Distributed Modular Toolbox for Multi-modal Context Recognition

David Bannach[1], Kai Kunze[1], Paul Lukowicz[1,2], and Oliver Amft[2]

[1] Institute for Computer Systems and Networks, UMIT, Hall in Tyrol, Austria
[2] Wearable Computing Lab, ETH Zurich, Switzerland

Abstract. We present a GUI-based *C++* toolbox that allows for building distributed, multi-modal context recognition systems by plugging together reusable, parameterizable components. The goals of the toolbox are to simplify the steps from prototypes to online implementations on low-power mobile devices, facilitate portability between platforms and foster easy adaptation and extensibility. The main features of the toolbox we focus on here are a set of parameterizable algorithms including different filters, feature computations and classifiers, a runtime environment that supports complex synchronous and asynchronous data flows, encapsulation of hardware-specific aspects including sensors and data types (e.g., int vs. float), and the ability to outsource parts of the computation to remote devices. In addition, components are provided for group-wise, event-based sensor synchronization and data labeling. We describe the architecture of the toolbox and illustrate its functionality on two case studies that are part of the downloadable distribution.

1 Introduction

As context awareness gains popularity and moves towards applications, tools for the efficient implementation of context recognition systems become even more important. Such tools need to address a broad range of issues from sensor management middleware through low-level signal processing and pattern recognition to high-level context modeling and utilization. We focus on the signal processing and recognition part. Motivated by the needs of two large industrial projects sponsored by the European Union (WearIT@Work [1] and MyHeart [2]) we have developed the *Context Recognition Network (CRN) Toolbox* for development, prototyping, and implementation of multi-modal, distributed context recognition systems. The emphasis of the CRN Toolbox is on three issues:

1. simplifying the step from prototyping (often done with tools such as MATLAB) to real life implementation,
2. easy portability between different devices and sensor systems with a particular focus on low-power mobile devices, and
3. facilitating the reuse of components and easy extensibility/adaptation of existing recognition systems.

W. Grass et al. (Eds.): ARCS 2006, LNCS 3894, pp. 99–113, 2006.

Problem Description. The design of the CRN Toolbox is based on a system model that has been studied by our groups on a theoretical level in [3]. We assume a set of sensors distributed on the user's body and in the environment, each with its own data rate and format. A typical context recognition application consists of a series of filters, feature extractions, and classifications successively applied to the sensor data. In general, the processing follows a feed-forward hierarchical data-flow model with initial computations being applied to the data stream of each single sensor. Then, the individual data streams are successively fused in joint features and possible partial classifications until, at the final classifier stage, a decision is made based on all or most of the gathered data.

From previous experience with context recognition and the requirements of the abovementioned EU projects we have found the following issues to reoccur in most implementations:

1. Most applications rely on components from a relatively limited set of filters, features, and classifiers. The differences between the applications are (1) the specific combination of such components, (2) the data-flow path, and (3) component-specific parameters such as sliding window sizes, filter frequencies, and – last but not least – classifier training.
2. In general, system development begins with data collection followed by offline experiments with rapid development tools like MATLAB or WEKA [4]. In advance of any online experiments, the algorithms have to be hard- or recoded for the specific platform. If problems occur or some sort of adaptation is required, the whole cycle restarts from the beginning because experiments tend to be difficult to conduct with optimized production code.
3. Porting complex context recognition tasks to mobile platforms, such as PDAs or phones, often requires parts of the implementation to be converted from floating point to integer because of hardware restrictions. In addition, it might be desirable to outsource the more computation-intensive higher-level algorithms to a remote server.
4. The synchronization of data from different sensors can be a major problem. This involves the merging of data streams with different sampling rates, finding a common start point for all sensors, and compensating for clock drifts and other sources of jitter to retain synchronization over longer periods of time.

Paper Contributions. Based on the above considerations, the central idea behind the CRN Toolbox is to provide a development environment offering (1) a set of parameterizable filter, feature, and classifier components, (2) a runtime system that controls the required data flow and handles synchronization, (3) parameterizable sensor interfaces, and (4) an easy-to-use GUI. With this system, a specific recognition application can be constructed by selecting the appropriate components from the GUI, specifying the component parameters and classifier training data, and connecting the components according to the required data paths. Extension and adaptation of the application are just a matter of adding/exchanging components in the GUI. Since sensor details are encapsulated in the interfaces, sensor changes are also easy to incorporate (as long as

the classifiers are not affected). Different parts of the application can be made to run on different systems using special TCP/IP-based interface components, enabling the outsourcing of computationally intensive parts to external servers. Interfaces are also provided for tools like MATLAB and WEKA. By basing all computations on an abstract 'Value' data type equipped with arithmetic, any application may switch between floating point, fixed point, and integer without any recoding. Finally, special components are provided for group-wise sensor synchronization through events and for data labeling.

Related Work. Several research groups have already addressed the issue of online sensor data processing and have proposed modular extensible architectures. However, none of them cover all the problems specific to wearable and ubiquitous computing.

Sicheneder et al. [5] from the University of Passau presented a framework that facilitates the graphical specification and execution of complex signal processing applications with focus on industrial monitoring. In contrast to our toolbox, this framework does not address the specific requirements of wearable computing environments such as portability between different devices, outsourcing of computationally expensive tasks, or abstraction from actual data type. Furthermore, there is no explanation or validation of how distributed processing and synchronization of multiple sources work.

The Lancaster CommonSense ToolKit [6] is a collection of tools that assist in the communication, abstraction, visualization, and processing of sensor data. CSTK's core qualities are its real-time facilities and embedded systems-friendly implementation. However, it does not support a flexible composition of the processing entities, synchronization, embedding of tools like MATLAB, or distributed execution which are all key features of the CRN Toolbox and are needed for most real-world applications.

IU SENSE [7] is a Java-based approach to a toolkit that allows for real-time processing, visualization, and analysis of data generated by multiple sensors. Despite its modular and extensible design it is not suited to run on wearable devices, mainly because of performance issues.

OSIRIS-SE [8], developed at Umit, is the stream-enabled version of the hyperdatabase infrastructure for process management that was initially developed at ETH Zurich. It is focused on reliable data-stream processing in distributed environments where mobile devices interoperate with stationary computers. It utilizes Peer-to-Peer techniques and is implemented in Java. Due to the overhead coming from the high-level approach, it can only process simple algorithms on mobile devices and is not able to adapt optimally to different hardware.

Triana [9] is a GUI-based data analysis tool developed at Cardiff University. It is written in Java and provides a large library of analysis algorithms mainly targeted for particle physics, but also useful for other signal processing applications. Triana is focused on distributed computing using Grid and Peer-to-Peer techniques. It lacks most of the features specific for the wearable/ubiquitous environment described above.

Another well-known tool is the Context Toolkit [10] from Georgia Tech. Unlike our CRN Toolbox, it focuses on the application level. Thus, we view it as complementary rather than a competition to our system.

The area of sensor networks and associated middleware contains a lot of work that could be useful together with or as an extension to our system. The EU-funded RUNES project [11] is targeted towards flexible distribution of data processing tasks in heterogeneous sensor networks. The TinyDB [12] lets the sensor network appear as a database which can be queried with an SQL-like syntax. SensorWare [13] uses mobile agents that can replicate themselves throughout the network to gain information. With DSWare [14], applications can subscribe for events that occur on certain groups of sensor nodes.

2 Toolbox Concept and Implementation

The aim of the CRN Toolbox is to allow distributed multi-sensor context recognition to be implemented by simply plugging together standard, parameterizable components. Thus, with the CRN Toolbox, the implementation of a multi-modal context recognition system distributed over several platforms consists of:

1. compiling the toolbox for all platforms that it needs to run on,
2. using the GUI to select and configure the algorithms and data flow that the toolbox needs to execute on each platform, and
3. starting the toolbox on each platform with the configuration files created by the GUI.

Custom code and extensions are easily added to the toolbox by means of new classes compiled and linked in with the rest as desired. A detailed description of the toolbox implementation is beyond the scope of this text. Instead, we focus on the concepts behind the main features elaborated on in the previous section: component reusability and parameterization, flexible data flow, handling of synchronization events, encapsulation of hardware-specific aspects including sensors and data types (e.g., int vs. float), the ability to run and communicate across devices, and the configuration GUI. A detailed documentation of the implementation is contained within the source code.

Parameterizable, Reusable Components. The basic building blocks of the CRN Toolbox are *StreamTasks*, or *tasks* for short. Each algorithm (filter, classifier, etc.) available in the toolbox is implemented as such a task. The abstract `StreamTask` class shown in Figure 1 is based on POSIX threads. It serves as the superclass for all other tasks. Therefore, each task is a separate thread executing concurrently. A task has $0 \ldots n$ `InPorts` and $0 \ldots m$ `OutPorts`. It continuously processes the data received at its in-ports and puts the results on its out-ports.

Each task has a number of startup arguments that correspond to the parameters of the respective algorithm. The KNN classifier task for instance, requires KNNs "k", the filename of the training data, and an optional step size as its parameters.

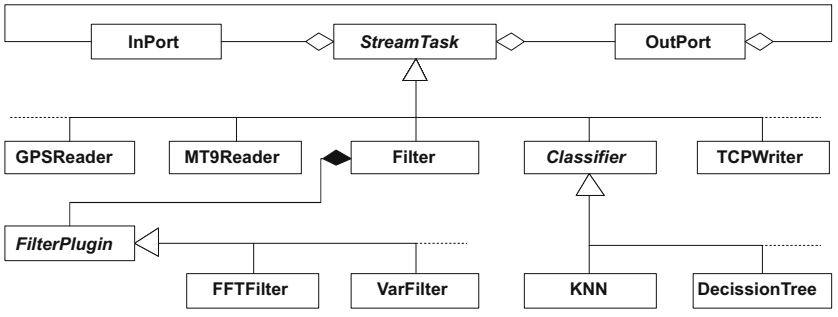

Fig. 1. Static structure of components

```
configuration = taskConf connectionConf [timeoutConf].
taskConf = taskKeyword "=" taskList.
connectionConf = connKeyword "=" connList.
timeoutConf = timeoutKeyword "=" number.
taskList = "[" [taskDef {"," taskDef}] "]".
taskDef = taskName "(" [param {"," param}] ")".
connList = "[" [connDef {"," connDef}] "]".
connDef = "Connection(" number "," number "," number "," number ")".
taskKeyword = "t" ["a" "s" ["k" "s"]]].
connKeyword = "c" ["o" "n" ["n" "e" "c" "t" ["i" "o" ["n"
    ["s"]]]]]]]]]].
timeoutKeyword = "s" ["e" "c" "o" "n" "d" ["s"]]]]]].
taskName = unquoted.
param = unquoted | quoted.
unquoted = letter | digit {letter | digit}.
quoted = '"' {character} '"'.
number = digit {digit}.
letter = "A" .. "Z" | "a" .. "z".
digit = "0" .. "9".
character = ( any ASCII character excluding " )
```

Listing 1.1. EBNF definition of the toolbox configuration language

The CRN Toolbox makes use of the Xparam library (http://xparam.sf.net/) for the de-serialization of objects. This allows the toolbox to be configured by a text file at runtime. See Listing 1.1 for the syntax definition of the configuration language in EBNF format. The 'tasks' section of the configuration file lists the class names and parameters of the `StreamTasks` that are instantiated and run in the toolbox. The connections between these tasks are defined in the 'connections' section.

Table 1 lists the algorithms currently existing in the toolbox. Customized algorithms can be added to the toolbox by creating a subclass of `StreamTask` and implementing the `run()` method as shown in Listing 1.2 for a sample task. The desired number of in- and out-ports must explicitly be allocated by the task constructor. If multiple in-ports are used, the task itself needs to take care in the `run()` method to avoid starvation problems. The `InPort` class provides both blocking and non-blocking access methods. For filter algorithms, there is a dedicated `Filter` class with a plug-in mechanism to support extended reusability of code. The `Filter` task handles packet I/O and calls the `filter()` method of the `FilterPlugIn` for each value. We recommend and prefer to implement filter

Table 1. List of existing algorithms in the CRN Toolbox

Reader Tasks:		
KeyboardReader	Keyboard reader task	available
MT9Reader	Reader for Xsens MT9-B sensors	available
XbusRawReader	Raw reader for Xsens Xbus	available
XSensLogFileReader	Reader for Xsens logfiles	available
NMEAReader	GPS reader task	available
ARSBReader	A reader for the ARSB	available
CricketReader	Cricket reader task	available
FileReader	Generic file reader	available
SerialReader	Generic serial port reader	available
BTnodeReader	BTnode reader	available
HexamiteReader	Hexamite reader	available
PhilipsReader	Reader for Philips protocol	available
RFIDReader	Reader task for ID-10 RFID reader	testing
Organizing Tasks:		
SelectiveSplitterTask	Splits a data stream into several streams	available
Synchronizer	Event based synchronizer	available
SyncMerger	Synchronizing merger task	available
SimpleMerger	Simple merger task	available
TransitionDetector	Transition detector	available
Filter Task and Plug-Ins:		
FilterTask	Configurable filter task	available
MaxFilter	Max filter plugin for FilterTask	available
MeanFilter	Mean filter plugin for FilterTask	available
MedianFilter	Median filter plugin for FilterTask	available
VarFilter	Variance filter plugin for FilterTask	available
SlopeFilter	Slope filter plugin for FilterTask	available
ScaleFilter	Scale filter plugin for FilterTask	available
ThresholdFilter	A two-thresholds filter	testing
FFTFilter	FFT filter plugin for FilterTask	testing
ASEFilter	Average signal energy filter	testing
BERFilter	Band energy ratio filter	testing
BWFilter	Bandwidth filter	testing
CGFilter	Center of gravity filter	testing
FlucFilter	Fluctuation filter (freq. and time domain)	testing
PeakFilter	Peak filter	testing
SFRFilter	Spectral rolloff frequency filter	testing
Classifier Tasks:		
ClassifierTask	Base class for classifier tasks	available
KNN	KNN classifier	available
RangeChecker	Very simple classifier	available
Hexamite2D	Very simple classifier	available
Distance2Position	Very simple position calculation	available
SimpleHexSensClassification	Simple Classifier using xsens and hexamite	available
SequenceDetector	Detects specified sequences	testing
Writer Tasks:		
TCPWriter	Write data to TCP port	available
TCPClientWriter	Write data to a server via TCP	available
SerialWriter	Multifunction serial writer	available
LoggerTask	Data logger task (FileWriter)	available
ConsoleWriter	Console logger	available
PhilipsWriter	Philips serial writer	available
Nothing	Data repeater (e.g. for debugging)	available
Nirvana	Quiet data sink	available

algorithms within such filter plug-ins. Finally, the use of the Xparam library makes it necessary to implement a copy-constructor for each task and to register all other constructors with special Xparam macros (not shown in sample code).

Parameterizable Engine with Data-Flow Control. The data streams between tasks are created by *directed connections* from out-ports to in-ports. The according section of the configuration file specifies the connections between tasks by indexing their corresponding out- and in-ports. A stream consists of a continuous sequence of `DataPacket`s. The `DataPacket`s, or *packets* for short, are the data entities that contain the sampled values belonging to a single time instant. Each packet bears its own time stamp and sequence number plus a vector of sampled values represented by the abstract data type `Value` (see Figure 2). The elements of this vector may be viewed as channels with equal sampling frequency. They are passed through the streaming network from task to task along the

```
class CustomizedTask : public StreamTask {
    public:
        CustomizedTask( Value *offset );
    private:
        Value *offsetVal;
        void run();
};

CustomizedTask::CustomizedTask( Value *offset ) {
    // initialize parameters
    offsetVal = offset;
    // create as many in- and out-ports as needed
    inPorts.push_back( new InPort() );
    outPorts.push_back( new OutPort() );
}

void CustomizedTask::run() {
    DataPacket *p = NULL;
    InPort *inPort = inPorts[0];
    OutPort *outPort = outPorts[0];

    while( running ) {
        p = inPort->receive();
        if( p ) {
            // get the value(s) from the data packet
            Value *val = p->dataVector.at(0);
            // process the value(s) here
            log( "processing the value:" ) << val;
            *val += offsetVal;
            // send the modified packet
            outPort->send( p );
            p = NULL;
        }
    }
}
```

Listing 1.2. Sample code for customized tasks

internal connections. Actually, only a pointer is passed around while the object data itself stays in place for better performance. When multiple receivers are connected to the same out-port of a task, the packet is cloned. If several streams need to be merged (e.g., to create a feature vector containing data from multiple sensors), a special `Merger` tasks must be used. As described below, `Merger` tasks may include synchronization of data streams with different sampling rates.

Synchronization. Ideally, sensors would have an exact clock to timestamp each data sample with the exact global time. Several methods for network time synchronization exist that are relying on smart sensors. In the real world, however, we have to cope also with simple sensor devices that send data samples with either internal sequence numbers or just a specified sampling rate. Therefore, when working with several sensors, their data streams must be synchronized to a common starting point. Such synchronization often needs to be repeated at runtime as the sampling rates are not reliably exact and might be jittered by communication delays. A well known method for this type of synchronization is the use of events that occur simultaneously at all involved sensors (e.g., jumping up to synchronize a set of acceleration sensors). Our system supports such synchronization through `Synchronizer` and `SyncMerger` tasks.

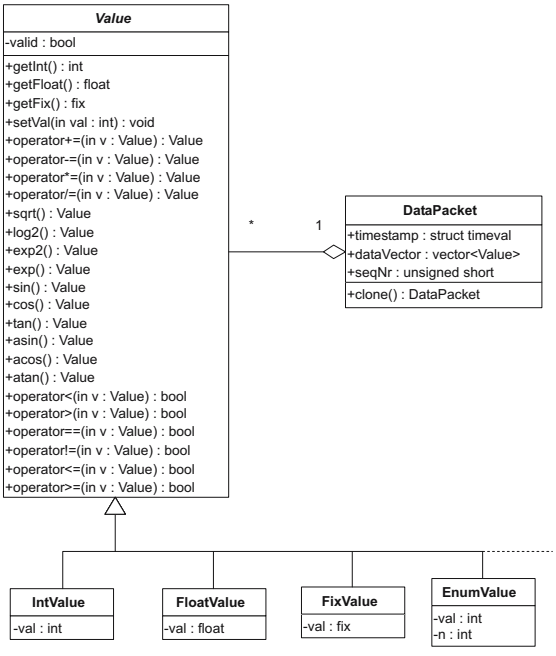

Fig. 2. Static structure of data packets

An example of such synchronization is shown in Figure 3. The `Synchronizer` searches for a distinct event[1] in the data received on the first in-port. The search is limited to a specified time-window which is triggered by a non-zero value on the second in-port. The time stamp t_e of the event is stored by the `Synchronizer`. The `Synchronizer` subtracts t_e from the time stamp of every data packet received later on. Hence, the time stamps of packets on the out-port will be relative to the event: $t_{out} = t_{in} - t_e$. Data streams synchronized this way can then be easily merged according to the time stamp to form one synchronized stream. The `SyncMerger` task merges two data streams that are synchronized to the same event. Packets on the second in-port are merged to matching packets from the first in-port. The matching criteria is the time stamp difference with a tolerance threshold. The data rate from the first in-port is maintained on the out-port, i.e., no packet from the first in-port is discarded. If the data packet P_a on the first in-port is older than the next available packet P_b at the second in-port (i.e., $t_a < (t_b - t_{tolerance})$), packet P_a is merged with a cached copy of the last packet from the second in-port to maintain the data rate. The copied values will be marked invalid. Otherwise, if P_b is older than P_a, P_b is discarded and merging continues with the next packet from the second in-port. The signals in Figure 3 stem from two MT9 acceleration sensors as configured in the GUI.

[1] We apply a variance filter with a sliding window of size 2 and take the maximal value as 'event'.

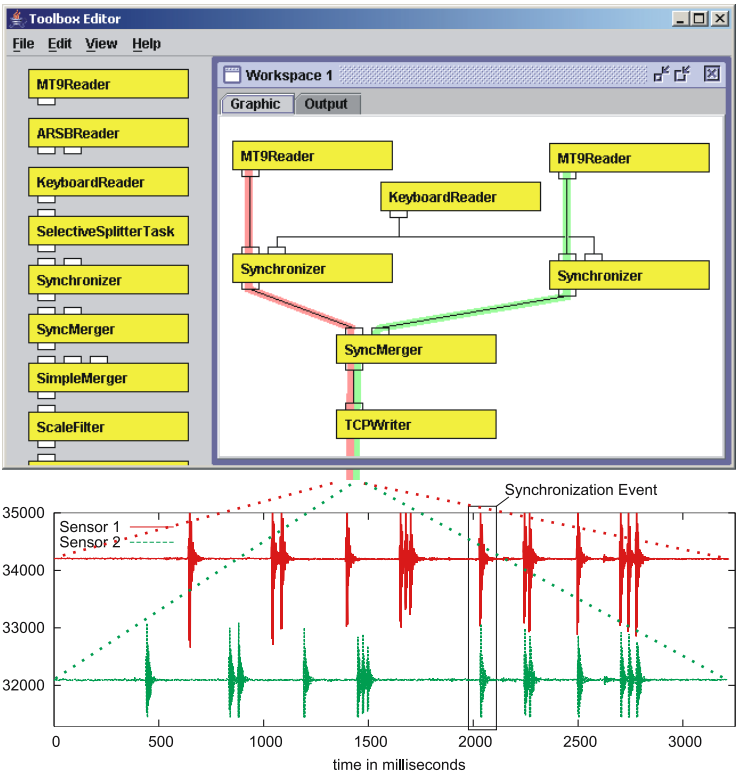

Fig. 3. Display of Sync event

Both sensors are moved at the same time but the signals are not synchronized at the beginning. After a short synchronization period, triggered by pressing a key, the toolbox is able to adjust the timestamps and merge the streams accordingly.

Sensor Hardware Encapsulation. Sensor interfaces are implemented as tasks with no in-ports and are called `Reader` tasks. They create a `DataPacket` for new sampling data as acquired from sensors (or other sources) and provide it on their out-ports. Our architecture supports multiple implementations of reader tasks that read from different sensors or even from other sources of information (e.g., web pages, other applications, files, etc.). We use a keyboard reader for online labeling of sensor data.

Data Type Encapsulation. The sampled values contained within data packets are all of the abstract data type `Value`. All mathematical operations and access methods are declared in the abstract `Value` class (see Figure 2). They are coded in the subclasses of `Value`. This allows algorithms to be implemented completely independent of the actual sampling data type when using the generic interface of the `Value` class. Such algorithms can process floating-point values on one machine and integer values on another without any recoding. The data

type of each stream (or even channel) can be configured to optimally match the needs of the application and the specific characteristics of the device the toolbox runs on. Implementations of the `Value` class exist for integer (`IntValue`), floating-point (`FloatValue`), and fixpoint (`FixValue`) values. Moreover, there is an enumeration value (`EnumValue`) for representing class labels and a raw value (`ByteBufValue`) for transportation of raw buffers. For performance reasons, the `Value` classes only provide mathematical operations that directly modify the object as, for instance, the operator '+=' does.

Distributed Execution and Tools Encapsulation. The key to distributed execution and the usage of external tools such as MATLAB are `Writer` tasks. They send the data received at their in-ports to external devices instead of an out-port. Such external devices can be files or displays but also network connections. For the latter, we currently use `TCPWriter` tasks that are based on TCP/IP sockets. Such tasks can send `DataPackets` in a serialized form to corresponding `TCPReader` tasks over the network. The serialization of data packets is done by an `Encoder` plug-in in the `TCPWriter`. Similarly, the `TCPReader` uses a `Decoder` plug-in for de-serialization. Thus, two toolboxes running on different machines can work as a single toolbox using a `TCPWriter` to transport `DataPackets`. In a similar way, the toolbox can communicate with any other program augmented by `TCPReader`/`Writer` compatible interfaces. Currently, such interfaces exist for MATLAB and WEKA.

GUI. We implemented a graphical editor (see Figure 3) for easy configuration of the toolbox. Tasks may be dragged from a library into the workspace where they are connected to other tasks with just a few mouse clicks. The editor is written in Java and automatically produces the configuration files for the toolbox according to the language definition shown in Listing 1.1.

3 Case Studies

Although still being work in progress, we already use the CRN Toolbox for in the WearIT@Work and MyHeart projects as well as in a variety of student works and demonstrators. The toolbox code including different sample applications can be downloaded from `http://csn.umit.at/download/toolbox/`. This section provides two case studies that show how to apply our toolbox to context recognition problems. The first is based on the demonstrators included in the software distribution. The second is a real-life example from the WearIT@Work project. The examples illustrate how distributed multi-modal context recognition systems can incrementally be constructed and adapted with the help of the CRN Toolbox.

3.1 Assembly Activity Recognition

We begin this case study with an explanation of how to use the toolbox to gather and save experimental data from a sensor. The real-time sensor data from an

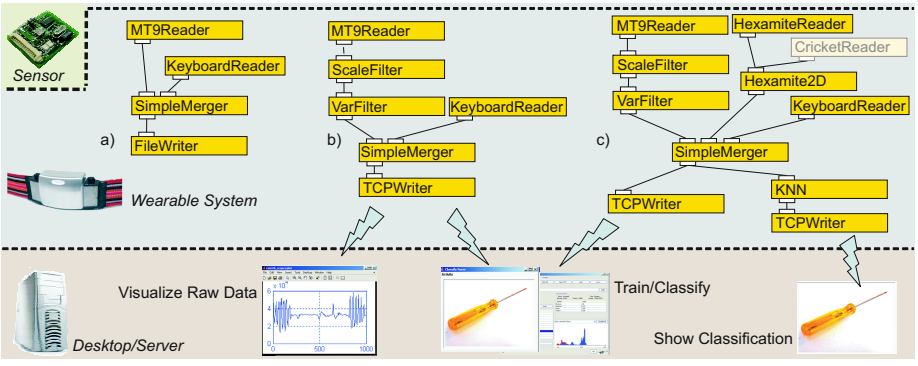

Fig. 4. Toolbox application schematics and MATLAB script reading sensor data

Xsens MT9 motion sensor is labeled by hand using a keyboard during the experimental trials. The sensor is mounted on the back of the user's hand. This setup resembles an initial stage in the development of an assembly activity recognition system for the WearIT@Work project. As shown in Figure 4a, the configuration consists of a `Reader` for the MT9 sensor, a `KeyboardReader` for labeling, a `SimpleMerger`, and a `FileWriter`, all executed on the wearable device. The `MT9Reader` acquires the data from the sensor while the experiment conductor can operate the keyboard and label the user's actions accordingly. The labels for the demonstrator included in the downloadable distribution mark the following activities: to hammer, to screw drive, to sandpaper, and to saw. The `SimpleMerger`, in turn, combines the sensor data with the labels and pipes them to the `FileWriter` which logs the labeled data to a file. In the same manner as the rest of the case study, this happens in perceived real-time.

In the next stage, the system is extended to include feature extraction using scaling (`ScaleFilter`) and a variance filter (`VarFilter`) on the signal processing side. Another component changed is the `TCPWriter` instead of the `FileWriter` to transmit the data to a remote server (see Figure 4b). The remote computer runs a MATLAB visualization application and/or our *SensServe* interface to the WEKA machine-learning software extended by a toolbox-compatible `TCPReader` module. Thus, the labeled data is forwarded to both applications. The MATLAB visualization application is able to display the sensor data in perceived real-time. This is a fast and easy way to ensure the correct operation of the sensors. It also proves to be a valuable help to get a first glimpse on characteristic features of the context recognition tasks. The *SensServe* interface can either be used to train a classifier or to do online tests and demonstrations. Naturally, as it interfaces to WEKA, it provides access to all classifiers and analysis algorithms implemented by this machine learning library. This eases the search for suitable features and classifiers dependent on the inference task.

Once the experimental stage of development is finished, the classification is moved from the server to a toolbox KNN classification component able to run on the mobile device. An already trained version of the toolbox KNN

classifier for the previously defined activities is included in the downloadable distribution. As the classifier runs on the mobile device, the `TCPWriter` can also be used to provide only the classification output to other external applications (e.g., a video capture application that may label the user video with his activities).

The third stage of the case study adds Hexamite ultrasonic sensors for hand tracking (see Figure 4c). To this end, a `HexamiteReader` and the module `Hexamite2D` for position computations are added while the classifier is trained to utilize position data. One ultrasonic listener is mounted on the user's same arm as the MT9 sensor. Disregarding height, two dimensions suffice for the position data because height is not crucial for the activities we defined above. Thus, the system is now able to differentiate between where in the room a specific activity is performed. This in turn can be used to determine regions of interest or to improve the activity recognition rate as certain activities happen at a specific places only. The analysis and training is done using WEKA in a setup similar to Figure 4b. The training can also be done on the mobile device. Finally, the trained classifier running on the mobile device transmits the resulting classifications using the `TCPWriter` to a desktop machine. The desktop simply visualizes the results.

To underline the flexibility of the toolbox, the Hexamite sensors may be replaced by Cricket ultrasonic sensors with hardly any effort. The operation of both sensors is very similar. We only had to write a `CricketReader` that outputs the data in the same format as the `HexamiteReader`, and insert it into the system using our GUI. This underlines that sensors with similar outputs can easily be interchanged by only using different readers in the toolbox and by adapting some filter parameters. The CRN Toolbox also supports readers with several output formats to enhance reuse. For example, the `MT9Reader` can either provide raw (int) or calibrated (float) data. Any application using the MT9 accelerometer data can easily be adjusted to use other accelerometers, simply by adding appropriate readers.

For systems with no sensors attached, the toolbox offers a `FileReader`. The `FileReader` reads previously recorded sensor data from a file and sends it to other components of the system in the same way as if the data originated from a real sensor. Thus, it is possible to re-run any experiment in real-time to fine-tune the toolbox components or debug a more complex application.

3.2 Gesture Recognition for Controlling a Document Browser

As mentioned before, the toolbox is currently used in the WearIT@Work project. One scenario in this project takes place during a doctor's ward round in a hospital. One problem of the ward round is the extremely limited time available per patient. Accessing each patient's documents on-site is important but operating a computer or PDA tends to be time-consuming and distracting. In the solution investigated in the WearIT@Work project, we apply context- and gesture recognition to automate and simplify the access to the patient's documents. The doctor is equipped with a QBIC [15] wearable computer, and an MT9 motion

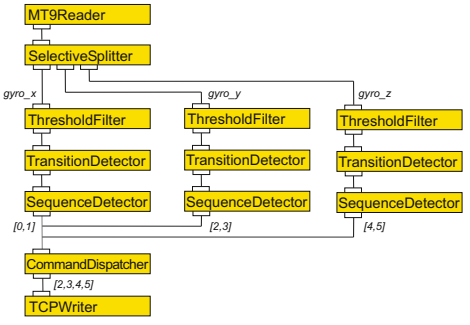

Fig. 5. Toolbox configuration for the gesture recognition

sensor plus an RFID reader at the right arm. The patient wears an RFID tag and the nurse carries a PDA. At each patient's bed, there is a bed-side monitor to display documents from the hospital database system. The QBIC connects wirelessly to both the PDA and the bed-side system. When the doctor approaches the patient's bed, the bed-side monitor automatically shows the specific patients list of documents. The doctor can then browse these documents by pointing at the monitor and swivel the forearm up and down or left and right. The following gestures are defined:

Forearm Gesture	*Command*
swivel up, then down	scroll up
swivel down, then up	scroll down
swivel left, then right	open document
swivel right, then left	close document
roll right, then left	activate gesture recognition
roll left, then right	deactivate gesture recognition

In the following, we briefly describe how the CRN Toolbox is extended with only three simple classes to deal with this gesture recognition and the controlling of the document browser. The configuration is shown in Figure 5. We use the 3–axis gyroscope of the MT9 motion sensor to detect the swiveling and rolling of the forearm. The x–axis of the gyroscope is aligned in parallel to the main axis of the forearm, and the y–axis in parallel to the plane of the hand. Rolling the forearm leads to either a positive or negative deviation of the angular velocity on the x–axis, depending on the roll direction. Similarly, swivel left/right is measured on the y–axis and swivel up/down on the z–axis.

We implemented a `ThresholdFilter` by extending the `FilterPlugin` class. The `ThresholdFilter` has two thresholds. All values greater than the upper threshold are set to 1, all value less than the lower threshold are set to 2, and all others are set to 0. With the appropriate thresholds, this filter applied to the x–axis gyroscope signal will output a sequence of values similar to

$$\ldots 00000011111111000222222222000000\ldots$$

when rolling the hand and forearm to the right and then back immediately. Then, we apply the existing `TransitionDetector` task and get the sequence below.

$$\ldots 01020 \ldots$$

The `TransitionDetector` allows for skipping sequences of equal values shorter than a specified length. Setting this parameter to 4, we get the following sequence instead.

$$\ldots 0120 \ldots$$

Now, we only need to identify exactly this simple sequence in the filtered data stream in order to tell that the *activate*-gesture has been executed. Similarly, all other gestures can be recognized. Therefore, we implemented the `SequenceDetector` task which accepts a list of value-sequences as its parameter. If one such sequence is detected in the data stream, the `SequenceDetector` sends the index of that sequence on the out-port. We apply these three algorithms on every axis of the gyroscope in parallel. We insert pseudo-sequences that never occur (e.g., [-1]) in the parameters passed to the `SequenceDetector` for the y- and z–axis to ensure that every gesture is assigned a unique index value.

The third class that had to be implemented for this scenario is the `CommandDispatcher` task. It simply forwards the gesture indices to its out-port if in *active* state and discards them otherwise. The state is set by the `activate`- and `deactivate` commands. This task is actually extended (but not shown here) with additional in- and out-ports to support RFID input and wireless connectivity to the nurse's PDA. The output of this task is sent to a `TCPClientWriter` that connects to the document browser of the hospital database system. The document browser can interprete the commands like real mouse and keyboard input.

4 Conclusion and Future Work

The CRN Toolbox is currently used in different projects. At the same time it is still evolving. In addition to the implementation of further components, the main directions are support for dynamic (re-) configuration of applications at runtime including ad-hoc cooperation between devices and better support for resource management. The later will include the back propagation of control messages through the processing network. Another immediate improvement of the CRN Toolbox that goes along with (re-) configuration is to provide a tighter coupling between runtime and GUI than just over configuration files. Furthermore, we envision extensions for ad-hoc cooperation of multiple toolboxes and a meta-model for sensors and context information.

References

1. WearIT@Work EU IST project http://www.wearitatwork.com/.
2. MyHeart EU IST project http://www.hitech-projects.com/euprojects/myheart/.

3. Anliker, U., Beutel, J., Dyer, M., Enzler, R., Lukowicz, P., Thiele, L., Troester, G.: A systematic approach to the design of distributed wearable systems. IEEE Transactions on Computers **53**(8) (2004) 1017–1033
4. Witten, I., Frank, E.: Data Mining: Practical machine learning tools and techniques. 2nd edn. Morgan Kaufmann, San Francisco (2005)
5. Sicheneder, A., Bender, A., Fuchs, E., Mandl, R., Sick, B.: A framework for the graphical specification and execution of complex signal processing applications. In: Proc. of ICASSP, Seattle, WA, USA (1998) 1757–1760
6. CSTK (CommonSense ToolKit) http://cstk.sourceforge.net/.
7. Caracas, A., Heinz, E., Robbel, P., Singh, A., Walter, F., Lukowicz, P.: Real-time sensor processing with graphical data display in java. In: Proc. of ISSPIT. (2003) 62–65
8. Brettlecker, G., Schuldt, H., Schek, H.: Towards reliable data stream processing with osiris-se. In: Proc. of BTW Conf., Karlsruhe, Germany (2005) 405–414
9. Taylor, I., Schutz, B.: Triana - A Quicklook Data Analysis System for Gravitational Wave Detectors. In: Second Workshop on Gravitational Wave Data Analysis, Editions Frontières (1998) 229–237
10. Dey, A., Salber, D., Abowd, G.: A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. Human-Computer Interaction (HCI) Journal **16**(2-4) (2001) 97–166
11. Runes project, EU IST http://www.ist-runes.org/.
12. Madden, S., Franklin, M., Hellerstein, J., Hong, W.: Tag: a tiny aggregation service for ad-hoc sensor networks. In: OSDI 2002, Boston, USA. (2002)
13. Boulis, A., Han, C.C., Srivastava, M.B.: Design and implementation of a framework for efficient and programmable sensor networks. In: The First International Conference on Mobile Systems, Applications, and Services (MobiSys 2003) San Francisco, CA. (2003)
14. Li, S., Lin, Y., Son, S., Stankovic, J., Wei, Y.: Event detection using data service middleware in distributed sensor networks. special issue on Wireless Sensor Networks of Telecommunications Systems, Kluwer **26:2-4** (2004) 351–368
15. QBIC - Belt Integrated Computer http://www.ife.ee.ethz.ch/qbic/index.htm.