Modelling correlations for Body Sensor Network information

Pedro Brandão[*] [†]
Instituto de Telecomunicações, Faculdade Ciências,Universidade do Porto
R Campo Alegre 1021/1055, 4169-007 Porto, Portugal
pbrandao@dcc.fc.up.pt

ABSTRACT

Body Sensor Networks (BSNs) are the natural candidates to provide multi-parameter patient monitoring. Tapping into multiple inputs and correlating them to infer new information, *cleaning* received data and inferring state should be objectives of BSNs. These systems will need to deduce information from a variety of *raw* sensor data and accuracy in their results will be paramount. Apart from having several different types of sensors (producing different types of data), BSNs will also have several applications wanting to access information. Not all of the information will be directly sensed, but some can be inferred from the *raw* sensor data. We propose a framework that enables modularization of information and its correlation. This enables re-use by different applications and optimization of the collection and calculation of the requested information by the system (the BSN). The framework also allows defining dependencies between modules for information *production*. Our architecture provides an abstraction on the way information is assessed and its processing flow. Applications issue requests to the middleware with requirements to be met. So we will discuss the optimization of resources, while honouring requirements.

Categories and Subject Descriptors

H.1 [**Information Systems Applications**]: Models and Principles; D.2.11 [**Software Engineering**]: Software Architecture—*data abstraction*; C.2.11 [**Computer - Communication Networks**]: Distributed Systems

1. INTRODUCTION

The concept of multi-parameter patient monitoring is regaining *momentum* [1], as the possibilities for enabling it are getting more mobile, user-friendly and ubiquitous. The research for digital computerized systems that are able to assess different inputs can be tracked back a few decades. The experiment by Shubin and Weil in 1965/6 of using a computer to monitor and produce records for seriously ill patients (in a shock unit) [2] is one such example. Using several different inputs to assess physiological parameters has been a growing effort: devices that monitor several parameters [3]; correlating data to *clean* information [4]; deriving new state based on qualitative models [5] and enabling formula-based calculations of physiological parameters [6].

BSNs are a very good (if not perfect) fit for multi-parameter systems. BSNs need to be able to tap into several sensory information sources and correlate the information from them. Applications can then use the sources more efficiently, with more confidence, accessing more information than that provided by raw data from the sensors. This is the ulterior goal of our work.

We follow a middleware approach. The aim is (as expected in middleware architectures [7]) to provide applications with an abstraction of the underlying complexity; be it hardware (what type of node, what temperature sensor, etc.), Operating System (OS) (TinyOS, SunSpot Squawk, etc.), communication layers (ZigBee/802.15.4, Bluetooth, etc.), Service Discovery (SD) (Bluetooth, ZigBee, SensorML, etc.).

In our case, adding to the previous responsibilities we state that the middleware should: **A)** collect data from sensor nodes; **B)** convert these data to relevant information in a human body model; **C)** collect metadata on the data received and correlate it with the information in the model; **D)** answer requests from applications based on the information in the model while providing the related metadata; **E)** optimize resource usage (turn on/off, increase/decrease frequencies of collection, etc) while complying to requirements set by the applications.

For this an information abstraction layer is also needed. In this layer a model of information correlation would describe how to derive information from other information. A information flow framework would direct the information to the components that need it, either to derive new information or consume it.

Figure 1 illustrates possible correlations of different types of information. In the diagram blue boxes represent producers of information; they use the information from the yellow circles to infer new information. As an example we can produce Cardiac Output (CO) from Heart Rate (HR) and Stroke Volume (SV), using the formula $CO = SV \times HR$ from [8]. Moreover, the figure shows that CO can also be produced by the *ImpCard Reader CO* (impedance cardiograph reader) or

[*]The work presented was done in the course of a PhD dissertation at the Computer Laboratory, University of Cambridge.
[†]This work was partially funded by the VitalResponder project CMU- PT/CPS/0046/2008

BODYNETS 2012, September 24-26, Oslo, Norway
Copyright © 2012 ICST
DOI 10.4108/icst.bodynets.2012.249972

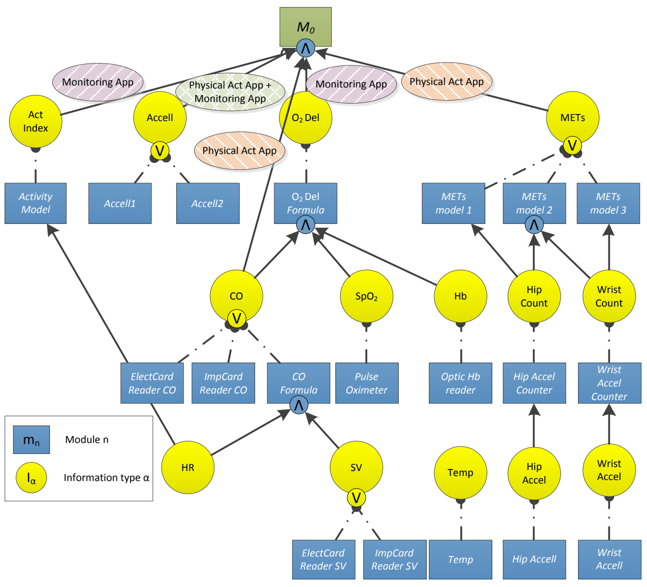


Figure 1: Correlation diagram example

ElectCard Reader CO (electrical cardiograph reader).

1.1 Problem statement

Our scenario involves a BSN based on a star topology centred on a more powerful node (a Base Station (BS)), as defined in IEEE 802.15.6's standard for Body Area Networks (BANs) [9][1]. In this scenario *several* applications running on the BS request information in order to function (providing it to the user, for monitoring purposes, etc.). In figure 1 ellipses represent applications requesting information. These needs may have requirements "attached", e.g. frequency of input, maximum error of the information, latency on getting it. There are costs associated with this retrieval, energy spent by a Hardware (HW) node, processing power, error introduced by the sensor or correlation, etc.

To answer applications requests, the middleware uses a model that defines how different inputs correlate to produce new information. There may be several ways to produce the same information. The objective is to allow applications to access not only raw information from the sensors, but also information combined or inferred from the "raw" sensor information. This means accessing any yellow circle. The middleware's objective is to optimize resource usage (the sensor system) while providing applications with the requested information. Our approach is to try to discover commonality between requests from the different applications. To improve the chances of finding it, we seek to have correlation **modules** in the middleware to enable "intersections" between requests. All requests' requirements must be met, so the most stringent of the intersecting requests is the one that must prevail.

From figure 1, if *MonitoringApp* requested *O_2 Del* with a frequency of update of 5 Hz and *Physical Act App* requested *CO* with a frequency of update of 10 Hz, the middleware should calculate *CO* with a frequency of 10 Hz. This would satisfy the strictest requirement.

We propose an approach to: **a)** compartmentalize the correlation between information in reusable software components; **b)** define how components depend on other components (information needed as input) using a graph analogy; **c)** optimize component usage by discovering intersecting needs; **d)** aggregate different requests so as to optimize according to a defined cost; **e)** define an infrastructure for the information flow using a Publish/Subscribe (pub/sub) architecture.

[1]The IEEE group also supports two multi-hp connections to the BS on the star. Our model works on a single hop connection, but this does not affect the work described herein. See also [10] for a discussion of BSNs network topologies.

2. RELATED WORK

To the best of our knowledge, there is no BSN middleware providing data correlation capabilities. They usually address communication to the sensors, such as [11] or processing data on the sensor such as [12]. They do not address information abstraction. As such we discuss two approaches for data correlation in BSN and then discuss middleware for Wireless Sensor Network (WSN) that tries to address correlation of different data.

Signal Interpretation and MONitoring (SIMON) [5] by Dawant et al. bases its work on the premise that "*alarms are too numerous to be correctly interpreted by humans, however highly trained*". To that end a system was built, capable of adapting its monitoring according to changes in the environment and on the patient's physiological state and its predicted evolution. The filtering, artifact removal and, especially, fault detection is also part of the framework's objectives. The project developed software components for a UNIX system based on Inter Process Communication (IPC) between modules for data acquisition from external medical devices. Data collection is based on either data acquired from sensors, where there is the possibility of acquiring data from several sources, or computed from different input types. These data can have associated thresholds and clinical medical ranges, which are user-friendly interval qualifiers. This is used as input to a model-based qualitative/quantitative reasoning framework. This model defines a hierarchy of objects that represent different levels of data abstraction, with defined boundaries, and artefact and fault models for data corruption detection. As such, the data abstraction component is used to: process and abstract the data; monitor/report according to the model; detect and react to possible errors in the data; adapt in real-time its behaviour according to context. The framework has a scheduler component, a modified earliest deadline first heuristic, to adjust tasks' frequency. Clearly we share objectives, but provide different functionalities, approaches and systems. We dwell in BSNs with the central component being the BS. We also aim to provide a hierarchical abstraction of the data, by defining software components that process their inputs, producing new information. Our added flexibility (the HW abstraction and information abstraction is more modular and transparent) is counterposed with the qualitative reasoning capability [13] of SIMON.

Fusion of multi-sensor data is used by Li et al. [4] to estimate HR. The idea is to use different sources for estimating HR, namely beat detection from ElectroCardioGram (ECG) and Arterial Blood Pressure (ABP) waveforms. The authors state that their main innovation is using signal quality assessments to characterize both estimates. They define how to assess signal quality of ECG and ABP and use this in a Kalman filter. This allows them to use the result of the Kalman innovation of the signal in a weighted fusion of both estimates. The objective is to use the better signal in the estimation, more precisely, to have a quality weighted "use" of the signals. This reduced artefacts and noise induced estimation errors in the resulting estimation.

The work from Heinzelman et al. in Middleware Linking Applications and Networks (MiLAN) [14], although related to WSNs, influenced what we propose. They address requirements of applications (that they call Quality of Service (QoS) demands), resource limitations and cooperative applications (i.e. different applications using the same network to achieve different objectives). MiLAN uses the applications' data requests with the requirements as inputs. Each sensor or group of sensors has a level of QoS that it can provide for each data requested, which is expressed in a sensor QoS graph. They define a State-based Variable Requirement Graph (SVRG), which determines the requirements, the quality needed in the measurement and what data to be sensed from the applications. This graph is based on the current application's state. The state influences the quality needed from sensed data. All the state is calculated by the application using the raw sensor data, although the SVRG indicates some relationship between the information. The intersection of the sensor QoS set and the SVRG defines the sensors to use. In MiLAN the notion of information correlation is left to the application, whereas in our approach the middleware knows the correlation model, allowing for reuse. The metrics defined are the quality of the values, network bandwidth and energy of the system. It is not mentioned how to re-use information for different states. Optimization is seen from a single application's point of view, however extrapolation for several applications could be achieved as an extra intersection of the different application solution sets. In our proposal we address optimization of several application requests from the start and derive intersecting requests/requirements.

In Semantic Streams [15], Whitehouse et al. use a model of "composable inference" for WSN data. They use a Constraint Language Programming (Real) (CLP(R)) framework to declare how different streams of data can be composed to produce new streams. Constraints can be defined for a query: confidence, relationship between streams (co-temporal, co-spatial). The "facts", for the streams of sensors, and "rules" for composing, including unit transformation, are defined on a Prolog engine. Optimization is supported for the defined metrics. When a query enters the system, the existing inference units can be composed to *generate new interpretations of sensor data*. It allows multiple applications, users in their nomenclature, to share the same resources on the network, while resolving conflicts. The easiness of expression, CLP(R) rules, mandates that a CLP(R) engine be available (SICStus Prolog in the paper's case) with *minutes or less* for a one time (per query) composition time. From the details on the article the proposed architecture does not seem to support composition of the same type of data streams ($\overline{Accell}$ from $\underline{Accell1}$ and $\underline{Accell2}$ in our example) so as to improve the quality/properties of a stream (e.g. minimize error). Composition rules are easy to express, an advantage of the logic programming language used, but modularity and the lack of different layers of abstraction may impair re-usability and flexibility. Moreover, processing time may be a deterrent.

In WSNs, declarative language approaches, as Cougar [16] and TinyDB [17], deal with query optimization using correlation between different sources of information in the network. The sensor network is seen as a distributed database. The reasoning follows from the highly correlated nature of WSNs, where values being sensed are usually of the same type and cross several nodes in their multi-hop trail to the sink. Sensor nodes have query proxies that are able to process queries and decrease power consumption in nodes on two avenues: aggregation of results and selective sensing. These approaches aim to optimize queries on correlated data as our proposal. The differences between BSNs and WSNs lead to different solutions. WSNs use several similar, if not identical, sensors to collect the same data types, which is of fundamental value for the declarative queries, specifically for aggregation purposes. Correlation is mostly done on the same data type. BSNs have different sensors collecting different types of data, making them less viable for these optimizations. This is accrued by the models being distributed through the sensor nodes, that either are all alike or will need to know about data types that they do not sense. The likely network star-topology for BSNs makes them single-hop, leading to fewer (if any) places to do aggregation. Correlation of different inputs is a driving point for our framework. The declarative language approach was aimed at other objectives and only with later work [18] was this accounted for. Another initial goal of our architecture is the possibility to optimize for multiple queries, so to allow several applications requesting different types of information. This is currently not done in the declarative language framework, as the optimization is done for a single query. Again, this may be a reflex of WSNs versus BSNs, the latter are more likely to have different applications running on top of its sensor network, as sensor information is more varied. The former, in most cases, has a single purpose and collects less varied information. Nonetheless, in TinyDB optimization of multiple queries is mentioned as future work.

3. MODEL

As mentioned, applications impose requirements for their requests. The middleware uses metrics to check if the requirements are being met. These metrics are also used for calculating the cost of using a specific configuration. This cost is the metric to optimize the system for. Examples of these metrics are frequency of updates, error, delay, energy consumption, processing.

The control that is available to the middleware is which modules to use (the decision variables) for the correlations. The end result is a configuration of modules to use to meet the requests while minimizing the cost.

From figure 1 the middleware would need to compute the modules to use so that *MonitoringApp* and *Physical Act App* would get the information that they are requesting, taking into account their requirements. *Monitoring App* needs activity index (a measure of a subject's activity), acceleration, ECG, and oxygen delivery (the quantity of oxygen that is transported to cells). The *Physical Act App* needs: acceleration, CO (measures the blood output of the heart) and Metabolic Equivalent (MET)[2].

3.1 Framework description

We represent information correlation in **modules**; they use **information** as an input to produce different information. In figure 1, blue boxes represent these software modules.

[2]MET is a measure to compare the metabolic expenditure of physical activities. It is a ratio to the resting activity, which has MET =1.

The yellow circles represent information. All of a module's inputs must be available for it to be able to produce its *unique* output. This is represented by the arrows leading to the boxes (to a $\wedge$ indicating the conjunction). The output produced by a module is represented by the "slash–dot" line directed to an information circle. The same information may be produced by different modules. This is represented by the lines leading to the information circle (to the $\vee$ indicating the disjunction). The diagram is a Directed Acyclic Graph (DAG) (no cycles are allowed by design) with two types of nodes. The direction is indicated by the arrows and the circles at the end of the "slash–dot" lines, which is bottom to top in figure 1.

We use the text notation of $\overline{overline}$ to represent information nodes and $\underline{underline}$ for module nodes. Although both information and modules are nodes in the graph we refer to *information nodes* and just *modules* from this point forward. When we want to address both we use *nodes.*

As an example, the $\overline{O_2 Del}$ information node can be produced by the $\underline{O_2 Del\ Formula}$ module. This module needs $\overline{CO}$, $\overline{SpO_2}$ and $\overline{Hb}$ to produce $\overline{O_2 Del}$. The top module, $\underline{M_0}$, represents the information requests made by the different applications to the middleware. As such it does not produce anything, but it needs all of its inputs, as it needs to answer all applications.

These correlations embedded in the modules are either defined by a particular application or globally accepted by the community, the latter improves reuse. In figure 1, METs calculation from activity counts models 1, 2 and 3 and are from [19]. The estimation of activity indices in $\underline{Activity\ Model}$ is from [20]. CO is derived from SV and $\overline{\text{HR}}$ using the Windkessel model described in [8]. The oxygen delivery model is discussed in [6].

As can be seen in the figure, some modules do not need any data input (e.g. $\underline{Accell1}$). These modules represent the sensors, which produce their outputs from the physical world.

Some other observations: **(a)** there is only one information node per information type in the model; **(b)** some modules and information present in the system might not be needed (e.g. $\underline{Temp}$) **(c)** the same physical sensor may have different modules associated with it (e.g. $\underline{Hip\ Accell}$ and $\underline{Accell1}$ are provided by an acceleration sensor on the hip); **(d)** different applications may request the same information with different requirements (for example $\overline{Accell}$) and different requirements may also occur for information that is commonly needed (e.g. $\overline{HR}$); **(e)** different modules producing the same output may have different requirements, cost and output quality (e.g. for the $\overline{METs}$ information there are three available modules with different accuracy on their outputs).

Application requirements mandate modules' requirements. A maximum error of 2% for the $\underline{O_2\ Del\ Formula}$ may impose a maximum error of 1% on $\underline{CO}$, 2% on $\underline{SpO_2}$ and 0.5% on $\underline{Hb}$. The influence of the inputs on the output of a module are defined by the module based on its correlation function.

3.2 Optimization algorithm

Information nodes can be produced by any capable module. Furthermore, information nodes can use more than one module for production in order to improve the metrics and meet the requirements. For example, $\underline{Accel1}$ and $\underline{Accel2}$ could be used for lower error or greater frequency. Any combination of one to all the modules is possible.

Some examples of metrics are frequency of output production, latency to output production, *Energy*, *CPU* and *IO* usage and *Error* associated with every node. We note that requirements (metrics) have inter-dependencies. Some of the metrics are merely cumulative (latency), but others may have a more complex influence on the nodes that are using the lower nodes (e.g. error propagation between input and output depends on the function utilized).

This complexity in metric correlation led us to look at modules as black boxes that can be queried regarding their performance (metrics and cost). Based on this we defined an algorithm to search all possibilities, discarding, as we search, the ones that do not meet the requirements. This is an improved "brute force" search, where we cut branches that do not satisfy requirements and use a cache for re-visited information nodes.

Our approach uses the model as portrayed in figure 1 searching from top to bottom. Part of the algorithm is defined in algorithm 1 where every set of possibilities of each node's descendants is combined with all others. Combinations depend on the node type. For modules, as every input is needed, we have a Cartesian product of the possibilities. For information nodes we can have any combination of the modules producing the information, ranging from a single one to all of them, i.e. $\bigcup_{i=1}^{\#desc} \binom{descOf(infoNode)}{i}$, where *desc* is descendants. Each of these combinations is then tested for requirements. If it satisfies the requirements the possibility is added to the set of possible solutions and cached for re-use.

Algorithm 1 CHECKPOSSIBLESOURCES(possListOfSets, node, requirements)

```
possComboSetOfSets ←
        COMBINEPOSSIBILITIES(possListOfSets, typeOf(node))
possForNodeSet ← {}
for each possComboSet in possComboSetOfSets
    (newPossibilityId, newMetric, newCost,
        nodesOfPossibleTree) ←
        AGGREGATENODES(possComboSet, node)
    if SATISFIES(newMetric, requirements) then
        possibility ← (newPossibilityId, node,
            newMetric, newCost, nodesOfPossibleTree)
        ADDTOTABLE(possibility)
        ADDTO(possForNodeSet, possibility)
return possForNodeSet
```

From a worst-case analysis [21] we have that the limiting part of our optimization is algorithm 1. Taking the number of modules (M) in the system as the input, without considering how they correlate, we have a $O\left(M^2 \cdot 2^M\right)$.

It is easily seen that this is a very bad worst case scenario, however it is very unlikely to happen, if at all possible. It would imply models where information nodes would be produced by M modules, leading to 2^M possible combinations for producing the information. Note that the algorithm discards possibilities that do not meet the requirements. However, in the worst case analysis all possibilities meet the requirements.

Worst case space analysis leads us to a similar pessimistic result of $O(M \cdot 2^M)$. In this case the cache table is the *hogging* factor where we have 2^M possibilities (thus table entries) each having M modules as data (which in a real scenario would not happen for every entry).

As an anecdotal example of performance from a model similar to figure 1, with 22 modules, we have 282 operation calls performed to find 135 possibilities (worst case with $M = 22$ is $22^2 \times 2^{22} = 2030043136$). This took an average of 0.4 ms in a laptop with an Intel 2.66 GHz CPU and 4 GB of memory, using an implementation in Java[3].

4. INFORMATION FLOW

The model pictured in figure 1 lends itself to a pub/sub architecture [22] so as to disseminate the information through the different modules. Modules subscribe to a type of event and publish in the system the information they produce. To accomplish this the middleware has a specific component that acts as the broker (`RegistrarProduction`) and a `Policy` component is responsible for finding the best module configuration (i.e. the set of modules that will produce the information requested while satisfying requirements) for the current applications' requests, using the model provided. A HW abstraction layer enables requesting and receiving the "raw" sensor data. As such our pub/sub system has: **a)** software modules that correlate information by publishing their outputs and subscribing to their required inputs; **b)** a framework to allow optimization of resource usage, taking into account modules' "costs"; **c)** different types of modules providing different functionalities.

4.1 Modules

Modules are the components that process information. They embed the principles for correlating inputs to produce a specific output. They may use several inputs, but only produce one output. Module production is published in the central `RegistrarProduction` that sends it to the modules that subscribed to that information.

This structure enables the composition of results by "chaining" modules together (connecting their outputs with inputs) to produce complex calculations/dependencies between data. As expected, outputs can be sent to different modules (multicast), enabling different interpretations of the same produced value in different contexts. Associated with the data produced, there are metadata fields that describe the accuracy and the time of production. These can be used by modules or applications for more detail about the data.

A module has some capabilities and a cost that describes how it is capable of producing a specific output. Modules can (and should) be implemented by the application developer if a specific correlation or functionality is not present.

Figure 2 illustrates the components of the information flow system on the BS, where some modules with different responsibilities are pictured. The HW abstraction layer is the middleware layer that hides the specificities of the HW and communications being used.

A `ModuleSensor` represents a sensor from the BSN. New sensors are discovered through a SD protocol [23]. This notification reaches the `DaemonGWRegistrar` that creates as many sensor modules as the sensors the node advertised. Data structures are used to keep the node and the sensor characteristics in the middleware on the BS.

The `ModuleSensor` maps the requests made to it to commands for the sensor. When the sensor produces data it flows to the BS using the specific communication system. A component in the HW abstraction layer de-multiplexes the messages received by the network, in this case a message with a measurement. This is sent to the `MeasurementTo-`

[3]Confidence interval of 95% leads to a top value of 3.8 ms.

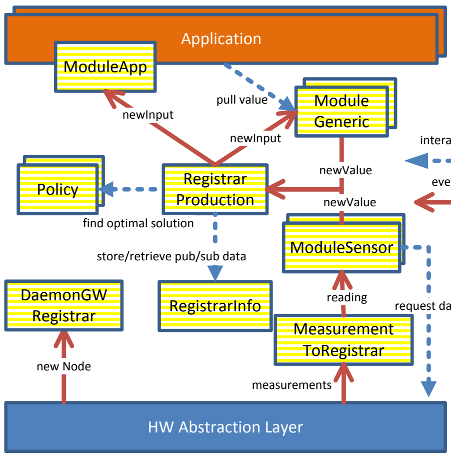


Figure 2: Information layer data flow (on the BS)

`Registrar` component that directs it to the module that is responsible for this sensor. The module then publishes the data on the `RegistrarProduction`.

The `ModuleGeneric` is a generic module that encompasses the functionalities and state that all modules have. As such it can publish new values and receive new inputs with information it subscribed to. Some other modules are already implemented to ease the development/usage of the architecture: `ModuleFormula` for formula-based calculations (e.g. for implementing $O_2DelFormula$), `ModuleStorage` for storing produced information and `ModuleAlarm` that checks the value of the information subscribed to and sees if it is within some specified limits. The `ModuleApplication` is used by the application to interact with the architecture, subscribing to the information requested.

4.2 Brokerage

The `RegistrarProduction` component from figure 2 is the central component that controls the subscription, production and optimization. The component needs to know about producers, which thus need to register, and the subscribers to those values, which need to subscribe. This enables one of its main tasks: distributing information.

These two lists of producers and subscribers are managed in the `RegistrarInfo`. Another list stored is the pending requests list, which holds the requests that could not be fulfilled due to a producer not being available for a needed `DataValue` in the production tree. `RegistrarInfo` also holds the current known policies for using the resources.

The `Policy` component is responsible for choosing which modules are used to satisfy the applications' requests while optimizing for a *defined* cost. The `Policy` is an *abstract* class that defines the interface that implementations of a policy need to conform to. This is so as to give flexibility of defining different policies that optimize to different costs. The default algorithm to use is the one mentioned in §3, but it is possible to define a policy that uses a different one.

5. COMPONENT INTERACTIONS

This section describes some of the interactions between components that are supported by the architecture.

Requests are done by modules that need information to function: application modules for the application, other

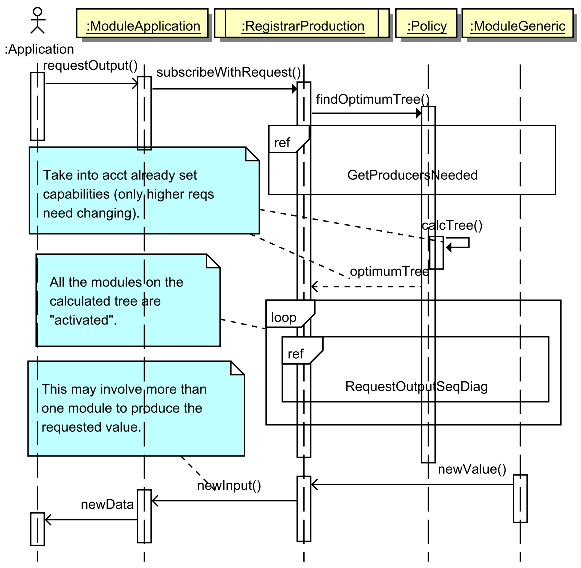


Figure 3: Request push value – subscribe (using an application as example)

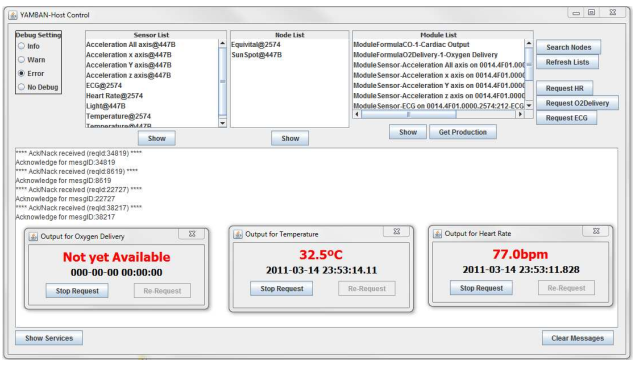
modules for storage, calculations, etc. There are two main ways of receiving data: pull and push.

As expected, *pull* is a one-time query that is made as needed. The requesting component may need to get a list of producers from which to request the information it needs. The pull can lead to a push based request if the value is not available; e.g. the producer module does not have the inputs needed to produce, or has not yet received them. The pull request can also have a minimum "freshness", where the requester can restrict the "age" of the information. This type of request is mainly used for getting information that is already being produced for a *push* request, as it does not entail any of the "information flow building" process.

The "push" request implies a subscription of the type of information requested and can have requirements associated. As illustrated in figure 3, this request leads to a chain of requests to an *optimum* calculated module tree. This tree is calculated by the `Policy` component so as to produce the requested information. The `RequestOutputSeqDiag` checks the modules to verify they are able to produce the requested values. It also encompasses starting production, if not yet started, or changing the settings if this request is stricter than the currently served one. Subscription to the needed inputs is also done by the module at this point.

Producer un-registering: a producer may decide to stop producing due to change in sensing behaviour, detected malfunctioning, battery draining[4], etc. The module should inform the `RegistrarProduction` of this action. This leads to a re-assessment of the solution used. The modules that were currently subscribing to the information being produced by the stopping module are found and the strictest request that each was serving assessed. After this, a search for another producer module (or combinations of modules) for substitution is done. If it is not possible to find a replacement module, the requests are all re-done as if they were new requests. This re-request is done after an un-subscription to

[4]This function could be implemented in a policy and lead to a re-assessment of the modules to be used.



Debug Setting: Error

Sensor List:
Acceleration All axis@447B
Acceleration x axis@447B
Acceleration Y axis@447B
Acceleration z axis@447B
ECG@2574
Heart Rate@2574
Light@447B
Temperature@2574
Temperature@447B

Node List:
Equivital@2574
SunSpot@447B

Module List:
ModuleFormulaCO-1-Cardiac Output
ModuleFormulaO2Delivery-1-Oxygen Delivery
ModuleSensor-Acceleration All axis on 0014.4F01.000
ModuleSensor-Acceleration x axis on 0014.4F01.0000
ModuleSensor-Acceleration Y axis on 0014.4F01.0000
ModuleSensor-Acceleration z axis on 0014.4F01.0000
ModuleSensor-ECG on 0014.4F01.0000.2574:212-ECG

```
**** Ack/Nack received (reqId:34819) ****
Acknowledge for mesgID:34819
**** Ack/Nack received (reqId:8619) ****
Acknowledge for mesgID:8619
**** Ack/Nack received (reqId:22727) ****
Acknowledge for mesgID:22727
**** Ack/Nack received (reqId:38217) ****
Acknowledge for mesgID:38217
```

Output for Oxygen Delivery
Not yet Available
000-00-00 00:00:00

Output for Temperature
32.5ºC
2011-03-14 23:53:14.11

Output for Heart Rate
77.0bpm
2011-03-14 23:53:11.828

Figure 4: Prototype Screen-shot

the previous requests, so as to clear the affected requests.

Producer unavailable: when the `Policy` tries to find the modules' tree to fulfil a request it may be unable to find a producer of a needed `DataValue`. In this case the requester should be notified that the request was unsuccessful. The request may additionally enter a "pending-requests" list, so that when a producer capable of producing the needed information registers, the request may be fulfilled.

6. IMPLEMENTATION

We implemented the architecture using SunSpots in a Java framework. SunSpots are general purpose sensors nodes developed by SUN that natively run Java code (a J2ME based version) on a Virtual Machine (VM) called *Squawk*. They have temperature and light sensors, a 3-axis accelerometer and connectors for adding further instruments (actuators or other sensors). We have developed the HW abstraction layer and the information abstraction using this platform. The information abstraction layer does not exist on the sensor nodes, as they only do the sensing (*dumb* components). We currently have the implementation for the BS running on a laptop (also done in Java).

We have developed a test application that requests O_2Del, CO, acceleration and ECG. These data are all faked by the SunSpot, except the acceleration. We have programmed the SunSpots to advertise themselves as other types of sensors able to sense O_2Sat, Hb and SV. These are profiles that are defined within the framework developed (see [24] for more details). The different requests are issued as if they were from different applications and the middleware combines them if possible. In figure 4 we can see the screen-shot of the prototype that runs on the BS. Here we can see the available sensors, nodes and modules. It is possible to check the metadata on these types and request the output production of any of the available nodes. This request will contact the needed modules so as to produce the output as described. In the figure there are also dialogues with the values that were being requested (0_2Del, HR and acceleration). Note that it is possible to request data that is currently unable to be obtained. The system will make the requests to the modules and start the production as soon as they are made available.

7. CONCLUSIONS AND OPEN ISSUES

In this paper we presented part of our framework to enable information distribution and correlation in BSNs. The framework's main purpose is to support a variety of applications and to ease their access to the data they need in order to operate, be it a raw sensor value or a derived value, while enabling optimization of resource usage. This correlation and calculation is done through the software modules described and the pub/sub system. The correlation of information, and thus the interrelation between modules, is described by an information model.

The architecture supports several applications running on a BS, requesting information (inferred or *raw*) with specific requirements. The model described compartmentalizes the information to provide opportunities for re-use and optimization of the BSN resources. We described an algorithm that searches for the optimal solution (for a defined cost) for all the requests from the applications within the given requirements. We analysed the worst case run time and space, which are $O(M^2 \cdot 2^M)$ and $O(M \cdot 2^M)$ respectively. In its defence the algorithm always finds the optimal solution (ignoring solutions that do not meet requirements and caching already found partial calculations) and the worst case conditions are rare, if at all possible.

Comparing requirements to find the stringent one, may be difficult when requirements encompass more than just one metric. This could be encapsulated in a middleware function. Further research is needed on this point.

We only gave as examples correlations that are very assertive, namely formula based relationships. It would be interesting to incorporate modules that produce *fuzzy* results. Based on several inputs the module would output a result with a probability of being correct. In this respect techniques as fuzzy logic and/or qualitative reasoning [13] are interesting to pursue.

8. REFERENCES

[1] Global Business Intelligence Research. Multiparameter Patient Monitoring Market to 2016, Dec 2010. PRLog news, Visited May 2012.

[2] Herbert Shubin and Max Harry Weil. Efficient Monitoring with a Digital Computer of Cardiovascular Function in Seriously Ill Patients. *Annals of Internal Medicine*, 65(3):453–460, 1966.

[3] U. Anliker, J.A. Ward, P. Lukowicz, G. Troster, F. Dolveck, M. Baer, F. Keita, E.B. Schenker, F. Catarsi, L. Coluccini, et al. AMON: a wearable multiparameter medical monitoring and alert system. *Information Technology in Biomedicine, IEEE Transactions on*, 8(4):415–427, December 2004.

[4] Q Li, R G Mark, and G D Clifford. Robust heart rate estimation from multiple asynchronous noisy sources using signal quality indices and a Kalman filter. *Physiological Measurement*, 29(1):15, 2008.

[5] B.M. Dawant, S. Uckun, E.J. Manders, and D.P. Lindstrom. The SIMON project: model-based signal acquisition, analysis, and interpretation in intelligent patient monitoring. *Engineering in Medicine and Biology Magazine, IEEE*, 12(4):82 –91, December 1993.

[6] R. Law and H. Bukwirwa. The physiology of oxygen delivery. *Update Anaesthesia*, 10:1–2, 1999.

[7] Philip A. Bernstein. Middleware: a model for distributed system services. *Commun. ACM*, 39, 1996.

[8] J.X. Sun, A.T. Reisner, M. Saeed, and R.G. Mark. Estimating cardiac output from arterial blood pressure waveforms: a critical evaluation using the MIMIC II database. In *Computers in Cardiology*, 2005.

[9] IEEE Standard for Local and metropolitan area networks – Part 15.6: Wireless Body Area Networks, February 2012.

[10] A. Natarajan, B. de Silva, Kok-Kiong Yap, and M. Motani. To hop or not to hop: Network architecture for body sensor networks. In *Sensor, Mesh and Ad Hoc Communications and Networks, SECON 6th Annual IEEE Communications Society Conference on*, 2009.

[11] Agustinus Borgy Waluyo, Wee-Soon Yeoh, Isaac Pek, Yihan Yong, and Xiang Chen. Mobisense: Mobile body sensor network for ambulatory monitoring. *ACM Trans. Embed. Comput. Syst.*, 10:13:1–13:30, August 2010.

[12] G. Fortino, A. Guerrieri, F.L. Bellifemine, and R. Giannantonio. SPINE2: developing BSN applications on heterogeneous sensor nodes. In *Industrial Embedded Systems, 2009. SIES '09. IEEE International Symposium on*, pages 128 –131, 2009.

[13] K.D. Forbus. *Intelligent Systems: Qualitative reasoning*, chapter VII. Computer Science Handbook. CRC Press, 2nd edition, 2004.

[14] W. B Heinzelman, A. L Murphy, H. S Carvalho, and M. A Perillo. Middleware to support sensor network applications. *Network, IEEE*, 18:6–14, February 2004.

[15] Kamin Whitehouse, Feng Zhao, and Jie Liu. Semantic streams: A framework for composable semantic interpretation of sensor data. In Kay Römer, Holger Karl, and Friedemann Mattern, editors, *Wireless Sensor Networks*, volume 3868 of *LNCS*. Springer, 2006.

[16] Yong Yao and Johannes Gehrke. The cougar approach to in-network query processing in sensor networks. *SIGMOD Rec.*, 31:9–18, September 2002.

[17] Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30:122–173, March 2005.

[18] Prithviraj Sen and A. Deshpande. Representing and querying correlated tuples in probabilistic databases. In *Data Engineering, ICDE IEEE 23rd International Conference on*, pages 596 –605, April 2007.

[19] A.N.N.M. Swartz, S.J. Strath, D.R. Bassett, W.L. O'Brien, G.A. King, and B.E. Ainsworth. Estimation of energy expenditure using CSA accelerometers at hip and wrist sites. *Medicine & Science in Sports & Exercise*, 32(9):S450–S456, 2000.

[20] George B. Moody. ECG-based Indices of Physical Activity. *Computers in Cardiology*, 19:403–406, 1992.

[21] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009.

[22] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35:114–131, June 2003.

[23] Pedro Brandão and Jean Bacon. Body Sensor Networks: Can we use them? In *M-MPAC - International Workshop.* ACM Digital Library, 2009.

[24] Pedro Brandão and Jean Bacon. BSN Middleware: Abstracting Resources to Human Models. In *HealthInf, International Conference on Health Informatics*, 2009.