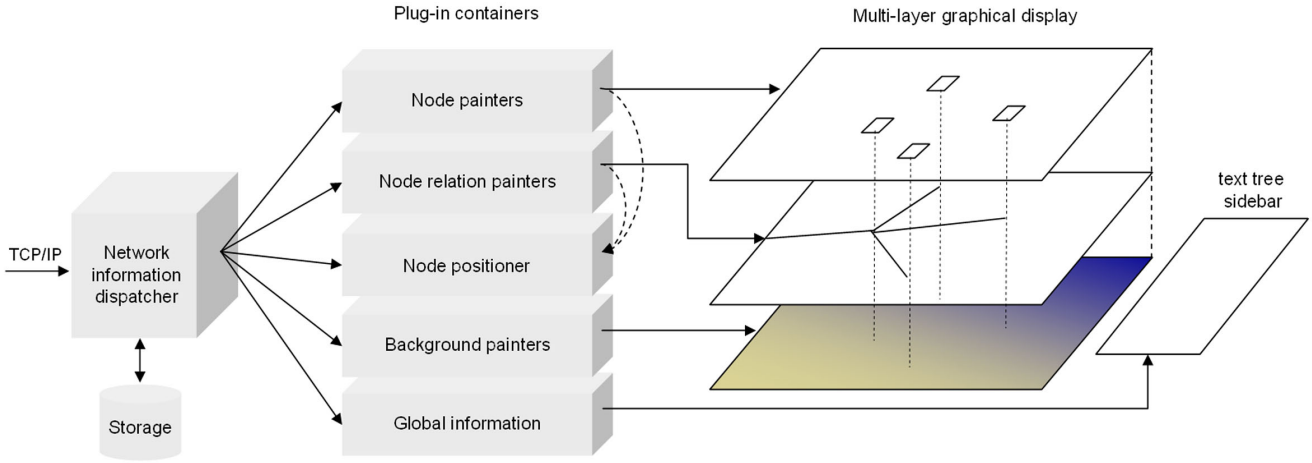

Figure 4: The architecture of the visualization component

nodes is decoupled from the positioning of the node representation, so both positioner plug-ins and node/relation painters can be replaces easily.

- *Global information plug-ins* display information about partitions or the whole network in a textual way. This information is displayed in a sidebar and can be structured in a tree. Examples are the overall number of nodes, average neighborhood degree, etc.

3.3 Flexibility as a key property

SpyGlass features a very flexible drawing and plug-in architecture. Most of its inner components can be exchanged or extended easily. This extensibility has its roots in the way how plug-ins and drawing instructions are implemented.

The plug-ins are not directly drawing on the canvas, but instead they use a set of drawing primitives available in SpyGlass. Calling these primitives results in the assignment of graphical objects (such as lines, rectangles, text etc.) with the layers. These graphical objects are then used by the canvas to represent itself in the graphical user interface, using appropriate drawing routines. Using this architecture has several advantages: Drawing on different layers avoids conflicts between plug-ins that have different priorities and the painting code in the plug-ins is independent from the actual canvas implementation.

The three canvas drawing plug-in types (*Node Painter*, *Relation* and *Background plug-ins*) have their own layer on the canvas on which their plug-ins exclusively draw. The user can change the order and visibility of the plug-ins within each plug-in container to achieve an optimal presentation of the data.

Because the canvas brings drawing routines that implement the drawing primitives, it is easy to add new canvas types, and SpyGlass is able to draw on a variety of different canvas types. One actual canvas implementation, the Java2D–Canvas, can be seen on Figure 3. Other canvas types may be implemented for special purposes. Implementing a new type of canvas is as easy as providing a mapping from the abstract drawing instructions to the concrete target canvas.

Like this, it is possible to easily create new canvasses like a Postscript-canvas to document the sensor network in Postscript graphics, a canvas that could create a series of JPEG images or an MPEG video for demonstration purposes. Note that it is possible to operate multiple canvases in parallel, so that e.g. videos could be created while watching the visualization on the regular Java2D–Canvas on the screen.

3.4 Recording and Playback

SpyGlass can not only be used for visualizing wireless sensor networks that are currently in operation. It is also able to record activities going on at a certain point of time, and playing it back later, for seeing it again or watching it at a different speed, similar to slow motion or fast forward. Whenever a sensor network is visualized, the user can select to additionally record all the information arriving at the visualization component. By choosing a filename and pressing the red dot button (see Figure 3), the recording is started. Note that not the current visualization is recorded, but all data packets arriving at the network information dispatcher (see Figure 4). This allows not only playing back the current visualization at a later point in time, but having completely new views on the current network situation by employing different plug-ins for interpreting and visualizing the recorded information. This even allows for developing special visualization plug-ins to show interesting details that were not recognizable during the actual visualization run. Apart from different playback speeds, Spyglass certainly implements features commonly known from video players like fast forward and jumping to certain point in playback.

3.5 Plug-ins

Currently, the SpyGlass visualization component has already a number of plug-ins available. They range through all five categories.

The *temperature map plug-in* belongs to the category of the background painters. It allows the visualization component to indicate the temperature distribution by coloring the background between the sensor nodes. Each sensor periodically measures the current temperature. On change, it broadcasts a packet containing its address, the current time and the corresponding temperature. The plug-in residing in the visualization component refers to the currently used node positioning plug-in to assign the temperature value to a position and maps the temperature to a color. Colors between the node positions are interpolated from the values belonging to the surrounding nodes. The resulting temperature maps can be seen in the screenshot in Figure 3.

Currently a very simple *node painter* is implemented. It displays a box for each node, with the node's address written in it (also see Figure 3).

The *temperature plug-in* indicates the temperature a nodes measures as a numerical value. It uses the same messages as the temperature map plug-in. It is at work in Figure 3.

The *battery plug-in* uses a blue bar to indicate how much energy the nodes have left. To do so, the nodes periodically measure their battery voltage, and forward packets augmented with their address and a timestamp to the gateway. Voltage indication can be observed in Figure 3.

The *topology plug-in* is a node relation painter. The sensor nodes periodically send out beacon packets that neighbors use to maintain a neighbor list. The list is broadcasted periodically and forwarded to the gateway node. In the visualization component, the plug-in processes the packet, and draws lines from the sender to all its neighbors. Again, the node positioner plug-in is consulted for the nodes' positions. Connectivity is indicated in Figure 3.

There also is a plug-in that can be used to *position* the nodes on the graphical display canvas. It assumes that the sensor nodes are aware of their real positions. Hence they periodically send out their coordinates together with their address and a timestamp. The plug-in subscribes to these packets and maps the coordinates to positions on the graphical display canvas. It automatically extends the coordinate boundaries the canvas represents whenever out-of-scope positions come to its knowledge.

The *spring embedder* is another node positioner plug-in. Opposite to the aforementioned one it assumes that the sensor nodes do not have any information about where they are. Hence it has to figure out appropriate positions for node display by other means: It subscribes to the neighborhood messages, and keeps track of the topology. Using a spring embedder, it then positions nodes that can hear each other close on the canvas, whereas it places nodes without connectivity far away from each other. The resulting node distribution can be seen in the screenshot.

The *average neighborhood size plug-in* is a global information plug-in. It subscribes to the neighborhood messages used by the topology plug-in, but keeps track only of the number of neighbors. Like this, it can display the average network-wide connectivity in the global information sidebar.

There is another general information plug-in that keeps track of the *number of nodes* in the network. Subscribing to the neighborhood list packets (or any other periodical packet type), it keeps track of the nodes in the network. When a node didn't send a packet for a certain time, it is considered not to be part of the network anymore.

3.6 Adding custom functionality

Adding new elements to the visualization is straightforward, two cases can be distinguished:

- If the new visualization element is driven by messages already emitted by the sensor network, all that needs to be done is implementing an additional plug-in. To implement for example a general information plug-in that could count the number of network partitions, the messages containing the neighborhood lists could be used. All that is needed is a plug-in that keeps track of the all the connectivity information, constructs graphs from it and displays their number in the sidebar.
- If new messages are needed, things get only slightly more difficult. To implement for example the visualization of motion detection sensor readings, a developer simply needs to define a new data type, construct it as a binary array in the sensor and use the common forwarding service to a gateway node. To visualize the data, a simple *Node Painter plug-in* must be implemented which registers itself as a handler for the new data type, parses it and attaches this information to the corresponding node.

4. STATUS AND FUTURE WORK

SpyGlass is currently used throughout several projects in our group for research as well as teaching purposes. It is implemented using the Java 2 Standard Edition and the Java2D framework. We are currently in the process of finishing support for all features and eliminating remaining bugs. The sensor network implementation is completed for the Embedded Sensor Board ESB 430/2 from the FU Berlin [1] in C using the provided firmware from the FU Berlin. As soon as this process in completed, we plan to publish it under an open source license (GPL) soon.

We tested the scalability properties for the visualization component using data generated by the Ns-2 network simulator [2]. We found that it scales to thousands of nodes without problems. In addition, there is experience in visualizing real-life sensor networks of about 10 devices. To be able to visualize bigger networks, we plan to enhance to data forwarding mechanisms used in the sensor network.

In addition to that, future work will include a TinyOS compatible implementation to support the Berkeley Motes hardware platform [6]. Additionally we want to integrate 3D visualization support using the Java3D framework and additional plug-ins. Current ideas include a map component to display geospatial information, more sophisticated node

drawing plug-ins and support for the network simulator Shawn [14].

5. ACKNOWLEDGMENTS

This work is part of the SWARMS and SwarmNet projects funded by the German Research Foundation (DFG) in the programs SPP 1140 (Basic Software for Self-Organizing Infrastructures in Networked Mobile Systems) and SPP 1126 (Algorithms for large and complex networks). For further information, see http://www.swarms.de and http://www.swarmnet.de.

6. REFERENCES

[1] Website of the Embedded Sensor Board ESB 430/2: http://www.scatterweb.com.

[2] The Network Simulator - ns-2: http://www.isi.edu/nsnam/ns.

[3] C. Buschmann. D. Pfisterer and S. Fischer: Experimenting with Computer Swarms: a Mobile Platform based on Blimps, Poster, The Second International Conference on Mobile Systems, Applications, and Services, June 2004.

[4] I.F. Akyildiz, S. Su, Y. Sankarasubramanian and E. Cayirci: Wireless Sensor Networks; A Survey, Computer Networks, Vol. 38, No. 4, March 2002, pp. 393 – 422.

[5] TinyOS: http://www.tinyos.net.

[6] Crossbow Technology Inc., "Mica2Mote," http://www.xbow.com.

[7] D. Estrin, R. Govindan, and J. Heidemann, "Embedding the internet: introduction," *Commun. ACM*, vol. 43, no. 5, pp. 38–41, 2000.

[8] V. Kumar, "Sensor: the atomic computing particle," *SIGMOD Rec.*, vol. 32, no. 4, pp. 16–21, 2003.

[9] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson, "Wireless sensor networks for habitat monitoring," in *ACM International Workshop on Wireless Sensor Networks and Applications (WSNA'02)*, Atlanta, GA, Sept. 2002. [Online]. Available: citeseer.ist.psu.edu/mainwaring02wireless.html

[10] R. Szewczyk, A. Mainwaring, J. Polastre, J. Anderson, and D. Culler, "An analysis of a large scale habitat monitoring application," in *Proceedings of the 2nd international conference on Embedded networked sensor systems*. ACM Press, 2004, pp. 214–226.

[11] P. Zhang, C. M. Sadler, S. A. Lyon, and M. Martonosi, "Hardware design experiences in zebranet," in *Proceedings of the 2nd international conference on Embedded networked sensor systems*. ACM Press, 2004, pp. 227–238.

[12] Mote-VIEW Monitoring Software, Crossbow Technology Inc.: http://www.xbow.com/Products/productsdetails.aspx?sid=88.

[13] Surge Network Viewer, Crossbow Technology Inc.: http://www.xbow.com/Products/productsdetails.aspx?sid=86.

[14] Shawn, an open-source discrete event simulator for sensor networks: http://www.swarmnet.de/shawn