UNIVERSITY OF OSLO
Department of informatics

A System for Storing and Reusing Data Sets from Sensor Networks

Master thesis

Eric L. L. C. Hansen

15. February 2012

Abstract

Advances in microelectronics and communication technology have provoked a significant increase in computing devices equipped with sensors and communication capabilities. This has led to an increasing interest in software applications that can detect events from sensor readings and react to these events, such as automated home care and smart environments. Developing such applications presents a number of difficulties, and a good source of sensor data is key to succesfully test and compare results.

In this thesis we analyze how sensor data can be reused in a general way and aim to design and implement a solution that can store and reuse sensor data for a variety of sensor types and applications. The results show that the implemented application provides a useful working tool to reuse and share sensor data sets.

Preface

I began working with this thesis in 2010 during the course of my master's degree. At the same time I worked a part-time job, gradually increasing work hours until I started working full time in the summer of 2011. Unfortunately I learned the hard way that combining a job in the IT sector with spare time thesis working can be a challenge, but at the same time working within the IT sector gave me an immediate recognition of the value of the subjects thaught at the University of Oslo's Department of Informatics, and the ability to use my work experience for my thesis.

I would like to sincerely thank Professor Dr. Thomas Plagemann for valuable insights and his patience and understanding during work with my thesis, the Distributed Multimedia Systems research group for interesting classes and projects, and the Departement of Informatics for providing me with the knowledge and insight that will allow me to be a productive and skilled worker in the IT industry, hopefully for many years to come!

Table of Contents

1. Introduction

1.1 Background and motivation

The current development of microprocessor technology is leading to ever smaller, faster and cheaper computing devices. Fields that have greatly gained from this development in recent years are those of sensor networks and software applications based on sensor input. Ever more areas can benefit from this development as the technology is used creatively to create smart systems that can base their control flow on different types of sensors in an environment. Diverse examples such as home care applications, environmental monitoring or even social networking can take advantage of the increasing availability of sensors, processors and networking technology allowing sensors to communicate their data over distance.

As more focus is aimed towards sensor-based event processing applications, the need for tools that aid in their development is increasing. While taking a course on "Advanced Topics In Distributed Systems" at the Department of informatics of the University of Oslo, I worked on the development of a system that used IP cameras to detect "intruders" at the coffee machine of the DMMS (Distributed MultiMedia Systems) research group. I soon discovered that developing a complex event processing system (CEP) based on sensor input could be tedious and time-consuming when the developers needed to generate the same sensor input each time a small code change was made to the system. In this case it meant me walking past 5 strategically placed cameras each time I wanted to test the system.

In another situation, the DMMS group wanted to assess the possibility of using sensor data generated at another research facility as input to a home care application being developed by a member of the group. It became evident that using this data would require a lot of work to be possible

During a discussion with my thesis supervisor Thomas Peter Plagemann, he suggested that analyzing, designing and implementing a system that could be used to store sensor data and reuse the stored data could be a topic for my thesis. This immidiately appealed to me as I had felt the need for such a system myself and I wanted to work on a real system related to the field of sensor networks and CEP for my thesis.

1.2 Problem definition

The goal of this thesis is therefore to analyze how data reuse can be achieved in a general manner for sensor-based CEP applications, and present a solution that answers this need.

We will need to answer:

- What are sensors and sensor networks? What are sensor-based CEP applications and why do we use them?

- Why can it help to reuse sensor data? What are the goals of the solution to be presented? What requirements wll we need to meet to present a general solution for data reuse?

- What will the solution design look like?

- How do we implement the solution design?

- How does the proposed solution perform and answer the defined requirements?

1.3 Main contributions

This thesis provides a proposal design on how data reuse can be achieved for sensor-based CEP applications, an overview of the requirements that must be met, and a working application to reuse sensor data that can be extended to suit the needs of diverse users.

1.4 Approach

Using litterature to find the state of the art in sensor-based CEP, I will analyze the requirements needed to meet the defined problem. Based on the requirements, I will design a solution to the problem at hand and implement the design by writing source code to create an application. Finally, the proposed application will be tested to determine how it meets or fails to meet the requirements and design.

2. Background

Advances in the fields of electronics, microprocessors and networking technologies have led to a growing number of devices that combine a processing unit with sensors and networking capabilities. Devices such as sensor nodes (also called motes), IP cameras and smart cell phones are examples of computing units that can send sensor data over a network. This has led to an increasing interest in creating software applications that can perform event detection and processing based on this sensor input, potentially from a broad range of heterogeneous sensor types, see figure 1.

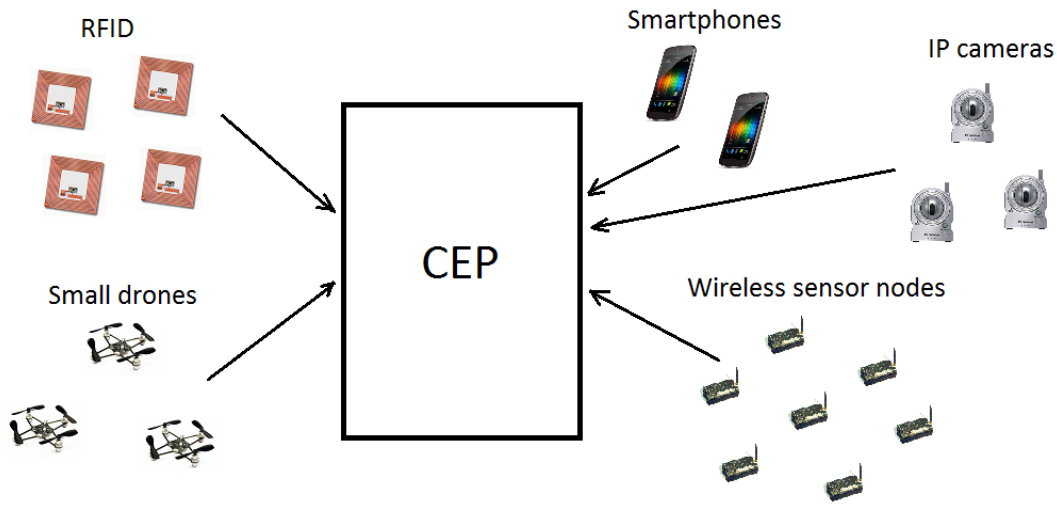


Figure 1: Example of potential sensor sources for a sensor-based CEP application. Includes images downloaded from [1] [2] [3] [4] [5]

Resulting applications range from wildfire detection and traffic monitoring to smart homes and home care systems. This chapter will describe the technologies involved in these applications, often labeled as Complex Event Processing (CEP) applications.

2.1 Sensors

A sensor is generally considered a device that can detect and measure a physical quantity and relay this measurement into a representation that conveys knowledge about an environment. The representation is usually an electric signal, such as a voltage, that can be converted to a numeric value. This allows sensor values to be read by computing devices that can then use the value in software. [6, p. 13]

Typically, sensors output number values that represent the value of a physical property in a unit of measure, e.g. a temperature sensor that outputs a value in the Celsius scale. Sensors can also output number values with predefined meanings such as true/false with 1/0 or any other mapping from a scale to known values.

In the context of computing, a broad range of input can be used to generate knowledge about an environment, and hence act as sensors. A camera may illustrate the idea. It will normally not be perceived as a sensor as it does not output a value in a unit of measure. But in a computer program, the camera images may be passed to a digital image processing algorithm that outputs some useful value, such as the number of faces in the image or address written on a letter.

We can use the term scalar to describe sensors that measure physical phenomena as simple quantities such as temperature or pressure, while multimedia sensors capture audio, image or video streams [7]. In the context of CEP, we may also encounter sensors that yield a state or condition. A motion sensor may output it's value as MOTION or NO_MOTION, or a camera-based sensor may send messages stating FACE_DETECTED if the necessary computation to detect such an event are performed on-board. For our purposes we will treat such sensors as scalar sensors.

2.2 Sensor nodes

Figure 2: The Tinynode sensor node. Image from [8]

Sensors are becoming cheaper, smaller and more widely available, and the same applies to microcontrollers and networking technology. This has made it possible to create devices that combine microcontrollers with one or more sensors that can be read and used by programs run on these devices. Adding radio chips or other networking capabilities to these devices yields small computers that can retrieve sensor data and send the data to other computing devices, see figure 2. The potential of this technology has caught the interest of researchers for many years and much effort has been put to develop suitable sensor nodes. Common to most of these devices is that they have limited resources while trying to perform resource-consuming tasks. [9] [10]

Sensor nodes generally consist of four main elements [11]:

- *Microcontroller Unit (MCU):*

Microcontrollers are small integrated circuits that contain a processor, memory, and programmable input/output units. They may also include components such as a clock

generator, analog to digital or digital to analog converters, timers or pulse-width modulation generators. They form the core of the sensor node and can be used to control the other components. Sensor nodes are programmed by transferring executable code, most often written and compiled on a desktop computer, to the internal memory of the microcontroller. There is an abundance of different microcontrollers providing variation in processor speed, memory size, available components and peripherals, and power usage. This allows for sensor nodes to be created for a wide variety of purposes.

- *Communication Unit:*

Sensor nodes are used to gather sensor data and they rely on a communication unit to transmit this data to other computing devices. Often they will use wireless communication as this presents greater flexibility for placement of the nodes, and a variety of technologies can be used to provide such capabilities. Some sensor nodes aim to communicate in sensor networks and use special gateways to interface with external networks, while other node types aim to provide integration with existing networks such as the Internet or cellular networks.

- *Sensor Units:*

The sensors are the components that retrieve data about the environment in which the sensor nodes are placed. Sensor nodes may include several sensors, and may also use their microcontroller unit to convert or translate sensor data to the application domain, e.g. translating data from an ultrasonic sensor to domain values of MOTION or NO_MOTION.

- *Power Unit:*

Sensor nodes consume power when using their components. In some cases the nodes may be connected to a continuous power source, but often they will rely on a battery and this poses serious limitations on how the components may be used, e.g. a sensor node may consume most power when using wireless radio and hence try to limit the use of this component as much as possible.

Sensor nodes may also include additional components such as [12]:

- *Location finding system:*

Sensor nodes may be deployed into a variety of environments and in some cases the exact location of the node may be unknown. If such a node senses interesting data, we may use an on-board location finding system such as a GPS module to detect the location if necessary.

- *Mobility system:*

It may be useful to equip sensor nodes with a mobility system to allow the deployers to change the location of the node in order to cover more suitable areas with the sensors. Thus a sensor node may be deployed on an RC vehicle with wheels, or even flying devices. Certain sensor devices may also have capabilities that can change the coverage area of its sensors without changing node location, such as pan-tilt-zoom controls on an IP camera.

- *Power generator:*

If a sensor node is to be deployed in remote locations for a long time it may be equipeed with power-generating components such as solar panels to prolong the lifetime of the node.

Parts of the research community have focused to a great extent on sensor nodes that are to be deployed in large networks of similar nodes, called wireless sensor network (WSN). These nodes focus mainly on the components discussed above, and the typical hardware architecture is depicted in figure 3.

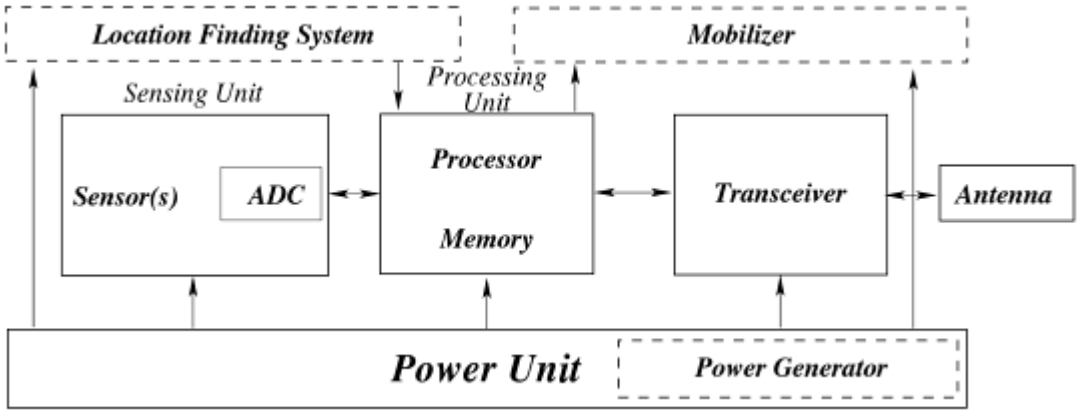


Figure 3: General architecture of sensor node hardware. Image copied from [12, p. 38]

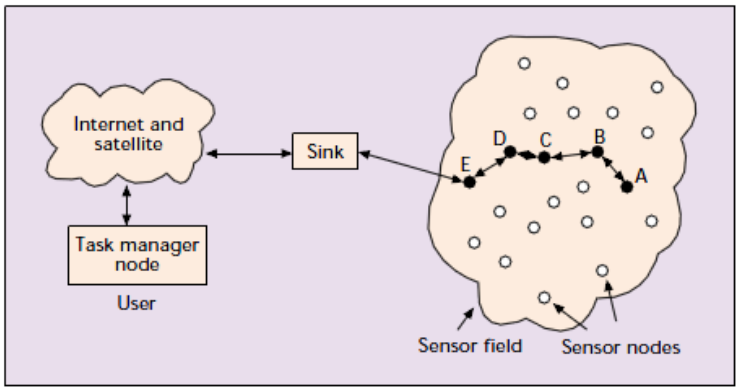
The consumer product industry has of course seen the development, and many networked computing devices with sensors are now being sold to consumers. The prime example is smart phones, which can have an accelerometer, an orientation sensor, a light sensor, a camera and many other sensor types. Another example is IP cameras that have an integrated web server. Such devices offer additional components compared to sensor nodes typically used in WSN's. In the context of sensor-based CEP applications, we should keep a broad view of what devices classify as sensor nodes, including devices such as smart phones and IP cameras, as they can be used as sensor input in a CEP system. The trend is towards even more devices being networkable, and researchers now envision a future where many every-day products are tagged with Radio-Frequency Identification (RFID) chips with computational capabilities to become smart objects connected to the Internet of Things [13]. In such a world, the boundaries between everyday objects and sensor nodes fade away.

2.3 Wireless Sensor Networks

A field that has received much attention has been related to issues about how a number of sensor nodes can best be combined into wireless sensor networks (WSNs). Some researchers denote that such networks can be labeled as wireless multimedia sensor networks (WMSNs) when multimedia sensors are included [7]. A common scenario is to deploy many low-cost, low-power sensor nodes in or close to an environment to gather data related to specific conditions. This can mean both predefined locations chosen by how suitable they are to gather data, or more random locations where a large number of nodes are scattered in the environment to achieve a sufficient sensor coverage without needing to choose specific sensor locations for each node. This can be necessary if the environment is unreachable or to vast to allow manual deployment. [10]

As the sensor nodes need to communicate wirelessly and are low-powered, they need efficient self-organizing network capabilties but algorithms and protocols that allow them to use as little energy as possible on their communication tasks. To achieve this the nodes may perform computing operations on-board to process raw sensor data and determine if the sensed data is interesting enough to transmit and thus use their own and other node's scarce power supply.

Nevertheless, much of WSN research is about finding the best network models and protocols to answer the specific needs of WSNs, see figure 4.



Figur 4: Wireless Sensor Network. Image copied from [10, p. 103]

The sensor field is scattered with densely deployed low-cost, low-power sensor nodes that need to implement multihop routing in order to allow to reach other nodes and the sink, which acts as a gateway to external networks where the collected data is sent. The WSN needs to handle a number of issues [10]:

- **Fault tolerance:** Some of the nodes may fail because of power failure, environmental interference or damage, and the network must be able to continue to operate despite this.
- **Scalability:** The network needs to handle situations where a great number of nodes are densely deployed.
- **Production costs:** As the network may consist of numerous nodes, the price of each node must be kept to a minimum.
- **Hardware constraints:** Sensor nodes have inherent limitations because of size, power-requirements and limitations of the components that compose them. The deployment may also inpose additional hardware constraints such as limited node size or resillience to environmental conditions.
- **Network topology:** The nodes can be deployed either manually one by one or mass-deployed, e.g. from an airplane. Once deployed, other factors such as changes in the environment, node mobility, node failure or redployment of additional nodes may alter the sensor network topology.
- **Environment:** The WSNs may be required to operate in a wide variety of environments such as oceans, arctic areas or even space.
- **Transmission media:** Sensor nodes may use a variety of wireless transmission methods, such as radio or infrared media.
- **Power consumption:** This is such an important factor for sensor nodes, because of power supply limitations, that the design of the network needs to take power consumption of the nodes into account.

The continued technical advances have also allowed researchers to add actor entities to WSN, yielding wireless sensor and actor networks (WSANs) [14]. In such networks, the data collected from sensor nodes are used to determine actions that should be taken from the actors, which typically are more rich in resources such as power, processing and communication capabilities as well as having capabilities to intervene in the environment.

2.4 Sensor-based complex event processing applications

Networks of data-gathering computing devices present many possibilities to application developers. One approach is to perform complex event processing (CEP) based on sensor input. Sensor nodes are used to gather information about physical conditions in an environment, and this information can either be transmitted as it is, sent only when certain conditions or thresholds are met, or sent after performing computations according to some relevant algorithm. Some of the sensor nodes may be given additional roles so that the sensor system not only sends real sensor data, but also aggregated data or event notifications when certain conditions are met.

CEP applications can rely on diverse types of sensors to detect simple events and combine such simple events into complex events when certain conditions are met. The steps in such applications are to:

1. Retrieve data from sensors in the environment.
2. Detect events from the data.
3. Detect complex events

2.4.1 Data Retrieval

Sensor data can be retrieved in a number of ways. Sensor nodes need to be connected to the server running the CEP application directly or indirectly. Several standard ports, such as USB, serial or parallel ports are available, but in many cases sensor nodes are wireless and rely on gateway nodes and custom software to be used to communicate with the sensor nodes. The custom software presents functionality to fetch the sensor data and often offers different ways to view data or extract it. The gateway nodes are connected to external networks, e.g. the internet, and can communicate with nodes in the sensor network, so that to use the sensor data externally, the custom software or other components need to interact with the gateway nodes. There are also initiatives that attempt to provide operating systems or other software that allow for more custom usage and communication with the nodes. An example is the Contiki OS [15], an operating system that allows individual nodes to use the Internet protocol and hence have IP addresses and the ability to be adressed directly. Using such frameworks gives developers more direct control over the nodes and how the data is transmitted even though the gateway nodes still may be necessary to access the underlying sensor node network. Such networks may rely on radio-based protocols such as ZigBee or 6LoWPAN and therefore need gateways or routers that can perform network format translation [16].

Two basic patterns are used to retrieve data from sensor nodes, push and pull [17]. The push pattern is used when the data source, the sensor node, is the originator of the data transmission. On the other hand, if the client application requests the data by sending a request to the sensor node, and the node responds with the data, the pull pattern is used. A sensor-based application may rely on both mechanisms.

The data received from sensor nodes is at its most basic level an array of bytes, and the way the data is encoded may take any form deemed advantageous. In some cases, the data may be encoded in a human-readable way, or the data may be compressed or encrypted to deal with

concerns about package size or security. The received data needs to be parsed by the application to be used for event detection.

2.4.2 Event Processing

Once data from sensors are retrieved and parsed by the application, they can be used to define or detect events. We define events as any occurrence that may be of interest to the application domain. Events can be classified as simple or atomic events if they are based on readings from a simple sensor source, such as a temperature reading or a motion sensor detecting a MOTION event. By combining multiple simple events, possibly from multiple sensor sources, composite or complex events can be determined [18].

Complex events can be detected from a single node if the node is implemented with capabilities to process sensor readings over time or the combination of readings from multiple sensors on the node. Complex events can also be detected from multiple sensor nodes if they can collaborate in a sensor network, or in a CEP application that receives input from mutliple sensors on multiple nodes. For instance if we combine sensors that detect if a window breaks, motion sensors and precious items tagged with RFID, we can define the complex event burglary_detected as the combination of the simple events window_broken, motion_detected and item_removed occuring in an area within 10 minutes.

There is a multitude of application domains for sensor-based event detection, and in most cases the goal is to react to the detected events. If the above burglary_detected event was to occure in a security system, the wanted reaction may be to sound the alarm and deploy security officers.

2.4.3 A state-of-the-art CEP application: CommonSens

Due to demographic conditions in the developed world, there will be a significant increase in elderly populations requiring attention in the near future. This has spawned interest in developing systems that can help in reducing costs of health care for the elderlies. Automated home care is one possible solution, and CEP applications based on sensors can participate in such solutions. CommonSens is a multimodal CEP system for automated home care. It is designed to be based on heterogeneous sensors and detect complex events and deviations from complex events [19].

In order to allow for sensors to detect activities of daily life, a wide variety of different sensor types are required. In [19] , Søberg distinguishes between three sensor categories: RFID tags and readers, programmable sensors and sensors that require substantial processing to be meaningful, i.e. multimedia sensors. Sensors of all these categories can be used with CommonSens in order to monitor elderlies in their home so that safety can be provided in an unintrusive manner. Both wanted and unwanted complex events are defined in the system, so that health personel can interact with the patient if wanted events fail to happen, e.g. taking medication, or unwanted events occur, e.g. falling on the floor and getting injured. The system can also be used for other patients that can gain from this type of monitoring, such as people with chronic diseases.

In CommonSens, events are defined as state or state transitions of interest. The home of the monitored person is described in the system using an environment model, and several locations of interest (LoI) can be defined. Events occur at LoIs and are timestamped with start and end times. Event relations can be defined using 5 classes of concurrency (equals, starts, finishes, during, overlaps) and a consecutiveness class (before).

Sensors in the system are defined with capabilities (e.g. MotionDetect) and not only physical sensors that sense the environment are used, logical sensors that combine and process input from other sensors can be defined, and also external sources that keep persistently stored data and can be used by logical sensors.

Based on the defined sensor capabilities, the users can describe events using a query-based event language. At first, atomic queries are described by a state condition (capability – operator - value), and possibly a LoI and temporal conditions (when should the condition occur, how long should it last, what ratio of time should the condition, etc). Atomic queries can be combined into complex queries using logical operators and the event relations described above.

Once the environment, sensors and queries are set up, CommonSens can start processing events using 3 core components. Based on the queries, CommonSens keeps record of the sensors that provide relevant data. The **data tuple selector** pulls data tuples from the relevant sensors. As many queries may run in parallel, the **query pool** is the structure that holds the individual queries, and the **query evaluator** processes the data and queries and evaluates if conditions are matched or not. The intent of CommonSens is to send notifications when certain conditions or deviations are detected.

CommonSens is a prime example of the type of sensor-based CEP applications that could benefit from sensor data reuse.

2.5 Data storage and reuse

Development and testing of event processing applications based on sensor input present a number of challenges. An important challenge in this respect is related to the data used to test the validity of the algorithms detecting events and responding to these events. The applications may be dependent on a large number of heterogeneous sensor nodes, and at some point, real data is needed to verify that the application will react as intended when in use. This can cause significant time to be spent on practical things such as setting up the sensor nodes, recharging or changing batteries on the sensor nodes, and more importantly to generate the events in the real world that should be captured by the sensors and detected by the application. Setting up hundreds of sensors in a forest and creating a forest fire might not be something researchers want to do many times for testing purposes! Even a simple event such as a person falling to the ground can become quite tedious if you wish to tune an algorithm based on camera input that detects this event. If someone needs to walk in front of a camera and fall to the ground repeatedly, a lot of time is wasted.

Researchers might also want to share data sets with others that can benefit from it. Some groups have access to resources that other researchers could greatly benefit from. If it were easy to share data from experiments, the research community could benefit as a whole. Additionally, it may be useful for researchers to compare their solutions. Using the same data

sets is the best way to achieve this task. The Opportunity project [20] aims to develop generic principles, algorithms and system architecture to recognize activities and contexts from sensors. In 2011 the project called for participants in the Opportunity Challenge. The goal of the challenge was to compare different techniques and methods using a **common benchmarking dataset.** This shows that there is a real need for tools that help to simplify the process of sharing data.

The goal of this thesis is to design and create an application that can store sensor data so that it can be reused by applications as many times as necessary. For instance, if a team is working with home care applications, they can set up an environment with a variety of sensors, simulate some daily activities of an elderly person, and store the data. Later, the application should be capable of outputting the same data as often as needed.

3. Requirements analysis

As discussed in the previous chapter, there is a need for a system that can provide a simple way of storing and reusing data sets for sens-based applications. In this chapter, I will analyze the requirements such a system needs to meet.

3.1 Target users

To be able to clearly analyze what requirements the application should aim to fulfill, we should start by identifying the users that can benefit from using this application. The aim is to simplify data reuse for sensor-based event detection/processing applications. As such applications are meant to be used for specific purposes, capturing and reacting to real live data, they give value to users when real live data is used. The main user groups that may benefit from storing and reusing data are researchers and developers that are developing algorithms and programs related to sensor-based event detection/processing.

Researchers in this field attempt to find new or better ways of performing event detection and processing. One of the goals of these researchers is to develop algorithms that can detect and process events better, faster, more precisely and with fewer false positives. To be able to recognize if one algorithm is better fit for a task than another, they must be compared. Data reuse provides a way to use the exact same data as often as needed, and could therefore aid in comparing algorithms precisely.

Further, researchers are confined to the equipment and facilities that are available to their research centers. For sensor-based research, this can present limitations as it relies heavily on equipment that can be hard to find and expensive, i.e. sensors and sensor nodes set in a specific environment. Some researchers may buy all the equipment they can find while others have funding for only a few sensor nodes. Tools that may assist in sharing sensor data among researchers anywhere in the world could improve the conditions for researchers with equipment-constraints. Nevertheless, many more research centers may also benefit from sharing sensor data as all experiments have differences, and more available data amounts to more flexibility for the researchers.

When implementing event detection and processing systems, testing is necessary, as with any other system development. Software testing will generally involve at least 2 aspects. The first is related to the implementation process and is often done informally by the implementers. The goal of these tests is to verify that the code works as intended and discover bugs as early as possible. The second test aspect is performed when developers deliver larger parts of the system, or the completed system, and the system needs to verified and validated. This is often a more formal phase where stakeholders seek assurance of requirements fulfillment and quality of the software. Testing is of course a complex subject that often will involve many steps and processes on several levels and it will depend on the goals and scope of the development project. My point is that even a small informal project will require implementers to test code along the way, and potential project managers to verify the bigger picture before delivering or presenting the project results.

All testing requires some type of input, be it mock data or direct user input. For a sensor-based application, the main input is data from an external sensor. This data is either pushed

from the sensor node and the application receives and processes it, or the sensor nodes need to be pulled to fetch sensor data. Whether the test-data is simulated or not, an application-external source of input is necessary to emulate this behavior. The sensor-based application will need to fetch the input data from a simulated, emulated or real sensor node at some point during testing to verify that data is retrieved and handled correctly. Our tool for data reuse could therefore also benefit developers, testers and project managers.

3.2 Functional requirements

The main goal of the intended tool is to store sensor data so that it can be reused later. The first requirement derived from this goal is that our tool must be able to store sensor data persistently. This implies that the user sets up sensor nodes, configures the nodes and/or the data reuse tool so that they can communicate, most often over a network, and starts a data retrieval and storage session. The application should then retrieve and store data in a session that lasts as long as the user sees necessary.

The second main requirement is that our application should allow the user to easily reuse the data. This implies that the user configures the data reuse application and the receiving application for communication, initiates a data retrieval and forwarding session, i.e. retrieves data from persistent storage and forwards it to the receiving application. In other words, the goal is to emulate the sensor nodes using the data gathered earlier. Key requirements to achieve this goal is that data is forwarded to the receiving application in the same format as received and at the right time.

3.2.1 Data storage

This is the first main requirement we wish to provide with our data reuse application. The goal is to receive data from sensor nodes and store it persistently. First of all, we need to provide a way to communicate with the sensor nodes. Sensors are generally physically located outside of the computer running the sensor based application, in the environment that is to be monitored. Several sensors are often combined on sensor nodes, and the node communicates with other nodes or computers through wired or wireless communication. Sensor node communication has several aspects:

- Direct/Indirect:
Here we differentiate between sensor nodes that can be communicated with directly and nodes that can only be reached through gateways or a software proxy. WSNs are becoming common and can be a valuable source of data for sensor-based applications. In WSNs, several sensor nodes are spread out in an environment and form a discrete network. Sensor nodes can communicate between each other through appropriate protocols, and often they will only communicate with outside networks through specific gateway nodes. In some cases, an overlay network may provide means to communicate with the individual sensor nodes directly, while other WSNs rely on custom server software and protocols to access the sensor nodes and their data.

- Push/Pull:

Sensor nodes can generally be considered to be push or pull. A pushing sensor node will send its data without being asked to by an external source. The decision of when to send data is implementation-specific and may be specific time-intervals or when certain conditions are met. In a pull-model, the sensor node will send data when it receives a request to do so. The node may send sensor data from all its sensors or provide a way to identify towards which sensors the request is aimed.

- Network stack:

Sensor nodes can use a number of ways to send sensor data. At the bottom, the physical/link layer provides the means of communication for networked entities that are connected or within reach of one another. 2 nodes may communicate through a wired or wireless connection. Some sensor nodes may use a protocol at this layer, such as the RS-232 protocol, but this is impractical if there is a need to connect to several sensor nodes because such protocols may only provide connection for directly connected nodes. The network layer allows connected entities to form a network, and find routes between entities not directly connected. Sensor nodes commonly form networks and require a protocol on top of the network, at the transport layer, to define the end-to-end communication between nodes, such as TCP/IP. Again there may be layers built on top of the transport layer, the application layer, to ease communication. The HTTP protocol is of course a very common example of an application layer protocol that may be used by sensor nodes.

Our focus towards the network stack need only be concerned with how the sensor nodes communicate with external network entities. We wish to communicate with sensor nodes as a sensor-based application would, and therefore are not concerned with the inner workings of sensor network protocols, only how the sensor network communicates with external networks. If there is a gateway node or custom software acting as a proxy, we only need to give attention to the protocol used to communicate with that node or software, as it will give us the data we need, even if that implies using elaborate protocols inside the sensor network. The same goes for sensor networks using a protocol such as 6LoWPAN. If we can use IPv6 and a gateway translates packets to another network layer protocol such as 802.15.4, we still only need to handle IPv6 network packets.

- Controllable sensor nodes:

With some types of sensor nodes, communication may involve other aspects than sensor data. Sensor nodes may offer some form of external control. Examples may be remote-controlled moveable nodes or IP cameras with remote pan, tilt and zoom controls. With such nodes, the input source may be affected by the commands. If a node is moved, it may capture other data than if it was not. At a simpler level, a pull sensor node may also be viewed as a controllable node: You send a data request command and receive data back. This does not however affect the input, only the frequency and amount of data gathered.

Our tool should cover a range as wide as possible with regards to the channels and modes of communication it can handle. Controllable sensor nodes present a major issue though. The controls sent to a node may affect the data captured by sensors. If we store and reuse this data, the sensor-based application may choose to send different control messages during subsequent

reuse of the data. To be able to provide data for all potential commands, we would need to gather all possible data at all times, which we can not achieve. For simpler cases, it may be possible to cover a satisfying number of cases. If we gather data from pull sensors, we may pull data at short intervals and return data that was pulled closest to when the application decides to pull. If we pull data every 5 seconds from a sensor in a storage session, and in an experiment get a pull from the sensor-based application 22 seconds into a data retrieval session, we can simply return the data that was pulled 20 seconds into the data storage session. Our application must therefore allow to implement such functionality.

We also need to consider the sensor data packet formats that sensor nodes may generate. Sensor data may include a wide variety of types and formats. First of all, different sensors yield different data types. Using a broad definition of sensors, we may receive anything from temperature sensor voltages to multimedia streams. We should be able to handle data storage of as many data types as possible. Data packets may also be formatted in any way judged to be advantageous by sensor node software implementers, such as XML, HTML or SSI (Simple Sensor Interface), and tool should have capabilities to handle a wide variety of such formats.

Once the data is acquired, it will need to be stored persistently and made easily available for later use or analysis. Some users may wish for this to be done seamlessly and do not care how the data is persisted, but others may wish to be able to access stored data for analysis, integrate the data with other software or for any reason use specific means of storage. We must therefore design a solution that can potentially satisfy different needs, yet provide a default means of storage for simple use.

3.2.2 Data reuse

Although storage of sensor data in itself can provide value, e.g. for analysis, emulating sensor nodes with stored data is at the core of the application we wish to provide. This emulation is achieved by retrieving sensor data from persistent storage, constructing data packages in the correct format and finally sending data packets with correct data at the right time. Most of the issues discussed for data storage under 3.2.1 are also relevant for data retrieval, but an important difference is that the application is required to determine when to send data, and which data to send where. For pull-sensors, the application needs to have a policy on what data to send when it receives a request for data, i.e. should the last received sensor values be sent, or the values that are nearest in time to when the request was received? The user should be allowed to set this policy as needed. For push-sensors, the data packets must be sent according to when data was received. This means that our application must contain a timing mechanism that can handle this requirement.

When reusing data, the sensor-based application expects to receive sensor value packets by a specific network protocol, i.e. the protocol that our application used to retrieve the sensor values from the sensor nodes. Our application must therefore be able to replicate the behaviour and protocols used by the sensor nodes. It should also be possible to change the behaviour and protocols used to send sensor data so that the same data packets may be sent to several sensor-based applications using varying types of communication.

Optimally the application should also separate between the sensor data and the data package format used to transmit it so that the same data source can be reused by different sensor-based applications expecting other packet formats if needed. This would also allow users to easily

use the tool to emulate sensor nodes from any sensor data, be it real data gathered by other researchers or emulated data. As one of the goals of the application is to provide a simple way to share data among researchers, data and format separation can aid in achieving this goal.

It should be simple to reuse data, and the user should be able to select what data to use from the storage sessions that are available. The users should be allowed to either choose sensor data from individual nodes, or replicate an entire session and send data from all nodes used in that session.

3.2.3 Other functional requirements

To simplify use, our system should present a simple and intuitive graphical user interface. As there are 2 different main ways of using the application (ie to store or reuse data), there should be 2 separate views that clearly present relevant functionality for the mode of operation being used.

Another aspect of our tool is that since it may be used potentially for such a wide variety of different sensors and sensor nodes, it will probably need to be extended for new sensor nodes and data packet formats. As target users are developers or researchers that work at least partly on software, we can assume that writing source code will be accepted as a natural way to extend the tool for particular needs. We should therefore aim to make it as simple as possible to implement new channels for communication to and from sensor nodes and sensor-based applications, and internal implementations to handle different packet formats or present new functionality.

For some application domains, localization of the sensor nodes can be a central aspect. Under most circumstances, location of the sensors plays a key role in detecting events at specific locations. Still, for some applications, the sensor nodes may be fixed and their location is known in the application. In other cases, the location of the nodes may be unknown or change if the nodes are mobile, and the nodes themselfes are required to provide data about their location. The data reuse application we aim to provide should be able to incorporate this concept into it’s semantic domain if necessary.

When storing sensor data aimed for reuse by a CEP application, there are usually events that are interesting for the application domain that are occuring in the sensed environment and should be detected by the system receiving the data. Adding the ability to tag specific events in a stored session in our application could provide users with a useful feature. Using such tagged events, one could inform users of the events in the user interface of the application to provide a simple way to assess if the sensor-based CEP application was detecting events correctly.

3.3 Architectural requirements

As an important aspect of our data reuse application is that it should be extendable, its architecture must be devised with extendability in mind. There needs to be clearly separated components handling different stages in the sensor data flow:

- Communication with external entities, i.e. sensor nodes and sensor-based applications

- Parsing and formatting of data packets and the data contained in these

- Persistent storage of data and how the storage module is accessed

These components should be easily extendable by relying on abstract types as much as possible for inter- and intra-module connection points. To allow for simple extendability, we also need to consider that our choice of programming language should be a common, well-known and widely used language.

The application will have to act both as a client and a server in both main modes of operation. It will need to handle many heterogeneous connections simultaneously (e.g. harvest data from several push and pull sensor nodes), and be able to rely on fine-grained timing to send requests or data packets at the right time. Our choice of runtime environment must therefore provide robust capabilities for:

- Threading: the application will need to handle several connections and handle incoming data

- Timing: push-sensor packets need to be sent at specific times and pull-requests sent at specific intervals

- Concurrency: the application may need to provide concurrent multi-thread access to resources

- Networking: the application should allow for multiple heterogeneous connection types simultaneously

The runtime environment used should be available for the most commonly used platforms in order not to limit the potential users who can benefit from using our application.

3.4 Performance requirements

Several aspects need to be analyzed to define the performance requirements. A stated previously, we do not wish provide a tool to perform load-testing specifically, even though this functionality would add to the value of the application and the application probably will be useful for such a purpose in many cases. We aim to provide a tool to help with data reuse for the development and validation of sensor-based applications or algorithms and therefore should aim to handle connections and data quantities that can be reasonably assumed to be necessary to accomplish this task. Again, as the application should cover a wide variety of sensor-based application domains, it is difficult to define requirements that would cover all potential ways of using it. Sensor nodes may yield different quantities of data at different frequencies, and we should handle a broad range of combinations of these two aspects. We also need to ensure that performance is satisfactory both when harvesting data and emulating sensor nodes.

First we need to consider how many connections our application should be required to handle. Assuming that systems based on sensor nodes generating larger amounts of data require fewer nodes to be tested and verified, and systems based on nodes generating little data need many of these, we should base the required amount of connections on the latter. We must therefore ensure that the appliaction can handle a large number of nodes.

Of course using the number of connections as a performance requirement gives little meaning without considering the amount of data being transmitted through each connection. For each connection, the data packets received will need to be parsed and sent to the persistent storage module. Sensor nodes generating multimedia streams would require another level of performance per node than nodes yielding one numerical sensor value, even though data transmission frequencies might complicate the matter as simple data sent at very high frequencies also would impose great performance requirements.

The implemented application will be tested thoroughly with different requirement classes in the evaluation section, chapter 6.

4. Design

The goal is to design and implement a system for storing sensor data and subsequent reuse of stored data in the context of research and development of sensor-based event detection and processing applications. Given the requirements that have been discussed, we will in this part discuss options and choices made.

4.1 Architecture - components in the system

Our desired tool can be best understood as a data storage and reuse system: it basically harvests data, stores it, and redistributes it later. Data is harvested from and redistributed to external entities, meaning that we need components to communicate with these entities. Data is to be stored to and retrieved from persistent storage, so we need a component that handles this functionality. Sensor data is received from sensor nodes in data packets conforming to a certain format, and we want to separate the format from the data, so we need components to deal with this too.

As we wish to provide as much flexibility as possible in order to provide a tool that meets different needs, these components need to be clearly separated so that users may adapt the implementation to their needs. We also aim to have loosely-coupled components so that replacing or changing components be as simple as possible. Additionaly, it should be simple to apply combinations of communication and format components for different sensor types.

4.1.1 Communication components

The task of these components will be to handle channels connected to external entities, i.e. sensor nodes and sensor-based applications. When harvesting data, we need to establish channels to sensor nodes. The module must be able to handle several connections to allow for our tool to harvest data from several discrete sensor nodes. The easiest way to gather data is from sensor nodes that push data. For these, we only need to create a connection to the node, and read incoming data when it is received. But for sensor nodes that must be pulled for data, the connection channel is more complicated as communication is two-way; our tool sends requests for data that the sensor nodes reply to. As the tool is harvesting data for later reuse, the channel must be able to generate requests at specific intervals so that the data harvesting frequency can cover potential later requests satisfyingly.

Similarly, when reusing data, we need to establish channels to the sensor-based application. Since the tool is to emulate sensor nodes, we need to copy the communication pattern used by the setup at hand. If the target application expects channels for each sensor node or all data sent through one channel, that's what we should give it. The user of our tool should have the option to configure how data is to be sent to the receiving application. In essence we simply take the opposite role compared to when we received data, and emulate the sensor node communication behavior.

As the communication module needs to handle multiple channels, some of which that may require implementing specific timing needs, it will need to use threading and timers. It may

receive many data packets from some sensor nodes simultaneously and at the same time send requests for data to other nodes. This can be implemented in a single thread working intensively or split into multiple threads doing less work. Using a single thread may save some operating system resources and limit thread context switching, thus being more efficient. On the other hand, it may also prove to not be efficient enough to handle situations where many communication operations need to be executed within short timespans, or not precise enough when handling multiple request-events that need to be triggered simultaneously. For this reason it is my view that splitting communication operations into separate threads is the better alternative. By using a thread for each communication channel, the result is many threads doing little work. A consequence of this is that instead of needing to implement the efficiency ourselves, we can rely on the platform we choose for the implementation and its ability to perform thread context switching efficiently. We may lose some control, but most of the available relevant programming platforms can be assumed to perform thread context switching more efficiently than one or few programmers trying to achieve the same within one big complicated thread. Separating communication channels into separate objects and threads also gives the benefit of being easier to maintain and extend.

4.1.2 Format handling components

Sensor nodes will typically transmit sensor data in a particular format. To allow for easy reuse of sensor data for applications requiring different formats than the ones used by a particular sensor node, we need to separate the data from the format used to transmit it and store sensor values. To achieve this we will need to parse the data packets after they are received when harvesting data, and reformat it when data is to be retransmitted during later reuse. This step must be performed between the communication and storage interaction steps. To allow for simple reuse of parsing and reformatting code, it should be divided into separate objects capable of parsing or reformatting specific formats.

4.1.3 Storage components

To allow for flexibility in choosing persistent storage, this functionality should be separated into its own components. There should be a component to interact with storage that presents storage capabilities to other components in the system, and the back-end storage component that takes care of persistency and only interacts with the component providing storage capabilities to the rest of the system. Using such a pattern facilitates changing the back-end storage without needing to change the interface used by the rest of the system. Note that this pattern also permits using an application-external storage system, such as a database running in its own process possibly on another computer than the data reuse tool.

4.1.4 Other components

The users need a simple way to interact with our tool, and therefore it should include a graphical user interface. As for the other modules, we should separate this part into a separate

component for clarity and flexibility. Finally there is a need to have controller components that manage, interconnect and interact with the other components.

An overview of our desired architecture can be visualized in figure 5.

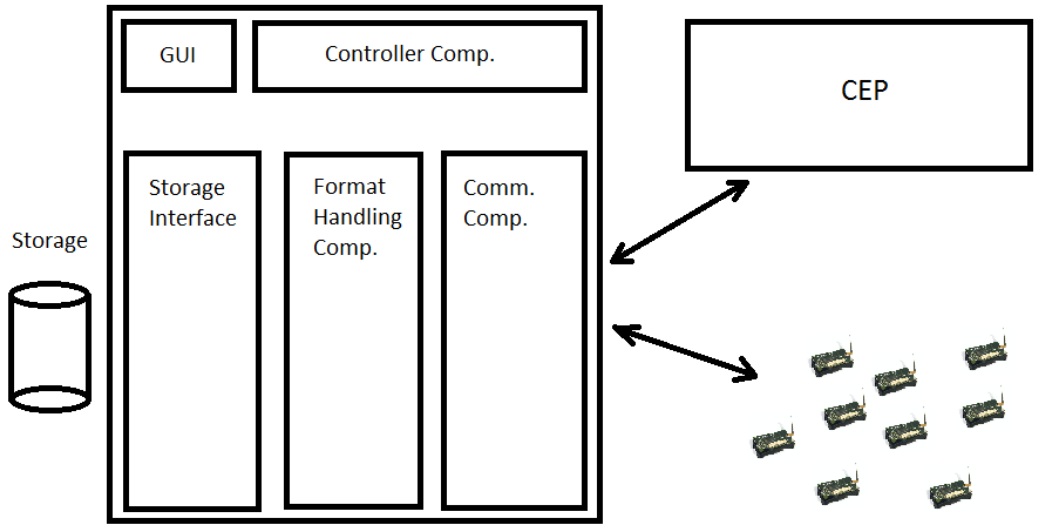


Figure 5: Architectural overview.

4.2 Data model

The core of the system we are designing is the sensor data, but to have a robust, complete and extendable system we need to model the entire application domain in an adapted manner.

Data generation relies on sensor nodes that include one or more sensors generating sensor values, see figure 6.

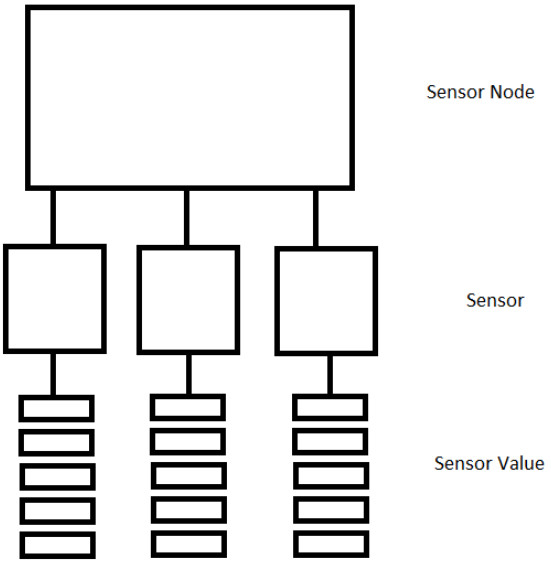


Figure 6: Relationship between sensor nodes, sensors and sensor values.

Sensor-based applications generally rely on a number of sensors, potentially of different types. We want to provide users with a way to categorize sensor nodes into types so that the users easily can define new nodes of a certain type, or find stored data from these node types later. As sensors also generally can be categorized into types, such as temperature or pressure, we further categorize sensor node types to consist of one or more sensor types:

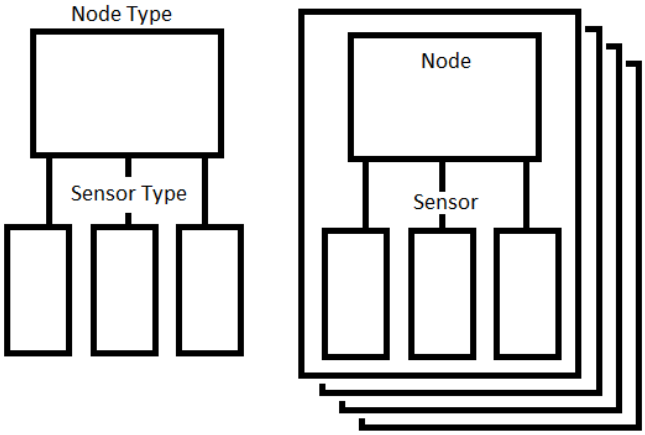
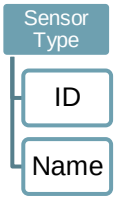
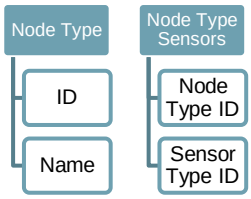


Figure 7: Node and sensor types versus defined nodes and sensors

Sensor types are defined simply by their sensing capability and given an id:

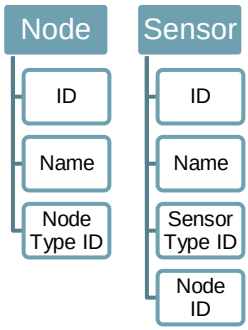


Once required sensor types are defined, sensor node types can be defined based on the types of sensors available:

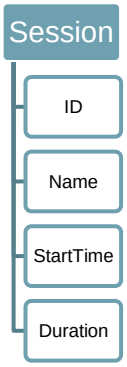


An example of a node type would be the MicaZ from Crossbow, that could be defined to contain two sensor types, e.g. temperature and light.

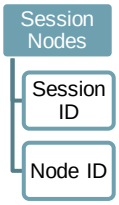
From these node types, users can define specific nodes easily, and the sensor types on such nodes follow implicitly. The defined nodes and sensors can then be given a unique ID and a name to differantiate them from other nodes or sensors of the same type.



Once our data model has defined nodes and sensors with unique IDs, sensor values received in data packets can be linked to the sensor IDs that generated them. Nevertheless, we need a way to differantiate between data generated during different runs. We therefore attach a unique session id to each run, a name, when it was run, and the duration:

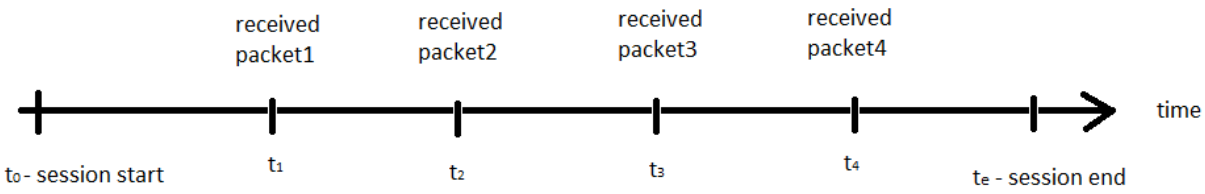


We should also keep track of nodes involved in a session, and in case we end a storage session before all nodes have transmitted data, this information should be tracked separately:



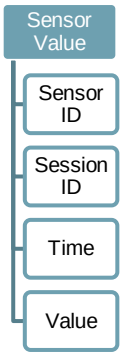
An important aspect of the sensor data is the timing aspect. When emulating sensor nodes in a data retrieval session, we need to transmit data at the right time for push nodes, and respond with the correct data for pull nodes. To be able to reproduce the transmission of data correctly

we need to store the time upon which the data was received. If we view the timeline of a storage session as:



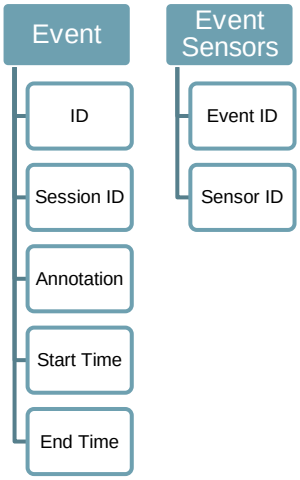
we will store the timestamp $t_n - t_0$ for the sensor value, i.e. the time of reception since time of session start. Assuming that the transimssion delay from our tool to the receiving sensor-based application will be equal for all emulated nodes, as all transmission will be sent through the same network infrastructure, we can send all data packets with the same intervals as they were received relative to t_0.

To model the sensor values adequately we then need the following information:



A major goal of sensor-based applications is to monitor and react to events in specific environments, such as forest fires or home care. With this in mind, it is evident that sensor location can play an important role in deciding which sensors are relevant for reuse for a user of our tool. Location can be described in a number of ways, such as GPS coordinate for a forest fire monitoring application or a three-dimensional Cartesian coordinate for a home care application. We must also notice that certain sensor nodes can be mobile and the timing aspect of the location must also be taken into account for such nodes. Node location can then be modelled as follows:

Another aspect of reusing data sets is that once the data has been generated and stored, the user may have an idea about the events of interest that have actually occurred during the storage session and have been sensed by the node sensors in the environment monitored. The data model should include a notion of event annotation so that events may be tagged in a given session. It may also be beneficial to link the annotated event to the sensors involved in sensing the event:



5. Implentation

Given the requirements and design discussed, we want to provide an implementation of the tool. In this section I will present the implementation-specific details of the resulting program, starting with implementation-relevant choices made and then continuing with important components. The components involved in the two modes of operation of the tool are similar, but as they behave differently depending on mode, I will present the implementation for each mode separately.

5.1 Implementation choices

5.1.1 Platform

Our requirements analysis yielded a number of demands from the platform. The tool should be capable of running on a variety of operating systems to allow as wide a range of environments to run it. It should also use a common programming language so that implementation of new features, adapting existing ones and implementing code that handles new sensor node types be less of a burden than needed.

Additionally, we found some specific technical requirements; we need threading, timing, concurrency and broad communication capabilities to provide the desired functionality in a satisfying way.

In my view, the Java platform stands out as the best candidate. It provides a robust, OS-independent platform and uses a well-known programming language that allows for using the modularization we need. Java is considered to be moderately slower than compiled platform-specific languages such as C++ or C but some tests show that Java is nearly equal to C++ [21], so for our purposes the performance of Java will probably suffice and therefore the other benefits can be given priority.

5.1.2 Communication protocol

Sensor nodes may use a variety of modes of communication, and I do not aim to implement all possibilities. Instead, implementing functionality for common forms of sensor node communication seems a more adapted approach. Current research indicates that the most relevant communication form to focus on is IP-based communication as this is how most modern sensor nodes network or aim to network with external systems [Basere dette på dok.]. The Java platform allows for implementing a number of IP-based communication protocols used by sensor nodes, but chief among these is the User Datagram Protocol (UDP), and I therefore use UDP as the reference implementation of communication channels.

5.1.3 Persistent storage

It seems obvious that using a database is the reasonable choice of persistent storage. Persistent storage requires writing data to non-volatile storage, which in many cases means hard disk drives or solid state drives, and the alternative to a database is writing the required data to files and storing these files in the filesystem of operating system running the tool. Even though it is easier to write data to a file than to set up a database, connect to it and then interact with it, this also implies that we would need to devise a system for organizing our data model in one or more files. Using a database we can create tables according to the data model and reap all the benefits of a database system, mainly a standard query language that allows us to store and retrieve data in a simple manner and the consistency and durabillity of data.

The system we implement is designed so that users can change storage medium according to preference, but to provide an easy-to-use reference implementation I chose to use the H2 Database Engine which is a small and fast Java SQL relational database that can be easily embedded into an application. It also stores the database in a single file, which can make it easier to share sensor data with other researchers or developpers. Other benefits include the small footprint of the database system (only about 1 MB of disk required for the system), simplicity of use as it is only Java and consists of one JAR-file, and has an integrated web server for simple access to stored data.

5.1.4 Unimplimented components

Due to time constraints, some of the discussed components have not been implimented. For knowledge about localization, I have not implemented any specific features. Such features could be useful to provide users with information about the location of nodes, e.g. to decide which nodes should be chosen if the user only wants to select certain nodes for reuse or analyze movement or event patterns with regards to location. Nevertheless, a user can define a sensor for location in our system and store data about location as a sensor value. The channel interfaces in the implementation are also defined so that they can both send and receive data or messages in case there may be specific situations where this behaviour is useful.

Another feature I determined could be useful is a system for tagging events occuring within a stored session. As location data, this is a feature that would give information to users that can be useful, by allowing the user to see clearly in real-time when events are occuring and thus providing a ground truth that can be used to measure the performance or validity of specific event detection methods and techniques. This is left as a future enhancement of the system, and the implemented framework can be extended with this feature without the need to make significant changes to the core components.

5.2 Components implementation

In order to implement an extendable system with clearly separated modules, we will make extensive use of abstract types, i.e. Java Interface types. All inter-component interaction

should be realized in this manner to provide a solution in which explicit implementations can be replaced as wanted without needing to change code in dependant components.

5.2.1 Data reception, parsing and storage

The tasks of the components involved in this mode are to:

- Interact with external nodes to fetch data (push or pull pattern)
- Parse data packets received if necessary
- Store data

The components interacting with sensor nodes must implement the ReceiverChannel interface (in the no.uio.ellhanse.channel.receiver-package). This interface extends Runnable so that it can be run as a thread, and requires the following methods to be implemented:

- **run()**: Used to initialize the channel (e.g. set up a socket) and perform additional operations if needed. The channel is started by the usual call to channelThreadName.start() that runs this method.
- **stopRunning()**: Stop the operations and thread gracefully. This thread and the following one all use the same pattern: the run-method initializes necessary objets, sets a boolean variable named `running` to **`true`**, then goes into a while-loop that keeps testing if the `running` variable is true. This method sets `running` to **`false`** and if there are objects that are blocking, such as a socket for channels, they are closed or interrupted.
- **receive()/send(byte[] data)**: Depending on how a channel needs to communicate with a sensor node, one or both of these methods can be used.

Implementing channels in this fashion provides a way to simply start the channels and letting them run while also allowing controller objects to perform operations by calling the receive and send methods to handle more complex protocols. For instance a push sensor node may only require the channel to start receiving data from a certain socket port. In such a case the socket can be initialized during run-method execution, and simply call receive() in a loop from the run() method (code from UdpPushReceiverChannel in package no.uio.ellhanse.channel.receiver):

```
public void run() {
	try {
		socket = new DatagramSocket(localPort);
	} catch (SocketException se) { /** removed for brevity **/ }
	running = true;
	if(sessionStartTime == 0)
		sessionStartTime = System.currentTimeMillis();
	while(running) {
		receive();
		}
	}
```

The receive()-method of the channel then only needs to call the blocking receive()-method of the socket and handle packets when they are received:

```
public void receive() {
	receiveBuffer = new byte[bufferSize];
	receptionTime = -1;
	try {
		receivePacket = new DatagramPacket(receiveBuffer, bufferSize);
```

```
            socket.receive(receivePacket);
            receptionTime = System.currentTimeMillis();
        } catch(IOException ioe) { /** removed for brevity **/ }
```

Some push-nodes may require an initial request message before data transmission starts or permit messages to alter frequency of sampling. In such cases the send() method may be used for this purpose.

Once the channel has received a packet containing sensor data, it creates an object of type SensorPacket (from package no.uio.ellhanse.receiver) that holds the reception time, the session ID that is created when the session starts, and the data bytes:

```
SensorPacket packet = new SensorPacket((receptionTime - sessionStartTime),
receivePacket.getData());
```

The SensorPacket object must then be sent to an object that handles the parsing of data from an array of bytes to values that can be connected to a sensor. The task of parsing is specific to each type of sensor node and needs to be adapted accordingly. In order to centralize control of the channel and parser objects that are relevant for a particular sensor node, we use a Receiver-object that is used to start and stop the channel and parser, and acts as a broker between the channel and the parser. Receiver-objects need to implement the Receiver-interface (in package no.uio.ellhanse.receiver) that requires the following methods to be implemented:

- **start():** Initialize communication with a particular node, i.e. intialize the channel and the parser
- **stopRunning():** Stop the channel and parser
- **dataReceived(SensorPacket p):** Called by the channel when a packet is received and needs to be parsed

The receiver also holds the session ID and the node ID of the sensor node it receives data from, and sets the sessionID variable in the SensorPacket object. Since these operations are similar for many types of sensor nodes, we provide a generic implementation, GenericReceiver in package no.uio.ellhanse.receiver.impl . When the ReceiverChannel object has received a data packet and created the SensorPacket object, it calls it's parent receiver-object's dataReceived-method:

```
parentReceiver.dataReceived(packet);
```

And the receiver-object sets the session ID and calls the parser to parse the packet:

```
public void dataReceived(SensorPacket packet) {
      /** removed some code for brevity **/
      packet.setSessionId(sessionId);
      parser.parsePacket(packet);

}
```

To allow the channel thread to work independently from the parsing operations, parser objects also run their own thread. Parser objects must implement the Parser interface (from package no.uio.ellhanse.parser) that holds the following methods:

- **run()**: The Parser interface extends Runnable so the parser can run in a thread, and this method starts the thread.
- **stopRunning()**: Used to stop running the parser gracefully

- **parsePacket(SensorPacket sensorPacket)**: Called by other objects to parse a SensorPacket.

To further simplify implementation of parsers, an abstract class implementation of a parser thread based on a concurrent queue is provided, GenericQueueParser in package no.uio.ellhanse.parser . It is implemented by using a LinkedBlockingQueue from the Java concurrency packages. When parsePacket() is called with a SensorPacket object, it simply queues the SensorPacket in the LinkedBlockingQueue:

```
public void parsePacket(SensorPacket current) {
	try {
		parseQueue.put(current);
	} catch(InterruptedException ie) {  /** removed for brevity **/ }
}
```

and the run()-method of the parser makes a continuous call to the LinkedBlockingQueue's take()-method that blocks until there is an element ready to be parsed:

```
public void run() {
	running = true;
	SensorPacket current = null;
	while(running) {
		try {
			current = parseQueue.take();
			parse(current);
		} catch(InterruptedException ie) {  /** removed for brevity **/}
}
```

When such an element appears, it is parsed by the GenericQueueParser subtype's implemention of the parse method. The parse method needs to extract sensor values contained in the data packet and map the values to specific sensor IDs. When parsing is complete, it calls the storage component's method to insert the sensor value.

As we are using a relational database to store data persistently, we need to take a closer look at the data that is to be stored. In many cases, the sensor data consists of values in a certain unit of measure, e.g. degrees for temperature, or even textual representations of a certain condition such as "motion detected" for a motion sensor. We do not need to care about what their meanings are in our application, only that they are scalar and can be stored as strings of text in the database, for instance as varchar's.

On the other hand, some sensors may generate large binary streams, like image or sound sensors. These binary values need to be handled differently from scalar values as they are potentially much larger. I therefore chose to use 2 database tables to store sensor values, one for scalar values storing values of type VARCHAR, and one for binary values storing values of type BLOB. For this reason there are also 2 methods available to store data:

- insertScalarSensorValue(int sessionId, int sensorId, long time, String data)
- insertBinarySensorValue(int sessionId, int sensorId, long time, InputStream is)

for storing data into the respective tables. These methods are implemented in the H2StorageDAO class that is located in the no.uio.ellhanse.storage.h2 package. Storage will be explained details later in this chapter.

5.2.2 Data retrieval and reuse

The main tasks in this mode are in essence the reverse of the data storage mode:

- Fetch sensor data from persistent storage
- Send data to the relevant formatter/unparser to create correctly formatted data packets
- Send data packets to the sensor-based application at the right time

Persistent storage contains a set of sensor values, and the user chooses the data that is to be sent (described later). Storage data retrieval is implemented in a thread that fetches data from the chosen data set from storage at specific intervals. To achieve this, the default Java implementation of timers is used, java.util.timer. This timer can be set to run a timertask (java.util.timertask) when needed. We use it to call a method that retrieves sensor data for the next period, i.e. until the next run of the timertask:

```
timer.scheduleAtFixedRate(new TimerTask() {
                  public void run() {
                        retrieveData();
                  }
            }, delay, period);
```

This timer is started when the data storage retriever-thread is started, and the retrieveData()-method that is called from the timertask inserts the retrieved data into a LinkedBlockingQueue named sensorValues . The retriever thread goes on to call the take()-method of the LinkedBlockingQueue, that blocks until elements are present, in a loop:

```
SensorValue current;
running = true;
while(running) {
      current = null;
      try {
            current = sensorValues.take();
            sendToProvider(current);
      } catch (InterruptedException ie) { /** removed for brevity **/ }
}
```

In effect, storage data retrieval is then achieved by 2 threads: the first thread calls retrieveData() at specific intervals (value of the period variable, e.g. 5000 milliseconds) through the timer and timertask and fills the LinkedBlockingQueue, the other thread loops and tests if there are elements in the LinkedBlockingQueue, and handles them as they are found.

The retrieveData method fetches sensor values from persistent storage. As described in the data model sectionof the last chapter, the values are stored in a table that also contains the sensor ID, session ID and time of the sensor reading. Before the retriever thread is started, it is given a set of sensor/session combination for which data must be retrieved. The retrieveData method therefore needs to run a query and select tuples where the session and sensor ID are in the set of chosen sensor/session combinations. Additionaly, the queries run at specific intervals and we only need to query every tuple once, so the query must include a clause to filter on tuple time as well.

This is implemented by building a query clause, when we start the thread, for the session and sensor ID’s that are to be retrieved:

```
public void buildWhereClauseSensors() {
	whereClauseSensors = new StringBuilder(" ( ");
	boolean first = true;
	for(String key : sensorNodes.keySet()) {
		int separatorIndex = key.indexOf("_");
		if(separatorIndex != -1) {
			int sessionId =
	Integer.parseInt(key.substring(0, separatorIndex));
			int sensorNodeId =
	Integer.parseInt(key.substring(separatorIndex + 1, key.length()));
			if(first) {
				first = false;
			} else {
				whereClauseSensors.append(" or ");
			}
			whereClauseSensors.append("( session_id = " + sessionId +
				" and sensor_id = " + sensorNodeId + " )");
		}
	}
	whereClauseSensors.append(" ) ");
}
```

and each time retrieveData is called, we add the time interval clause to the sensor/session clause before the query is run and the resulting sensor values added to the sensorValues LinkedBlockingQueue described earlier:

```
StringBuilder whereClause = new
	StringBuilder(whereClauseSensors.toString());
whereClause.append(" and ( time > ");
whereClause.append(lastSessionTimeValue);
lastSessionTimeValue += period;
whereClause.append(" and time <= ");
whereClause.append(lastSessionTimeValue);
whereClause.append(" ) ");
sensorValues.addAll(App.storage.getSensorValues(whereClause.toString()));
```

The retriever thread keeps track of the sensor nodes it emulates in a Java hashmap (java.util.HashMap) called `sensorNodes`. This map uses a String of format <sessionId>_<sensorId> as a key to point to a provider-object that keeps references to the relevant unparser/formatter and channel for each sensor, and several sensors may share the same provider-object as their data may come from one node. The sendToProvider-method then finds the right provider from the hashmap, and calls its receiveSensorValue-method. The provider-objects must implement the Provider-interface (from package no.uio.ellhanse.provider) that require the following methods to be implemented:

- **run()**: Data providers run in their own threads and need to implement this method
- **stopRunning()**: Used to stop the provider gracefully
- **receiveSensorValues(SensorValue sensorValue)**: Used to receive sensor values from the storage data retriever thread.
- **setStartTime(long startTimeMillis)**: The providers need to time their data transmission carefully.

As the work done by provider objects is similar for most of the sensor node types that are to be emulated, I provide an implementation of the provider interface, GenericProvider in the no.uio.ellhanse.provider.impl-package, that can be used in most cases.

The GenericProvider (hereby simply called provider) object uses a sychronized list (created from the java.util.Collections.synchronizedList()-method) to keep track of sensor values that are to be updated soon. When the storage retriever thread calls receiveSensorValue in the provider, the sensor value is added to the synchronized list of the provider:

```
public void receiveSensorValues(SensorValue sensorValue) {
	sensorValues.add(sensorValue);
}
```

and the provider object's thread checks the list for new values in a loop:

```
public void run() {
	running = true;
	while(running) {
		if(!sensorValues.isEmpty()) {
			/** removed for brevity, described under **/
		} else {
			try {
				Thread.sleep(50);
			} catch (InterruptedException ie) {
				/** removed for brevity **/
			}
		}
	}
}
```

and sleeps for a short time (50 milliseconds in above code) if there are no values present.

When sensor values are present in the list, the provider thread sorts the elements according to the time they should be sent:

```
synchronized(sensorValues) {
	Collections.sort(sensorValues, new Comparator<SensorValue>() {
		@Override
		public int compare(SensorValue arg0, SensorValue arg1) {
			return (int) (arg0.getTime() - arg1.getTime());
		}
	});
}
```

Note that code accessing a synchronizedList through iterators need to be synchronized to assure thread-safety. Therefore the above code is set in a Java synchonized-block.

As different sensor values from the same sensor node sent in one packet are stored in separate database tuples (in the sensorValue-tables), but will have the same timestamp, the provider thread needs to find all the sensor values with the same timestamp to correctly reassemble the data packet:

```
// Create a list to hold sensor values that are to be sent next
List<SensorValue> nextPacketValues = new ArrayList<SensorValue>();
// Remove first element in list that has the smallest timestamp
nextPacketValues.add(sensorValues.remove(0));
// Hold the timestamp in a variable
long nextPacketTime = nextPacketValues.get(0).getTime();
// Find all sensor values with that same timestamp
synchronized(sensorValues) {
	for(SensorValue sensorValue : sensorValues) {
		if(sensorValue.getTime() == nextPacketTime) {
```

```
                        nextPacketValues.add(sensorValue);
                }
        }
        // Remove sensor values with that timestamp from the list of
        // remaining values that are to be sent later
        for(SensorValue sensorValue : nextPacketValues) {
                sensorValues.remove(sensorValue);
        }
}
// Format sensor values to a sendable packet and schedule channel update
scheduleDataUpdate(nextPacketTime, unparser.unparse(nextPacketValues));
```

Once the sensor values that should next be sent are found, the provider calls an unparser-object that formats the values correctly and returns a byte array that can be sent through the network channel. With the bytes to send ready, the provider needs to schedule the transmission of data for push-nodes or schedule to set the channel's data to the correct packet at the right time for pull-nodes so that the channel can respond with the appropriate data if a data request comes in from the sensor-based application. This is performed by the scheduleDataUpdate-method of the provider, that sets p a timer and timertask to call the channel's updateData-method:

```
public void scheduleDataUpdate(long nextPacketTime, final byte[] data) {
        timer.schedule(new TimerTask() {
                @Override
                public void run() {
                        channel.updateData(data);
                }
        }, new Date(startTime + nextPacketTime));
}
```

The channel can then react to the data update appropriately. If it is part of a push-node provider, it needs to send the data packet to the sensor-based application promptly, whereas if it is part of a pull-node provider, it only updates the data that is to be responded with if a request was to be received.

A difference to note here is that a push-node provider does not need to run it's own thread; when the timertask of the node's GenericProvider is run, it calls the channel's updateData-method and that thread can be used to send the packet. Pull-node provider channels on the other hand need to listen for incoming data request packets and therefore need to run in their own thread.

The provider channels need to implement the ProviderChannel interface (package no.uio.ellhanse.channel.provider) and must implement the following methods:

- **run():** Used to run the channel in a thread. Can be left empty if the channel does not need to run in a thread.
- **stopRunning():** Stop the thread gracefully if channel is running a thread
- **updateData(byte[] data):** New sensor values are available and the channel must either send a data packet to the sensor-based application if emulating a push-node or update the data that will be responded if a request comes in for a pull-node.
- **send(byte[] data)/receive():** Used to interact with the sensor-based application, i.e. send data packets and optionnaly listen for data request packets.

As stated earlier, if the channel emulates a push-node, when it receives a data update, it only needs to call send:

```
public void updateData(byte[] data) {
	send(data);
}
```

which sends the data, e.g. for a push-node that uses UDP:

```
packet = new DatagramPacket(data, data.length, address, port);
socket.send(packet);
```

For a pull-node, the channel can keep the bytes to send in a variable:

```
public void updateData(byte[] data) {
	this.sndData = data;
}
```

The channel thread listens for requests and sends responses in a loop:

```
public void run() {
	running = true;
	while(running) {
		byte[] rcvBytes = receive();
	// Here we just send back data-packets for whatever request we get.
		if(rcvBytes.length > 0) {
			send(sndData);
		}
	}
}
```

The receive-method simply listens for packets (in this case datagrams since UDP is used) and returns the content bytes of the packet:

```
public byte[] receive() {
	rcvData = new byte[rcvBufferSize];
	rcvPacket = new DatagramPacket(rcvData, rcvBufferSize);
	try {
		socket.receive(rcvPacket);
	} catch (IOException e) { /** removed for brevity **/ }
	return rcvPacket.getData();
}
```

And the send-method simply sends bytes:

```
public void send(byte[] data) {
	sndPacket = new DatagramPacket(data, data.length,
rcvPacket.getAddress(), rcvPacket.getPort());
	try {
		socket.send(sndPacket);
	} catch (IOException e) { /** removed for brevity **/ }
}
```

5.2.3 Persistent storage

The physical storage of data is achieved using a database following the data model design discussed in the former chapter. Interaction with the database is performed using the classes located in the no.uio.ellhanse.storage package.

The majority of the methods are presented in the interface StorageDAO that includes method declarations to create new elements, update existing elements, retrieve one, many or all elements and delete elements for the different elements described in the data model such as nodes and sensors. Additionally, this interface presents methods to initialize and close a connection to the database, and methods related to updating the tables that keep data about relations between the various elements, such as the node – session relation that stores the nodes used in a particular session.

Our storage implementation is based on the H2 Database Engine [22], which is embedded into the application as a JAR-file. The class implemeting StorageDAO, H2StorageDAO, is located in the no.uio.ellhanse.storage.h2 package. When the application is started, the initialize()-method of H2StorageDAO is called. This method loads the H2 driver and sets up a connection to the H2 database engine:

```
public boolean initialize() {
	boolean success = false;
	/** removed some code for brevity **/
	try {
		Class.forName("org.h2.Driver");
	} catch (ClassNotFoundException cnfe) { /** removed for brevity **/
		return success;
	}
	try {
		conn = DriverManager.getConnection(dbUrl, username, password);
	} catch (SQLException se) { /** removed for brevity **/
		return success;
	}
```

This will automatically start the H2 engine if necessary, and create the database if it doesn't exist. To initialize the tables, I provide the class H2TableCreator that is used to create the tables used by the application if necessary. The initialize()-method of H2StorageDAO goes on to do:

```
H2TableCreator creator = new H2TableCreator(conn);
boolean tablesOk = creator.createTables();
```

and returns `true` if the tables are set up correctly. The createTables method starts by checking if the tables are already present in the database by checking if a table named sensor is present using the testIfCreated()-method:

```
public boolean testIfCreated() {
	boolean exists = true;
	try {
		stmt = conn.createStatement();
		rs =  stmt.executeQuery("SELECT * FROM
INFORMATION_SCHEMA.TABLES where TABLE_SCHEMA = 'PUBLIC' AND TABLE_NAME =
'SENSOR' ");
		if(!rs.isBeforeFirst())
			exists = false;
	} catch(Exception e) { /** removed for brevity **/ }
	} finally {
		try { rs.close(); } catch(Throwable ignore) { /* ... */ }
		try { stmt.close(); } catch(Throwable ignore) { /* ... */ }
	}
	/** removed for brevity **/
```

```
        return exists;
}
```

If the tables are not present, the createTables method goes on creating them:

```
public boolean createTables() {
        if(testIfCreated())
                return true;
        for(int i = 0; i < tables.length; i++) {
                /** removed for brevity **/
                if(!createTable(tables[i], tables[i].split(" ")[3]))
                        return false;
        }
```

by calling the createTable method:

```
private boolean createTable(String query, String tableName) {
        boolean success = true;
        try {
                stmt = conn.createStatement();
                stmt.execute(query);
        } catch(Exception e) {
                /** removed for brevity **/
                success = false;
        } finally {
                try { stmt.close(); } catch(Throwable ignore) { /* ... */ }
        }
        return success;
}
```

for each String in an array of CREATE TABLE statements:

```
String[] tables = {
                " CREATE TABLE session (" +
                " id INT AUTO_INCREMENT PRIMARY KEY," +
                " name VARCHAR(255) UNIQUE," +
                " duration BIGINT DEFAULT 0," +
                " start_time TIMESTAMP DEFAULT NOW())"
                ,
                " CREATE TABLE sensor_type (" +
                " id INT AUTO_INCREMENT PRIMARY KEY," +
                " name VARCHAR(255) UNIQUE," +
                " is_blob BOOLEAN)"
                ,
                " CREATE TABLE node_type (" +
                " id INT AUTO_INCREMENT PRIMARY KEY," +
                " name VARCHAR(255) UNIQUE)"
                ,
                " CREATE TABLE node_type_sensors (" +
                " node_type_id INT," +
                " sensor_type_id INT," +
                " PRIMARY KEY(node_type_id, sensor_type_id)," +
                " FOREIGN KEY(node_type_id) REFERENCES node_type(id)," +
                " FOREIGN KEY(sensor_Type_id) REFERENCES sensor_type(id))"
                ,
                " CREATE TABLE node (" +
                " id INT AUTO_INCREMENT PRIMARY KEY," +
                " name VARCHAR(255) UNIQUE," +
                " node_type_id INT," +
```

```
            " FOREIGN KEY(node_type_id) REFERENCES node_type(id))"
            ,
            " CREATE TABLE sensor (" +
            " id INT AUTO_INCREMENT PRIMARY KEY," +
            " name VARCHAR(255)," +
            " sensor_type_id INT," +
            " node_id INT," +
            " FOREIGN KEY(sensor_type_id) REFERENCES sensor_type(id)," +
            " FOREIGN KEY(node_id) REFERENCES node(id))"
            ,
            " CREATE TABLE scalar_sensor_value (" +
            " time LONG," +
            " data VARCHAR(2048)," +
            " sensor_id INT," +
            " session_id INT," +
            " PRIMARY KEY(time, sensor_id, session_id)," +
            " FOREIGN KEY(sensor_id) REFERENCES sensor(id)," +
            " FOREIGN KEY(session_id) REFERENCES session(id))"
            ,
            " CREATE TABLE binary_sensor_value (" +
            " time LONG," +
            " data BLOB," +
            " sensor_id INT," +
            " session_id INT," +
            " PRIMARY KEY(time, sensor_id, session_id)," +
            " FOREIGN KEY(sensor_id) REFERENCES sensor(id)," +
            " FOREIGN KEY(session_id) REFERENCES session(id))"
            ,
            " CREATE TABLE session_nodes (" +
            " session_id INT," +
            " node_id INT," +
            " PRIMARY KEY(session_id, node_id)," +
            " FOREIGN KEY(session_id) REFERENCES session(id)," +
            " FOREIGN KEY(node_id) REFERENCES node(id))"
            };
```

Next, the sensor types and implemented node types are inserted into the database using the insertData method:

```
for(int i = 0; i < startValues.length; i++) {
      /** removed for brevity **/
      if(!insertData(startValues[i]))
            return false;
}
```

The insertData method uses code similar to the createTable code to insert values from the startValues String array:

```
String[] startValues = {
"INSERT INTO sensor_type(name, is_blob) VALUES('temperature', FALSE)," +
      "('humidity', FALSE)," +
      "('pressure', FALSE)," +
      "('gravity', FALSE)," +
      "('magnetic', FALSE)," +
      "('gyroscope', FALSE)," +
      "('light', FALSE)," +
      "('motion', FALSE)," +
      "('image', TRUE)," +
```

```
        "('sound', TRUE)," +
        "('video', TRUE)",
"INSERT INTO node_type(name) VALUES('udp_push_xml_test'),
        ('udp_pull_xml_test')",
"INSERT INTO node_type_sensors(node_type_id, sensor_type_id) VALUES" +
        " (SELECT id FROM node_type WHERE name = 'udp_push_xml_test'," +
        "  SELECT id FROM sensor_type WHERE name = 'temperature')" +
        ",(SELECT id FROM node_type WHERE name = 'udp_push_xml_test'," +
        "  SELECT id FROM sensor_type WHERE name = 'humidity')" +
        ",(SELECT id FROM node_type WHERE name = 'udp_push_xml_test'," +
        "  SELECT id FROM sensor_type WHERE name = 'motion')" +
        ",(SELECT id FROM node_type WHERE name = 'udp_pull_xml_test'," +
        "  SELECT id FROM sensor_type WHERE name = 'temperature')" +
        ",(SELECT id FROM node_type WHERE name = 'udp_pull_xml_test'," +
        "  SELECT id FROM sensor_type WHERE name = 'humidity')" +
        ",(SELECT id FROM node_type WHERE name = 'udp_pull_xml_test'," +
        "  SELECT id FROM sensor_type WHERE name = 'motion')"
};
```

Since node types need to at least have some custom parsing and unparsing code, it is my view that it is acceptable to require the user to add node types to the database manually by entering their name and on-board sensor types to the above INSERT statements. Once a node type is present in the database (and is implemented in the application code), the users will have the possibility of creating as many nodes of the type as necessary in the GUI.

Once tableCreator.createTables returns in the H2StorageDAO object, the boolean tablesOk variable is used to test if the tables are set up properly. The application exits if not, otherwise the application can go on and present the user with the GUI for normal application usage and the H2StorageDAO object is ready to present storage interaction functionality to the other components.

The StorageDAO interface declares basic methods that perform typical create-, get-, delete- or insert-operations towards specific tables in the database. The methods implemented in H2StorageDAO executes them by constructing SQL queries and executing them using the standard JDBC technology defined in the java.sql package. For operations that return tuples, we construct objects from each row using classes defined in the no.uio.ellhanse.model package. These classes map to the tables and their variables basically mirror the attributes of the corresponding table, e.g. the node_type table which is defined as:

```
" CREATE TABLE node_type (" +
" id INT AUTO_INCREMENT PRIMARY KEY," +
" name VARCHAR(255) UNIQUE)"
```

in the tables array, is defined as the following Java class:

```
public class NodeType {
        int id;
        String name;

        /** removed constructors, getters and setters for brevity **/
}
```

And to retrieve a SensorType from the database, the H2StorageDAO implements the following method declared in the StorageDAO interface:

```
public SensorType getSensorType(int id) {
	Statement stmt = null;
	ResultSet rs = null;
	SensorType sensorType = null;
	String query = "SELECT * FROM sensor_type WHERE id = " + id;
	try {
		stmt = conn.createStatement();
		rs = stmt.executeQuery(query);
		rs.next();
		sensorType = new SensorType(rs.getInt(1), rs.getString(2),
rs.getBoolean(3));
	} catch(Exception e) {
		/** removed for brevity **/
	} finally {
		try { rs.close(); } catch(Throwable ignore) { /* ... */ }
		try { stmt.close(); } catch(Throwable ignore) { /* ... */ }
	}
	return sensorType;
}
```

To centralize the acces point to methods that require more complex interaction with persistent storage, a class named StorageUtils is also provided and includes methods that use the StorageDAO implementation (such as H2StorageDAO) to perform their task and utility methods that implement various helpful tasks. An example is the getNewSession method that creates a new session with a session_<timestamp>-format as session name:

```
public static Session getNewSession() {
     DateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ss");
     Date now = new Date();
     return App.storage.createSession("session_" + dateFormat.format(now));
}
```

5.2.4 Controller components

With all the central functionality implemented, we need components that can bind the other parts together in a simple manner. To achieve this, I have implemented 2 session manager classes, ReceiverSessionManager and ProviderSessionManager in package no.uio.ellhanse.app, that are used respectively for receiving and storing or reusing sensor data. Their task is to be the central structure that keeps track of the nodes the session is to receive data from or emulate, and be a central point of control to start and stop the session. The goal is to present a simple interface for the user interaction components, such as a GUI, to run sessions.

They do this by keeping a list of wrapper-objects, ReceiverWrapper or ProviderWrapper, that contain references to the necessary objects for each node, such as channels and formatters, in the session. If a user has set up some sensor nodes to store sensor data from and wants to start a session, the user interaction component creates a new session by calling StorageUtils.getNewSession() and then creates a ReceiverSessionManager object by calling it's constructor with the new session object as an argument:

```
public ReceiverSessionManager(Session session) {
	receivers = new ArrayList<ReceiverWrapper>();
	this.session = session;
}
```

Next the user interaction component needs to create ReceiverWrappers with the appropriate information for each sensor node by calling the constructor:

```
public ReceiverWrapper(InetAddress address, int port, int sessionId, int
            sensorNodeId, String nodeType) {
      this.address = address;
      this.port = port;
      this.sessionId = sessionId;
      this.sensorNodeId = sensorNodeId;
      this.nodeType = nodeType;

      channel = SubtypeObjectFactory.getReceiverChannel(nodeType, address,
            port);
      parser = SubtypeObjectFactory.getParser(nodeType, sensorNodeId);
      receiver = SubtypeObjectFactory.getReceiver(nodeType, channel,
            parser, sessionId, sensorNodeId);
}
```

The SubtypeObjectFactory class will be described later in this section.

The ReceiverWrapper's can then be added to the ReceiverSessionManager object by calling:

```
public void addReceiver(ReceiverWrapper receiver) {
      receivers.add(receiver);
}
```

The session is started by the user interaction component by calling the startReceiving method of the ReceiverSessionManager:

```
public void startReceiving() {
      List<Integer> nodeIds = new ArrayList<Integer>();
      for(ReceiverWrapper receiver : receivers) {
            nodeIds.add(receiver.getSensorNodeId());
            receiver.start();
      }
      App.storage.insertSessionNodes(session.getId(), nodeIds);
}
```

For each ReceiverWrapper, it's start method is called, which will start the Receiver, Channel and Parser that were described earlier in this chapter. The same pattern is used by the ProviderSessionManager.

Similarly stopReceiving or stopProviding can be called to stop the session and nodes. Continuing with the ReceiverSessionManager, stopReceiving is implemented as follows:

```
public void stopReceiving() {
      for(ReceiverWrapper receiver : receivers) {
            receiver.stopRunning();
      }
      App.storage.updateSessionDuration(session.getId(),
            System.currentTimeMillis() - session.getTimestamp().getTime());
}
```

The SubtypeObjectFactory class called by the wrapper object constructors is simply a class that can be used to create objects of the correct channel, parser, unparser, receiver or provider implementing class, as the member variable is of the interface type. For instance, the

ReceiverWrapper must have a channel, a parser and a receiver object. These members are declared using the interface of the type:

```
Receiver receiver;
ReceiverChannel channel;
Parser parser;
```

In the constructor, the SubtypeObjectFactory is used to instantiate the objects, e.g. :

```
channel = SubtypeObjectFactory.getReceiverChannel(nodeType, address, port);
```

The SubtypeObjectFactory class contains String members of the node types that are implemented, e.g. :

```
final static String TYPE_UDP_PUSH_XML_TEST = "udp_push_xml_test";
```

and has methods to construct all needed objects for a declared node type:

- Receiver
- ReceiverChannel
- Parser
- Provider
- ProviderChannel
- Unparser

When called, the methods check the nodeType argument to determine the correct implemented class to instantiate and return. For instance the getReceiverChannel method:

```
public static ReceiverChannel getReceiverChannel(String type, InetAddress
            address, int port) {
      if(type.equalsIgnoreCase(TYPE_UDP_PUSH_XML_TEST)) {
            int bufferSize = 2048;
            return new UdpPushReceiverChannel(port, bufferSize);
      }
      return null;
}
```

It is up to the users who implement new node types to add relevant lines in this class, and the declared node type name must match the name declared in the database.

5.2.5 Graphical User Interface

To provide a simple user interaction component to the users of the application, I have implemented a graphical user interface. It uses Swing which is the primary Java toolkit to build GUIs, and all GUI-related files are located in the no.uio.ellhanse.gui package.

Once the application has been started, the database is initialized and connected to as described in the last section. If this fails, the application exits, otherwise, the GUI is launched and presents the sensor data reception and storage view, see figure 8.

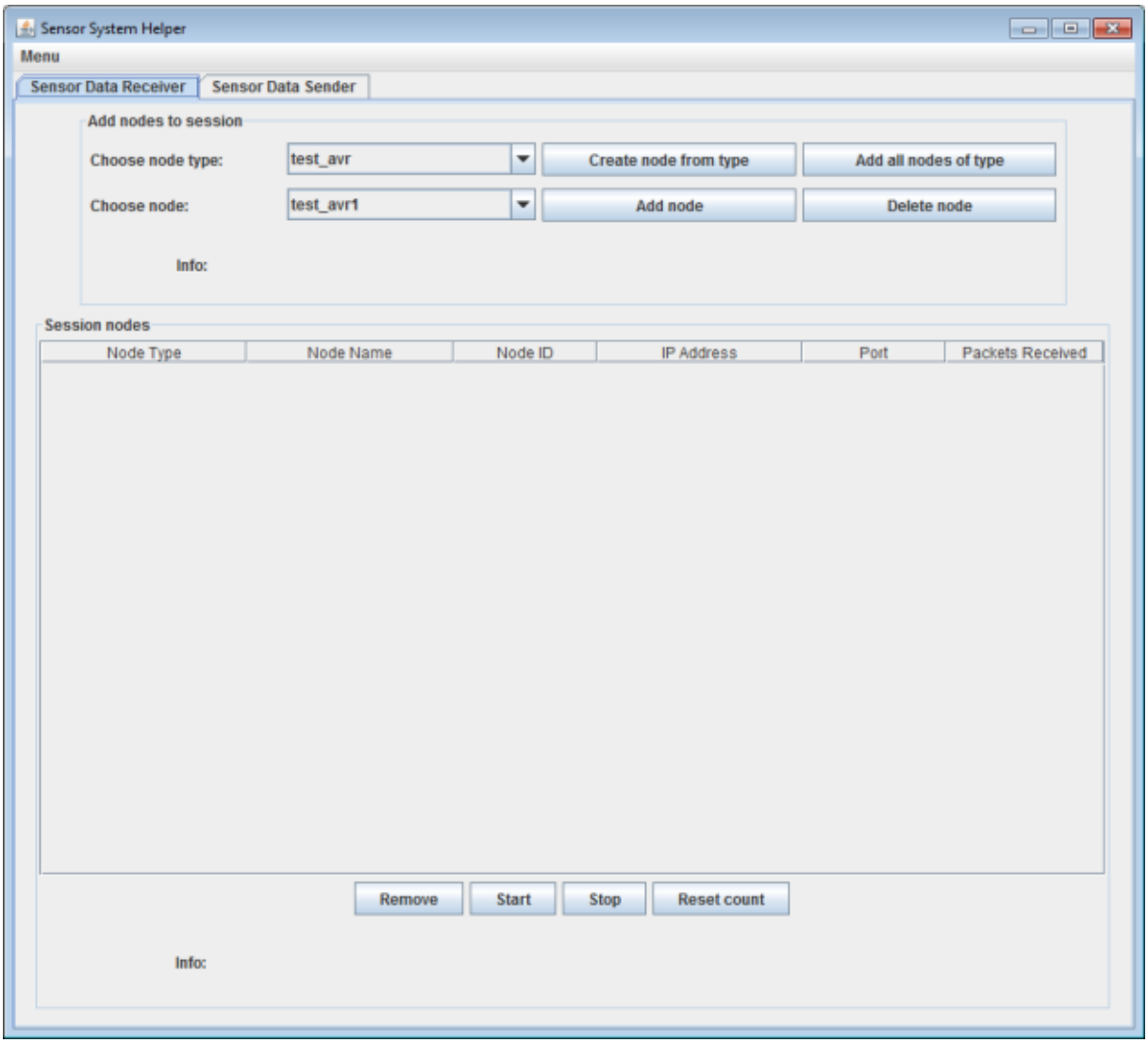


Figure 8: Sensor data storage view in GUI

The GUI consists of a frame that contains a menu bar with a menu item to quit the application, and a tabbed pane that has 2 tabs for the 2 modes of operation. The sensor data reception and storage mode, shown above, contains an upper area to create, delete and select nodes for the session. The lower area contains information about the selected nodes and allows the user to edit IP address and port for the nodes to connect with. Push nodes only need localhost as IP address as they receive data packets on a local port, but pull nodes require an external IP address to connect to the sensor node. The information about selected nodes for the storage session also keep count of the number of packets that have been received from the node.

The user can remove previously selected nodes, start and stop a session, and reset the packet count under the session node information section. Both areas also contain an info-area that is used to present the user with information when actions are performed or attempted but are fail or not allowed.

For the sensor data reuse mode, the GUI uses a similar type of view, see figure 9.

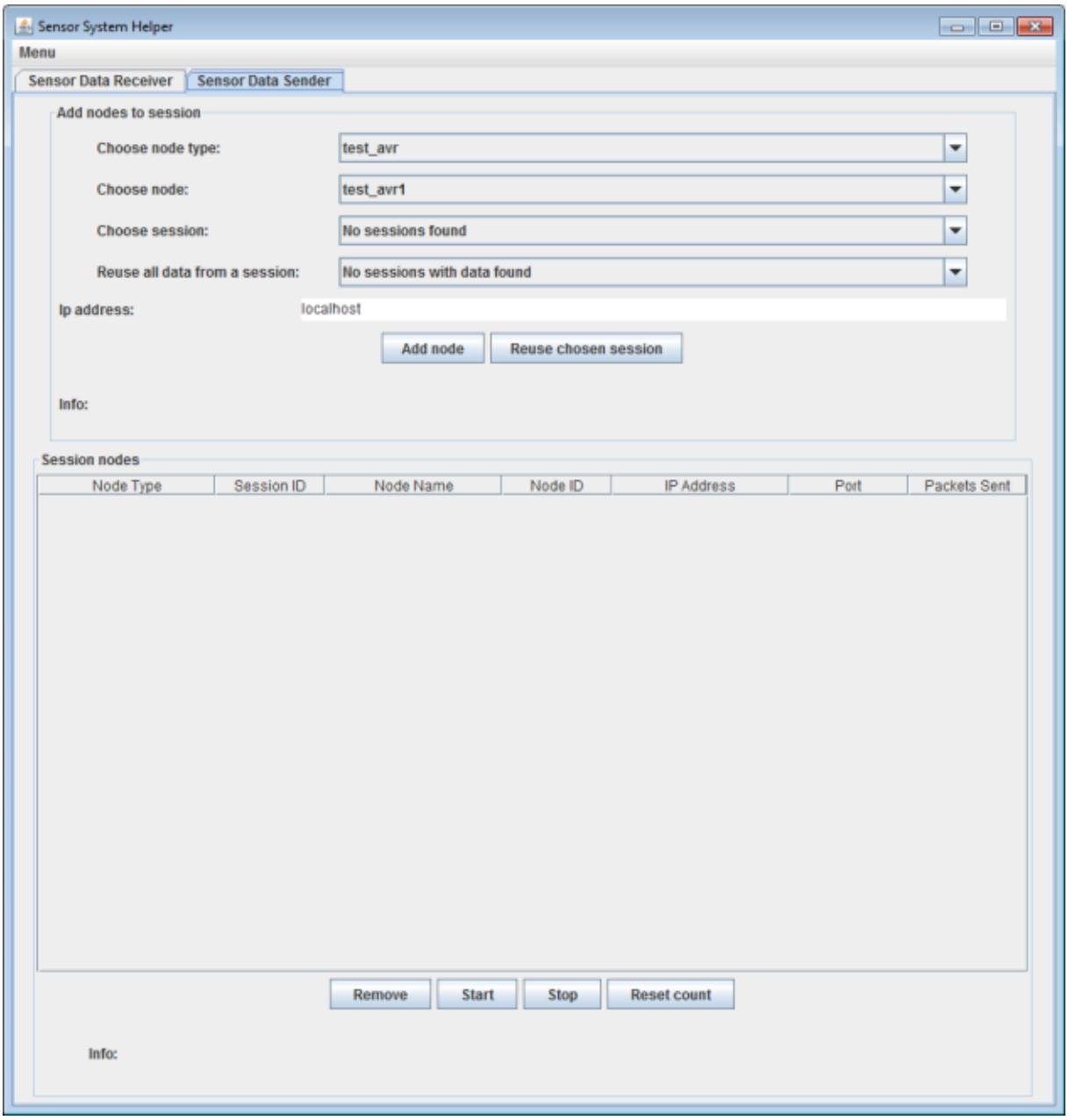


Figure 9: Data reuse view in GUI

Here, the user can either choose a specific session for the existing nodes, or reuse data from all the nodes in a session. Again, there is a section displaying information about all the selected nodes, with editable IP address and port columns, and a column that shows the number of packets sent from each emulated node.

6. Evaluation

The implementation meets most of the requirements discussed in chapter 3. The solution can retrieve data from sensors nodes, allows to parse any format by permitting to implement custom parsers, and stores sensor data persistently in a database. The application allows custom creation of communication channels to handle a multitude of potential data retrieval protocols and methods. Because of inherent limitations of the concept of reusing data, the communication between sensor nodes and sensor-based applications can mostly go one way and we do not provide a solution for systems based on sensor nodes requiring complex two-way communication, e.g. for sending control messages to nodes such as remotely controlled mobile nodes. We do however propose a way to handle pull-nodes by stating that our application can pull such nodes at specific intervals and the user can implement an adapted policy to respond with the sensor data that are stored with timestamps near in time to when the client application requests data.

The application also provides a solution to reuse stored data, and can emulate the behaviour of sensor nodes by retransmitting sensor data packets similarly to how and when the sensor nodes sent the data. The data packets can be customized to any format by creating specific unparser classes. A simple and intuitive GUI is provided, and all components are loosely coupled to permit simple change or replacement of components.

In the following sections, we will evaluate how the implemented solution performs for these tasks. I will start by presenting how the solution is tested, followed by a section discussing results.

6.1 Evaluation method and setup

The application has two modes of operation, data storage mode and data reuse mode, and in both cases the application depends on external entities to perform its task. For the data storage mode, the application relies on external sensor nodes that generate and transmit sensor data. In data reuse mode, the application is dependent on another application that receives the reused sensor data packets. I have therefore implemented 2 additional systems that are used to replicate the behaviour of the external entities. The first is a simple sensor node simulator that can be used to simulate multiple sensor nodes that send data packets The other application is used to mimick a sensor-based application, but it only receives packets and displays packet counts for each channel and can display contents of the data received in a packet.

I will run the implemented application and the external entity on 2 different computers to assure that the network is used. For this purpose, “HP Compaq Elite 8100 CMT” desktop computers located at the premises of the department of informatics at the University of Oslo will be used, including Intel(R) Core(TM) i7 870 processors running 2.93 GHz and 8 GB of RAM memory. These are quite powerful computers, but I expect users of the implemented application to use similar machines. One of the test machines uses Red Hat Linux, the other Windows 7.

6.2 Evaluation using Opportunity-project data set

The opportunity-project made data sets publicly available for the Opportunity Challenge, see section 2.5. The data sets consist of files with lines of sensor data, separated by space-characters. The first column is the timestamp of the data (number of milliseconds since start), the rest of the columns contain numbers that are values from sensor measurements. The sensor values are read from 5 sensor nodes, each reading X, Y and Z values from an accelerometer, a gyroscope and a magnetic sensor. This yields 45 sensor values on each reading, and sensor are read approximately every 33 milliseconds, resulting in a frequency of 30 readings per second and 1350 sensor values each second.

The chosen sample file, S4-ADL1.dat, downloaded from [20] , has data for about 23 minutes. For practical reasons, i.e. to be able to run many tests in an acceptable time, the first 10089 lines of this file was used for testing, yielding over 5 minutes of readings.

The Opportunity sensor readings were modelled in the application as one sensor node with 45 sensors, as this was most practical for the sensor simulator application.

6.2.1 Data storage

First, data storage was tested. The sensor simulator was used to send data packets (datagrams) as a push-node using UDP, at the times recorded in the trace file, and the data packets contained 45 sensor values separated by a whitespace-character. The application used the generic UdpPushReceiverChannel class for receiving data, a GenericReceiver class, and a custom UdpPushOpportunityAllTestParser class to split the data and insert it into the database.

8 tests were run, 4 with both the sensor reuse application and the sensor simulator running on on one computer, 4 with the applications running on 2 separate machines (within the same network), to see if there were differences in the number of values stored.

The tests showed that all sent packages were received by the application, and in most cases with the same time difference as the trace file, i.e. 33 milliseconds. Each sensor value was stored with the timestamp at which the packet it was contained in was received. This means that if all packets were to be received, parsed and stored correctly, the number of different timestamps in that session would match the number of packets received. Yet, data shows that sensor values were stored for a number of timestamps slightly below the packet count. This indicates that some of the packets encountered problems upon parsing or storage. Figure 10 shows the number of different timestamps stored for the session compared with the amount of packets sent. These numbers should have been equal if all sensor values had been stored correctly.

Console output during the tests showed that some of the attempted sensor-value database insertions failed because of primary key-violations. As shown in section 5.2.3, the primary key of the sensor_value tables is the combination of a timestamp, a sensor ID and a session ID. This indicates that the reason the insertions failed was that several values for the same sensor in the session had the same timestamp. The packet count in the applications indicated that the correct number of packets were sent and received, so an explanation to the lacking sensor values may be that some packets arrived in bursts due to some congestion, and hence

had the same timestamp, but I did not manage to find the cause of this discrepency with certainty.

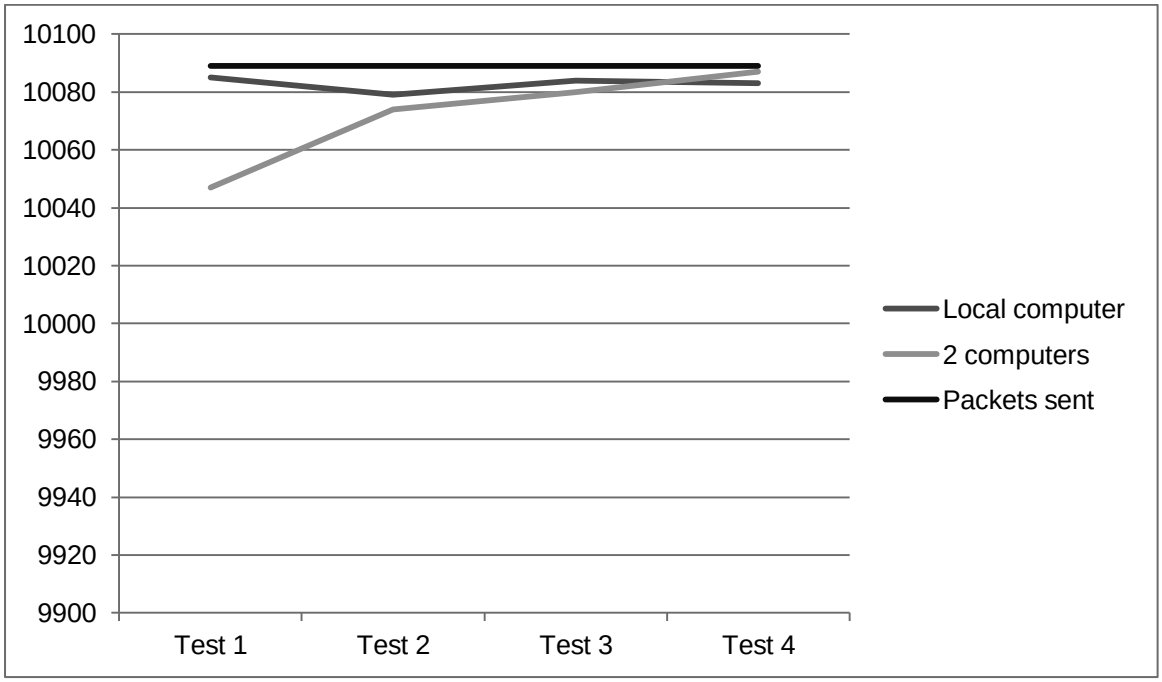


Figure 10: Number of timestamps versus packets sent

Despite some missed sensor values, with such a high-intensity data set we see that less than 0.5 % of the packets were not handled correctly, which is a decent result. This does however show that there is room for improvement. Arriving at similar results when running on one or two computers also could indicate that the cause of these missed values lays in the implemented software and not the network.

6.2.2 Data retrieval

For data retrieval tests, we used the stored sessions to reuse the data generated in the storage tests. The receiving application only showed the number of packets received and the content of packets as they were received. This does not allow us to verify that all sensor values are sent correctly with certainty, but rather gives an indication as to if it can handle the processing required to handle such a number of sensor values.

4 tests were run for this mode on one machine, and in all cases the number of received packets was the same as the number sent. Viewing the data received in the display view of the simulated sensor-based application indicated that correct data was being sent and that the packages were arriving as fast as they were sent.

6.2.3 Other observations

Using the large number of data in the Opportunity data sets had certain implications for the usage of the system. Since there are over 450 000 sensor values in the data set we used, this posed some great performance requirements to both the application and the underlying database. I discovered that after storing such an amount of data, some operations in the application caused to freeze for some time. Operating system process managers showed that during the freezes, the application process was using the CPU intensively. Viewing the stored data directly in the H2 Database Engine console (which runs in a browser) showed the same freeze and CPU-hogging. This could indicate that the chosen platform for persistent storage, i.e. H2 DB Engine, doesn't quite have the power to handle the amounts of data we wanted to use efficiently and fast in queries.

Other than this, the application seemed stable and I didn't encounter any specific problems using it besides the problem of some missed sensor values. My impression is that using the Opportunity data set presented the application with tough requirements with regards to processing efficiency and throughput, maybe tougher than for the purposes we may envision it being used for. Still, with some minor debugging and coding changes, it is my view that the application will be capable of handling such requirements without missing sensor values.

It must also be conceived that more testing could provide more information about potential problems with the application. We have only tested one data-intensive sensor node. Testing several sensor nodes sending data more sparsely and multimedia sensor nodes would have given a more complete view of the performance and validity of the implementation. Running more rigourous tests to verify that the sent packets really did coincide with the stored data would also present additional validation. Unfortunately, due to time constraints, this must be left for future work.

7. Conclusion

In this thesis, I have attempted to provide a thorough analysis of why we should and how we can store sensor data for reuse. Sensor and sensor node technology will be developed further and lead to an increasing number of applications for sensor-based event processing applications. After discussing the requirements that should to be met for a data reuse system, I designed and implemented a solution for this purpose. The result is versatile, extendable and developed for a variety of user platforms and applications. The evalution also showed that the implemented system is usable for its purposes and performs well. Some of the discussed features were not implemented, but the system can clearly provide a tool for developers and researchers working sensor-based applications.

The field of sensors, sensor nodes and sensor-based event processing applications is intriguing, and it is my hope that it will benefit a variety of domains in the future. I believe that my application can help bring progress to the field by simplifying the testing and thus the development process, and by giving a possibility to share sensor data in a simpler way. The usage of data sets from the Opportunity Project shows that this is at least a step on the way.

Appendix

A1 – Format of Opportunity Challenge data sets

Data columns:

Column: 1 MILLISEC

Column: 2 InertialMeasurementUnit BACK accX; value = round(original_value / 9.8 * 1000), unit = milli g

Column: 3 InertialMeasurementUnit BACK accY; value = round(original_value / 9.8 * 1000), unit = milli g

Column: 4 InertialMeasurementUnit BACK accZ; value = round(original_value / 9.8 * 1000), unit = milli g

Column: 5 InertialMeasurementUnit BACK gyroX; value = round(original_value * 1000), unit = unknown

Column: 6 InertialMeasurementUnit BACK gyroY; value = round(original_value * 1000), unit = unknown

Column: 7 InertialMeasurementUnit BACK gyroZ; value = round(original_value * 1000), unit = unknown

Column: 8 InertialMeasurementUnit BACK magneticX; value = round(original_value * 1000), unit = unknown

Column: 9 InertialMeasurementUnit BACK magneticY; value = round(original_value * 1000), unit = unknown

Column: 10 InertialMeasurementUnit BACK magneticZ; value = round(original_value * 1000), unit = unknown

Column: 11 InertialMeasurementUnit RUA accX; value = round(original_value / 9.8 * 1000), unit = milli g

Column: 12 InertialMeasurementUnit RUA accY; value = round(original_value / 9.8 * 1000), unit = milli g

Column: 13 InertialMeasurementUnit RUA accZ; value = round(original_value / 9.8 * 1000), unit = milli g

Column: 14 InertialMeasurementUnit RUA gyroX; value = round(original_value * 1000), unit = unknown

Column: 15 InertialMeasurementUnit RUA gyroY; value = round(original_value * 1000), unit = unknown

Column: 16 InertialMeasurementUnit RUA gyroZ; value = round(original_value * 1000), unit = unknown

Column: 17 InertialMeasurementUnit RUA magneticX; value = round(original_value * 1000), unit = unknown

Column: 18 InertialMeasurementUnit RUA magneticY; value = round(original_value * 1000), unit = unknown

Column: 19 InertialMeasurementUnit RUA magneticZ; value = round(original_value * 1000), unit = unknown

Column: 20 InertialMeasurementUnit RLA accX; value = round(original_value / 9.8 * 1000), unit = milli g

Column: 21 InertialMeasurementUnit RLA accY; value = round(original_value / 9.8 * 1000), unit = milli g

Column: 22 InertialMeasurementUnit RLA accZ; value = round(original_value / 9.8 * 1000), unit = milli g

Column: 23 InertialMeasurementUnit RLA gyroX; value = round(original_value * 1000), unit = unknown

Column: 24 InertialMeasurementUnit RLA gyroY; value = round(original_value * 1000), unit = unknown

Column: 25 InertialMeasurementUnit RLA gyroZ; value = round(original_value * 1000), unit = unknown

Column: 26 InertialMeasurementUnit RLA magneticX; value = round(original_value * 1000), unit = unknown

Column: 27 InertialMeasurementUnit RLA magneticY; value = round(original_value * 1000), unit = unknown

Column: 28 InertialMeasurementUnit RLA magneticZ; value = round(original_value * 1000), unit = unknown

Column: 29 InertialMeasurementUnit LUA accX; value = round(original_value / 9.8 * 1000), unit = milli g

Column: 30 InertialMeasurementUnit LUA accY; value = round(original_value / 9.8 * 1000), unit = milli g

Column: 31 InertialMeasurementUnit LUA accZ; value = round(original_value / 9.8 * 1000), unit = milli g

Column: 32 InertialMeasurementUnit LUA gyroX; value = round(original_value * 1000), unit = unknown

Column: 33 InertialMeasurementUnit LUA gyroY; value = round(original_value * 1000), unit = unknown

Column: 34 InertialMeasurementUnit LUA gyroZ; value = round(original_value * 1000), unit = unknown

Column: 35 InertialMeasurementUnit LUA magneticX; value = round(original_value * 1000), unit = unknown

Column: 36 InertialMeasurementUnit LUA magneticY; value = round(original_value * 1000), unit = unknown

Column: 37 InertialMeasurementUnit LUA magneticZ; value = round(original_value * 1000), unit = unknown

Column: 38 InertialMeasurementUnit LLA accX; value = round(original_value / 9.8 * 1000), unit = milli g

Column: 39 InertialMeasurementUnit LLA accY; value = round(original_value / 9.8 * 1000), unit = milli g

Column: 40 InertialMeasurementUnit LLA accZ; value = round(original_value / 9.8 * 1000), unit = milli g

Column: 41 InertialMeasurementUnit LLA gyroX; value = round(original_value * 1000), unit = unknown

Column: 42 InertialMeasurementUnit LLA gyroY; value = round(original_value * 1000), unit = unknown

Column: 43 InertialMeasurementUnit LLA gyroZ; value = round(original_value * 1000), unit = unknown

Column: 44 InertialMeasurementUnit LLA magneticX; value = round(original_value * 1000), unit = unknown

Column: 45 InertialMeasurementUnit LLA magneticY; value = round(original_value * 1000), unit = unknown

Column: 46 InertialMeasurementUnit LLA magneticZ; value = round(original_value * 1000), unit = unknown

Label columns:

Column: 47 Locomotion

Column: 48 Gesture

A2 - Source code

The source code can be found in the subversion repository of UiO at [23] .

Bibliography

[1] "Google Galaxy Nexus," [Online]. Available: http://www.google.com/nexus/img/content/gallery/04_gallery.png. [Accessed 20 01 2012].

[2] "Small Businees Trends," [Online]. Available: http://smallbiztrends.com/2011/02/qr-codes-barcodes-rfid-difference.html. [Accessed 10 10 2011].

[3] "Nanokopter.at," [Online]. Available: http://www.nanokopter.at/tiki_v6/. [Accessed 20 01 2012].

[4] "Panasonic Shop," [Online]. Available: http://shop.panasonic.com/dotAsset/77ecc45b-5c1a-4782-bb90-16a2d95fae8f.jpg. [Accessed 25 11 2010].

[5] [Online]. Available: http://www.eecs.harvard.edu/~konrad/projects/motetrack/manual/figs/micaz.jpg. [Accessed 15 01 2012].

[6] L. Gavrilovska, S. Krco, V. Milutinovic, I. Stojmenovic, R. Trobec and G. Rakočević, Application and Multidisciplinary Aspects of Wireless Sensor Networks, Springer, 2011.

[7] I. F. Akyildiz, T. Melodia and K. R. Chowdhury, "A survey on wireless multimedia sensor networks," *IEEE Wireless Communications Magazine*, vol. 14, no. 06, p. 3239, December 2007.

[8] "Tinynode," [Online]. Available: http://www.tinynode.com/?q=product/tinynode584/tn-584-868. [Accessed 15 09 2011].

[9] Hill, Jason; Horton, Mike; Kling, Ralph; Krishnamurthy, Lakshman, "The platforms enabling wireless sensor networks," *Communications of the ACM*, pp. 41-46, 6 June 2004.

[10] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam and E. Cayirci, "A survey on sensor networks," *Communications Magazine, IEEE*, vol. 40, no. 8, pp. 102-114, 2002.

[11] C. S. Raghavendra, K. M. Sivalingam and T. Znati, Wireless sensor networks, Springer, 2004.

[12] I. F. Akyildiz and M. C. Vuran, Wireless Sensor Networks - Ian F. Akyildiz Series in Communications and Networking - Advanced Texts in Communications and Networking, John Wiley & Sons, 2010.

[13] H. Kopets, "Internet of Things," in *Real-Time Systems - Design Principles for Distributed Embedded Applications*, Springer, 2011, pp. 307-323.

[14] I. F. Akyildiz and I. H. Kasimoglu, "Wireless sensor and actor networks: research challenges," *Ad Hoc Networks*, vol. 2, no. 4, pp. 351-367, 2004.

[15] "The Contiki OS," [Online]. Available: http://www.contiki-os.org/.

[16] C. Gomez and J. Paradells, "Wireless home automation networks: A survey of architectures and technologies," *Communications Magazine, IEEE* , vol. 48, no. 6, pp. 92-101, June 2010.

[17] A. Silberstein, "Push and Pull in Sensor Network Query Processing," 2006.

[18] A. V. U. Phani Kumar, A. M. Reddy V and D. Janakiram, "Distributed collaboration for event detection in wireless sensor networks," in *MPAC '05 Proceedings of the 3rd international workshop on Middleware for pervasive and ad-hoc computing*, 2005.

[19] J. Søberg, "CommonSens: A Multimodal Complex Event Processing System for Automated Home Care," Faculty of Mathematics and Natural Sciences, University of Oslo, 2011.

[20] "Opportunity Project - Activity and Context Recognition with Opportunistic Sensor Configurations," [Online]. Available: http://www.opportunity-project.eu/.

[21] J. Lewis and U. Neumann, "Performance of Java versus C++," 2003. [Online]. Available: http://scribblethink.org/Computer/javaCbenchmark.html.

[22] "H2 Database Engine," [Online]. Available: http://www.h2database.com/html/main.html.

[23] E. C. C. L. Hansen, "ellhanse-SensorSystemHelper," [Online]. Available: https://svn.ifi.uio.no/repos/users/ellhanse-SensorSystemHelper.