Expressing and Configuring Quality of Data in Multi-purpose Wireless Sensor Networks

Pedro Javier del Cid, Daniel Hughes[1], Sam Michiels and Wouter Joosen

IBBT - DistriNet, Katholieke Universiteit Leuven,
3001 Leuven, Belgium
{name.lastname}@cs.kuleuven.be
[1]Department of Computer Science and Software Engineering,
Xi'an Jiaotong-Liverpool University, 215123 Suzhou, China
daniel.hughes@xjtlu.edu.cn

Abstract. Wireless Sensor Networks (WSNs) are evolving towards interconnected, sensing, processing and actuating infrastructures that are expected to provide services for multiple concurrent applications. In a multi-purpose WSN, concurrently running applications share network resources and each may have varying Quality of Data (QoD) requirements. Our middleware targets these multi-purpose WSN deployments. Specifically this paper discusses how one should express and configure QoD properties for multi-purpose WSNs. We contribute by presenting our approach; which leverages per-instance QoD configuration and a separation of operational concerns to achieve simpler configuration and improve adaptability and customize-ability of the WSN. A prototype implementation and comparison to the related state of the art in WSNs are provided.

Keywords: Wireless Sensor Networks, Resource Management, Middleware, Adaptive, Context Aware, Quality of Data

1 Introduction

Wireless sensor networks (WSNs) support the integration of environmental data into applications, from mobile devices to backend enterprise infrastructure. WSNs are evolving towards interconnected, sensing, processing and actuating infrastructures that are expected to provide services for multiple concurrent clients [1]. In a multi-purpose WSN, applications share network resources and each may have varying Quality of Data (QoD) requirements. Notably, QoD requirements, i.e. the required data reliability, resolution and the importance of a single reading, vary from application to application [2].

Previous approaches address the challenge of multiple applications on shared infrastructure by decoupling of the applications and the network, e.g. Milan [3], Servilla [4], TinySOA [1] and TinyCOPS [5]. In early approaches, such as Milan, QoD is expressed at the application level i.e. once for every application, it is assumed that: (i) there is no run-time variability in required QoD, (ii) considerable collaboration between applications is possible and (iii) a-priori knowledge of all applications that will use the network is available. In multi-purpose WSNs these assumptions are not reasonable thus application level configuration of QoD is not an adequate approach. Approaches such as TinySOA [1], Servilla [4] and TinyCOPS [5] configure QoD in a per-instance manner. An instance refers to each service request or query an application may have, where an application may have multiple concurrent requests or queries.

Per-instance configuration allows for higher flexibility and optimization. These multi-purpose WSNs may serve different types of applications with arbitrary requests or query patterns with no a-priori knowledge needed. They provide application developers the flexibility to meet variable QoD requirements of new applications and yet expect the same levels of performance that would result from an application-specific deployment [1]. Fine-grained optimization is possible because every instance may be customized with specific QoD requirements allowing for higher component re-usability, more efficient parameterization and improved reliability through lightweight run-time capacity planning [6].

We present our approach that leverages per-instance QoD configuration and a separation of operational concerns to achieve simpler configuration and improve adaptability and customize-ability of the WSN. A*daptability* refers to the system's capacity to enact context-aware run-time reconfiguration. *Customize-ability* refers to the capacity the system has to fine-tune or extend its functionality at run-time without service interruption.

This paper is structured as follows: Section 2 elaborates on the operational setting of our research. Section 3 presents an overview of our middleware. Section 4 presents our prototype implementation. Section 5 further describes WSN configuration and highlights our achieved benefits. Section 6 discusses these benefits in the context of the current state of the art. Section 7 concludes the paper and maps the road ahead.

2 Operational Setting

One of the main objectives of shared enterprise deployments is to maximize the return on investment in the WSN infrastructure [1,7,8]. We leverage on per-instance QoD configuration and focus on maximizing the amount of concurrent and varying QoD-aware requests the network can successfully support while providing simple configuration abstractions. Quality of Data (QoD) can be broken down into reliability and resolution [2]. The former specifies the accuracy of the data i.e. reported data corresponds to reported phenomena and the latter refers to the granularity of data and its temporal and spatial qualities. These are properties that are specified by the application and may vary for each request instance.

In multi-purpose WSNs the main operational concerns involved in application development and use should be undertaken by the following operational roles as defined by Huygens et al. in [9]: application developers, service developers and network administrators. Application developers (application owners in [9]) will be concerned with achieving high-level business goals and will undertake the implementation of domain specific business logic. Service developers (component developers in [9]) will be concerned with developing prepackaged functionality to support the goals of the network administrators and application developers. They will undertake the implementation of application-independent and platform-specific common use services e.g. temperature sensing on a SunSpot [10] sensor node. Network administrators (infrastructure owner in [9]) will be concerned with monitoring network QoS, configuring and maintaining common use software services e.g. temperature, aggregation. They also have high-level goals, usually system-wide requirements driven by concerns such as system lifetime optimization or service level agreements with application stakeholders.

Consider a WSN deployed in a corporate warehouse (see Fig. 1). The deployment is shared by multiple applications each with different QoD requirements. An HVAC application monitors environmental conditions to determine cooling and heating requirements and periodically gathers sensing information. A tracking application is used to provide information on package movement and environmental conditions during storage.

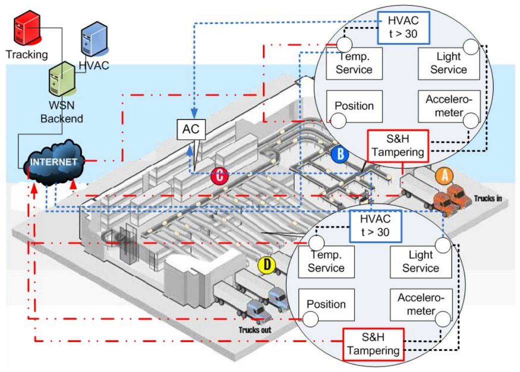


Fig. 1. Deployment scenario

The HVAC application periodically requests temperature and light readings throughout the warehouse and deploys specialized components to specific nodes that locally determine if an actuating action needs to be taken e.g. if temperature exceeds 30 degrees increase power to the AC unit in this area. The tracking application continuously monitors temperature and position of packages. Additionally a specialized component is deployed to high value packages which use light and accelerometer readings to locally determine package handling and tampering.

An application developer would be concerned with implementing the required functionality of each deployed application; that is HVAC and tracking, as well as the respective application-specific components. The implementation of the temperature, light, accelerometer services would be undertaken by the service developer but monitoring their run-time performance would be undertaken by the network administrator. The network administrator is in charge of monitoring the infrastructure and taking corrective action when needed to ensure expected quality levels are achieved. Current approaches do not properly separate these operational concerns and do not provide appropriate abstractions to achieve the required adaptability and customize-ability.

3 Middleware Overview

We propose a middleware approach based on configurable components that may be used in multiple concurrently running compositions and allow different QoD parameterizations for each composition. Figure 2 illustrates an overview of the different elements that compose our middleware. Through decoupling of applications and the components implementing the underlying application functionality, and the provision of structure and behavior patterns we achieve simple compositions and per-service instance parameterization. We consider a service instance to be: each service request from the moment it is submitted to the middleware until it has been processed. The service request specification is used to express the desired functionality and QoD for every service instance.

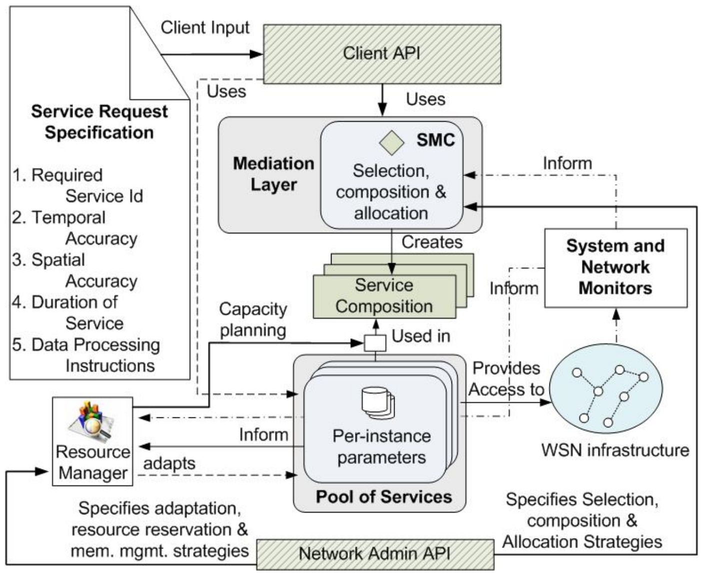


Fig. 2. Middleware overview

3.1 The Mediation Layer

The mediation layer runs in the backend and in cluster heads in the WSN. The mediation layer is implemented by the Service Management Component (SMC). It automatically interprets requests, selects the optimal service providers and instantiates an individual service composition involving specified services from a shared pool of components interacting in a loosely coupled manner. Every application may submit multiple service requests, each representing a service instance. As such, every composition allows for per-service instance parameterization of how this pool of components is used.

3.2 The Service Request Specification

Application developers use the service request specification to express their QoD requirements for each service instance. In the specification one includes the request Id, which is a unique sequential number generated by the WSN backend middleware. The service Id represents a globally unique service identifier defined at service implementation time. Each sensing service e.g. temperature, humidity, has a unique service Id. The temporal resolution required from the specified service is expressed through the sampling frequency. Duration of service is the amount of time one requires the selected sensing service to collect data samples. Spatial resolution is specified by selecting a target location e.g. <warehouse A> or < node21>. A Post-collection data processing service Id, which is globally unique identifier for services like averaging or specialized data filters. Finally a parameter to be passed to the post-collection data processing service may be required e.g. in case of the averaging component, one may use the parameter 30 to indicate the average must be done in 30 minute intervals. Each service request may be configured with different QoD requirements and it may or may not include one or more post-collection processing instructions.

```
  serviceRequest#(requestId,serviceId,samplingfrequency,duratio
n,targetLocation,DataProcessServiceId[],parameter[]);
```

3.3 WSN Services

As one may see in Fig. 3A, we typify services based on two primary types: sensing services and post-collection data processing services. These are the meta-types used to implement all services that comprise the pool of services available to create service compositions. They are implemented in Sensing Service Components (SSC) and Data Processing Component (DPC) respectively. Sensing services are components offering typical functionality such as the retrieval of temperature or light readings. They provide access to the various sensors. Data processing services are components implementing typical post-collection data processing functionality such as averaging, data filtering or persistence. A sample composition may be: (i) an SSC reads, timestamps and stores temperature readings which, (ii) an averaging component processes and finally (iii) a persistence component stores to static memory. SSC and DPC meta-types impose structure and behavior of implemented services. This predictable structure and behavior allows for higher re-usability of services in compositions because all DPCs can transparently be used with any SSC. SSCs are unaware of the existence of any other SSC or DPC. All SSCs provide and require the same interfaces. All DPCs provide and require the same interfaces.

Elaboration on how the imposed structure and behavior of components helps achieve more efficient reconfiguration and service composition can be found in our previous work [6]. Data reliability requirements may be achieved with the use of specialized data filters implemented with DPCs. For example, erratic sensor readings may be excluded from a sample on the basis that may indicate a defective sensor.

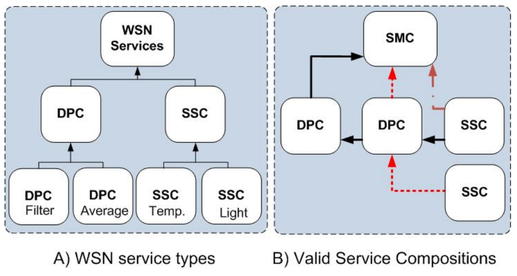


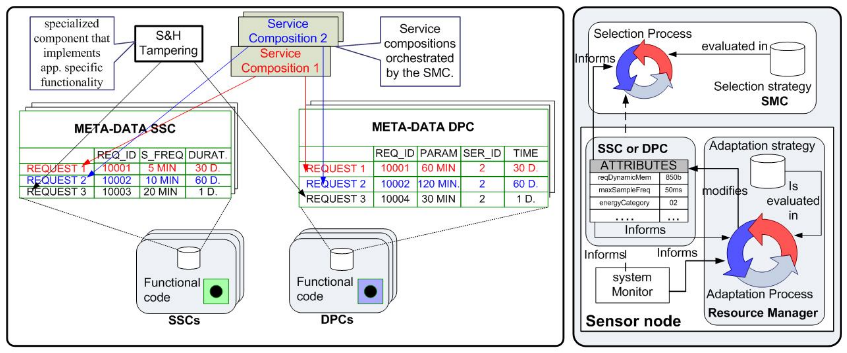
Fig. 3. WSN Services and valid service compositions

3.4 Service Compositions

The submission of a service request starts a service instance which is fulfilled with an independent service composition. Service compositions can have only 1 SSC and zero-to-many DPCs (see Fig. 3B). Dash-dotted lines depict a composition that only involves an SSC, the dashed lines depict a composition that uses one SSC and one DPC and the continuous lines depict a composition that involves an SSC and 2 DPCs. All components may be used in multiple compositions concurrently and have different QoD parameters for each composition. Due to the imposed structure and behavior, DPCs may be transparently combined with any SSC. Additionally new services can be introduced or additional requests supported and none of the existing compositions need to be modified or any service interrupted, resulting in efficient reconfiguration. Elaboration on how our simple composition rules help achieve more efficient service reconfiguration and lower complexity overheads can be found in our work [6].

3.5 Share-able and Adaptive Components

We consider that introducing a new component for every service requiring different parameterization is not efficient, as exemplified in [6]. We separate the functional code from the meta-data and share the same component instance across multiple service compositions (see Fig.4A). This meta-data contains the configuration semantics to be used to serve each service request. Each component is associated with a particular service composition through a request Id; this association contains per-instance configuration semantics. Configuration semantics for each service composition are extracted from the specified service request. The configuration semantics include specified QoD, services involved in each composition and related parameterization. The SMC parses the service request and extracts configuration parameters which it autonomously submits to the corresponding SSC or DPCs. Clients may also submit these configuration parameters directly to the SSC or DPC using ProcessRequest@SSC or ProcessData@DPC as one may see in Fig. 5A. This allows a single instance of our components to be used across multiple service compositions with varying parameters in each composition and avoids substantial increases in required static and dynamic memory per additional service request. For further details on how each SSC and DPC are configured we refer the reader to [6].



META-DATA SSC

	REQ_ID	S_FREQ	DURAT.
REQUEST 1	10001	5 MIN	30 D.
REQUEST 2	10002	10 MIN	60 D.
REQUEST 3	10003	20 MIN	1 D.

META-DATA DPC

	REQ_ID	PARAM	SER_ID	TIME
REQUEST 1	10001	60 MIN	2	30 D.
REQUEST 2	10002	120 MIN.	2	60 D.
REQUEST 3	10004	30 MIN	2	1 D.

ATTRIBUTES	
reqDynamicMem	850b
maxSampleFreq	50ms
energyCategory	02
....	...

A) Configuration semantics for components B) Component atts.& adaptation

Fig. 4. Configuration semantics, annotated attributes and adaptation in components

Furthermore these configuration parameters are available for introspection in SSCs and DPCs. Introspection provides insight as to currently allocated requests and their parameters. Additionally these inform of current component dependencies.

Annotated Component Attributes: SSCs and DPCs are annotated with attributes that are mandated as part of their structure. They may be static or dynamically modified at run-time depending on the attribute and intended use. For example: an energy category attribute is used to represent energy consumption incurred in the invocation of a particular sensor, given that energy use may vary considerably by platform / sensor hardware as exemplified in [11]. At implementation time the service developer would include the value for the energy category of the SSCs he implements and these are specific to the platform/sensor of deployment. These attributes provide semantic information which is used during the evaluation of system strategies, such as: adaptation and resource reservation. E.g. The result of the evaluation of an adaptation strategy may modify the maxSamplingFrequency attribute in an SSC modifying the execution of the component's functional code, in turn influencing the selection of service provider during the evaluation of the selection strategy in the service matching process of the SMC (see Fig.4C). Sykes et al. in [12] discuss how the use of quality attribute annotations in components improves decisions about adaptive reconfigurations. Using these attributes is how the service developers may express their concerns, regarding relevant usage parameters, provided accuracy or energy consumption of implemented services.

Component Level Adaptation: Component behavior may be modified at run-time i.e. the parameters used in the execution of its functional code may be dynamically modified as previously discussed. These adaptations give the network admin effective mechanisms that account for current working conditions and help prolong system lifetime. As exemplified in Levels [13] modifying available functionality has proven very efficient in prolonging system lifetime in sensor networks. The adaptation strategy specifies that data resolution is lowered more aggressively on services with higher energy categories when battery levels decrease, thus avoiding excessive use of high consumption services. This is done independently from any request being processed and transparently for application developers and service developers.

3.6 Capacity Planning

Each node has a light-weight Resource Manager (RM) that does run-time capacity planning and reservation of any resource required to effectively support allocated service requests. For further details on the benefits of run-time capacity planning we direct the reader to [6]. Currently we focus on required dynamic and static memory to support allocated requests. The RM uses a memory reservation strategy specified by the network administrator. This strategy specifies how much memory to reserve based on estimated requirements and when and how to release these reservations. This guarantees that every service request will have the needed resources e.g. memory, to be processed successfully through the service duration.

Calculating Required Memory: In order to effectively calculate the amount of static and dynamic memory that will be used by the middleware, the service developer must realize an off-line process to establish a memory baseline and an autonomic run-time process to establish run-time memory requirements. During the off-line process a baseline of memory use is recorded for every implemented service and the corresponding annotated attributes of each service are updated e.g. requiredStaticMemory, requiredDynamicMemory, memoryManagementOverheads.

The on-line process parses each submitted service request and extracts: (i) services involved in the composition (ii) QoD parameters. These are used to calculate the amount of memory needed to successfully process each request. For example: in the SSC the amount of required memory depends on the output dataset size which is directly proportional to the amount of records. These will vary depending on sampling frequency and service duration. Further details can be found in our previous work [6].

Memory Management: The memory management strategy dictates how and when data should be transferred to static memory. The strategy accounts for frequency of read and write operations. Additionally this strategy may specify memory related actions to increase reliability, e.g. make all sensed and processed data persistent when battery levels drop under 15%. Both these strategies are evaluated in the RM. Making data persistent in static memory is done by a persistence service DPC.

3.7 The Client API

Clients interact with the middleware through a distributed API (see Fig. 5A). We consider middleware clients to be applications, which are developed by application developers. The mediation layer is accessible to the clients at the back-end or in every cluster head and exposes interfaces A and B. The former interface is used to submit service requests and the latter to retrieve processed data. Clients may also access the middleware directly at the sensor nodes commonly through the use of application-specific components. This interaction happens through interfaces 1 and 2 available on SSCs and DPCs. The former is used to submit configuration parameters which are used by the component to parameterize each service instance and the latter is used to retrieve data.

```
ProcessRequest@SSCorDPC(requestId,samplingfrequency,duration);
ProcessData@SSCorDPC(requestId,serviceIdToCollectData,parameter,
timeToExecute);
```

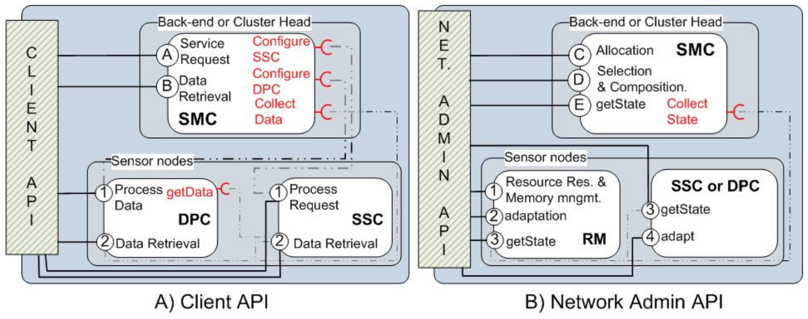


Fig. 5. Middleware component and interface views

This distributed API gives the application developer access to WSN services directly on each sensor node in a consistent manner. She is able to leverage in-network processing to achieve improved application performance while still undertaking only her concerns for the implementation of business functionality. She can develop application-specific components that take business reasoning into the network and react locally to improve reaction times while still being abstracted away from platform specific programming. For example: the HVAC components that locally determine if an actuating action needs to be taken, as described in section 2 (see Fig.1).

3.8 Network Administrator API

The network administrator interacts with the middleware through the API depicted in Fig. 5B. The SMC exposes interfaces C, D and E. Interface C is used to configure allocation strategies which are used by the SMC in the first step of the selection process, to narrow down possible providers to a potential sub-network or cluster. Additionally once the potential cluster is selected workload distribution or node tasking preferences are accounted for e.g. nodes at the edges of the network should sense, intermediate nodes should only transmit and process. Interface D is used to configure selection and composition strategies. Once the potential cluster of providers is selected, the selection strategy selects a node from within said cluster. This is done by comparing the current battery level and sampling frequency provided by each node. Once the provider is selected composition strategies are used to determine the proper order of composition according to the composition pattern expressed in the strategy. E.g. For a request that involves a temp, average and data filter services, based on a Sequence pattern [14]: the resulting composition would first have the temp service then the data filter and finally the averaging service. The sequence pattern (a→b) states that b is carried out after the completion of a. In [14] the authors further elaborate on the benefits of leveraging composition patterns to achieve significant benefits in the composition process.

Interface E is used to introspect the SMC e.g. service requests being processed and their respective parameters. The SMC may in turn use the introspection interfaces of relevant SSCs and DPCs to obtain all requests being process by each one and their respective annotated attributes. It is important to notice that this introspection provides the details on all current component dependencies and can be used to create a component graph.

On each node the RM exposes interfaces 1,2,3 and every SSC/DPC expose interface 3 and 4. In the RM interface 1 provides access to resource reservation and memory management strategies (see section 3.6). In the RM interface 2 provides access to adaptation strategies (see section 3.5). In the RM interface 3 provides introspection of current resource allocations e.g. current static and dynamic memory allocations with all relevant request related information.

In the SSC/DPC interface 3 provides access to requests being processed with all their respective configuration parameters and the annotated component attributes; which in essence describes all current component dependencies and when each dependency will expire. Expiration of a dependency depends on the duration of the service that required that dependency i.e. when the service terminates the dependency expires. This gives the admin valuable insight into current system configuration. Interface 4 provide access to the modification of annotated attributes hence potentially altering how the component's functional code is executed and the outcome of strategy evaluation as described in section 3.5.

4 Middleware Implementation

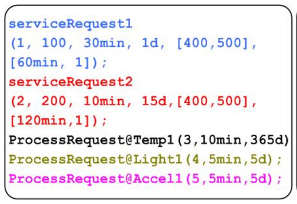
We implemented a prototype to validate our approach; it was implemented in Java ME CLDC1.1 configuration on the SunSPOT platform [10]. The implementation supports the operational scenario as described in section 2. To provide the reader with explicit background on our implementation we provide details of the evaluation conducted in the context of our previous work. This is described in more detail in [6] and is summarized below.

We recorded relevant footprint information and run-time memory consumption in our middleware while executing the following use case: The HVAC back-end application requires that for service request 1, temperature is to be sensed every 30 minutes during the next 1 day, averages of samples for every 60 min. should be processed and the results should be made persistent. The tracking application requires that for service request 2 light readings be taken every 10 min. for the next 15 days, 120 min averages are processed and the results be made persistent. These service requests are submitted to the SMC and are depicted in Fig. 6A (as is not directly relevant to our scenario, we omit the specification of target location).

The HVAC specialized component that determines AC related actions requires temperature readings every 10 min. for the next year. Since this specialized component is deployed on the node, it submits these parameters directly to the temperature SSC with the ProcessRequest@Temp1. The deployed tracking specialized component that determines product handling and tampering requires light and accelerometer readings every 5 min. for the next 5 days. This component submits these parameters directly to the corresponding SSC.

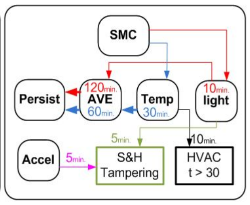
In Fig.6B one can see the component configuration implemented and the service composition that resulted from the submitted service requests and process requests. Five service components are used to serve 5 concurrent requests which are fulfilled with 5 independent service compositions. As the number of served requests increases the amount of instantiated components remains constant. Each SSC consumes 750 bytes of dynamic memory and 7.8 Kb of static memory. Each DPC consumes about 750 bytes of dynamic memory and 11.9 Kb of static memory. Each additional service request processed in a SSC consumes about 5Kb of dynamic memory. Each additional request that runs in a DPC consumes about 800 bytes of additional dynamic memory.

We then recorded footprint and run-time memory consumption for a comparable approach. We implemented the same use case with LooCI [15] component model. We selected this approach because of its very loosely coupled component interaction and published subscribe functionality provides effective mechanisms to implement multi-purpose WSNs. It was necessary to instantiate 9 LooCI micro-components. Each LooCI component requires 3 Kb of dynamic memory and 1.7Kb of static memory.

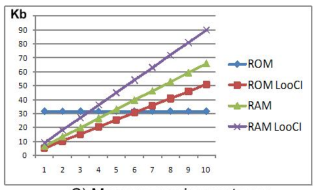


```
serviceRequest1
(1, 100, 30min, 1d, [400,500],
[60min, 1]);
serviceRequest2
(2, 200, 10min, 15d,[400,500],
[120min,1]);
ProcessRequest@Temp1(3,10min,365d)
ProcessRequest@Light1(4,5min,5d);
ProcessRequest@Accel1(5,5min,5d);
```

A) Service requests used for evaluation



B) Service Compositions



C) Memory requirements per additional service request processed.

Fig. 6. Implemented scenario

In Fig.6C we plotted the amount of dynamic memory (RAM) and static memory (ROM) required to instantiate the components needed to process concurrent requests with both approaches. We varied the amount of concurrent requests processed from n=1 to n=10. Each of these requests is equivalent in functionality as serviceRequest1.

In terms of transmission overhead*:* One service request (order of 64 bytes) is needed to support an additional request, where in LooCI two new components are needed (each component is in the order of 1.7Kb).

In terms of request load: The additional functionality offered in our components comes with some overhead in static memory which as one can see from Fig.6C is an acceptable trade-off given the improved efficiency under higher request loads.

5 Discussion

In this section we further discuss how a separation of operational concerns allows for simpler WSN configuration while improving adaptability and customize-ability of the middleware.

5.1 Simpler WSN Configuration

To elaborate on how the proposed middleware simplifies configuration we look at the manner in which the WSN is to be configured i.e. complexity of abstractions used and how operational concerns are accounted for. We consider that a clear separation of operational concerns allows for simpler system configuration. We elaborate on the benefits for network administrators, application and service developers below.
Application developers: In our system they only have to concern themselves with the implementation of business functionality. Our client API uses the service request specification to allow them to express their functional requirements only. They are not burdened with platform specific commands to implement sensing or processing services. They are not burdened with issues like selecting low consumption services to maximize network lifetime. The service request is easy to use and yet expressive enough to adequately address complex use cases as the one described in section 4. The service request specification does not require any coding or instantiation of run-time constructs. Furthermore, they can use application-specific components that implement extended business functionality and deploy them anywhere in the network and still access a simple and consistent API at node level.
Service developers: Service implementation may be done by following the specified structure and behavior imposed by the SSC and DPC meta-types. Annotated attributes allow them to enhance the implementation with semantic information regarding relevant usage parameters, provided accuracy or energy consumption. These attributes can be easily extended to account for new requirements. In this way the service developer is not burdened with operational considerations e.g. what should be the offered data resolution given a particular battery level. Service developers do not require any domain specific or business knowledge.

Network administrators: The network admin API gives them access to configure allocation, selection, composition strategies in the SMC, adaptation, resource reservation and memory management strategies in the RM. These strategies give the administrator the abstractions needed to define specific actions to be taken under changing system conditions that will result in compliance with expected quality levels and system lifetime as describe in section 3.5. In [9] Huygens et al. elaborate on the importance of providing the network administrator access to mechanisms that can influence the ensemble of running applications and thus fine-tune system functionality. The admin is able to execute his responsibilities without the burden of the implementation of business functionality or low level sensor programming. In our implementation, strategies are described with event condition action semantics in human readable form, facilitating comprehension and extension (see the code snippet below). In our model the network administrator is not restricted to use a specific notation. Using the provided interfaces she is able to evaluate and enact required adaptations.

```
Event: If battery level > 80
Condition: for energy category = 2;
Action: do maxSamplingFreq = 100;
```

The evaluation of these strategies is orthogonal to the running applications. This implies that network administrators need no a-priori knowledge of what applications will be using the WSN. Finding the extent of efficacy and the optimal strategies to provide the highest benefits in multi-purpose WSNs is the main focus of our future work.

5.2 Adaptability and Customize-Ability of the Middleware

Adaptability: is the system’s capacity to enact context aware run-time reconfiguration. Strategies are the way one may express reconfiguration actions which account for contextual conditions. We provide the SMC and RM abstractions which enforce these strategies to achieve autonomic and lightweight run-time reconfiguration. The reconfiguration can be system-wide e.g. using allocation, memory management strategies (see sections 3.5 and 3.8) or fine-grained e.g. adaptation strategies (see section 3.5). The strategies are currently implemented using event condition action notation which can be easily understood and extended. E.g. currently component level adaptation is limited to restrict the sampling frequency to save energy but they may be easily extended to account for monetary costs or other factors. Introspection capabilities offered in the network admin API allow our system to provided run-time details on component dependencies and execution of all services; which is essential information in reconfiguration efforts.
Customize-ability: is the system’s capacity to fine-tune or extend its functionality during run-time without service interruption. As we discussed in section 3 our loosely coupled system allows for new services or additional clients to share a common pool of components without modification to any current service composition. The pool of components can be easily extended with the implementation of additional SSCs or DPCs with no modifications on existing services (see section 3.3). Application-specific functionality may be implemented and leverage the node level client API to achieve significant benefits from in-network processing. Furthermore a running application may have varying QoD requirements which can be easily expressed in the per-instance service request and transparently configured by the SMC (see sections 3.4 and 3.5).

6 Related Work

In this section we discuss the previously highlighted benefits in the context of state of the art [1,4,5,11]. We considered approaches that contribute programming abstractions designed to provide application support in the context of multi-purpose WSNs. We narrowed the landscape further by selecting one approach from some of the more prominent models used in WSNs. These include: service oriented [1], modularized agent like abstractions [4], content-based publish-subscribe [5] and database oriented [11]. These approaches provide a high level abstraction that allows each application to express its QoD and non-functional concerns, a mediation layer to manage service selection and low level abstractions to expose network resource and allow these to be shared.

All these approaches are designed to be application independent and all offer the possibility of specifying some QoD requirements per every query, request or subscription instance. In this context service requests, queries and subscriptions all serve the same functional purpose, expressing application interests and commonly specify: required service, the duration of the service, temporal resolution or sampling frequency and spatial resolution or target location. Access to WSN resources is offered as software services using code modularization, mainly implemented with component-like abstractions.

6.1 Configuration in State of the Art

The main issue impacting configuration in these approaches is that they do not clearly separate operational concerns and they all rely on an all-knowing programmer. This programmer is expected to know low level platform specific programming and high level implementation of business functionality; this of course implies domain specific knowledge. Additionally they need to have in-depth knowledge about network monitoring and management. This of course is only feasible in research oriented WSN deployments but not for commercially viable WSN deployments. We will briefly elaborate on relevant differences of the evaluated approaches:

Servilla [4] uses ServillaScript to express queries to be executed by the WSN, which requires the application developer to learn a scripting language similar to JavaScript. Additionally complexity arises because there is no notion of time in queries and application developers are expected to know how many transmission hops the query can execute, which is not easily estimated under changing conditions. Service specification requires the use of yet another programming language ServillaSpec which further burdens service developers.

TinySOA [1] uses an extended event condition action syntax and a service oriented query model to specify service requests which is comparable to our service request specification but limited, service composition is not possible and deploying application-specific components is not discussed.

TinyCOPS [5] uses subscriptions to specify service requests and leverages the concept of subscription meta-data to express quality requirements for each subscription. Expressing application requirement in terms of subscriptions is rather straight forward but limited, service composition is not possible and deploying application-specific components is not possible.

TinyDB [11] uses SQL syntax to specify queries and related quality requirements. It offers a rather static view of the WSN since it abstracts away all functionality and presents it strictly as a database. Retrieving sensed information and executing aggregation or filtering processes is possible but the information is only accessible through a centralized user interface.

6.2 Adaptability and Customize-Ability in State of the Art

The evaluated approaches are all designed strongly toward either a macro-programming e.g. TinySOA, TinyCOPS, TinyDB or node-centric approach e.g. Servilla. The former are usually characterized by higher-level abstractions that focus mainly on the behavior of the entire network. Node-centric programming generally refers to programming abstractions used to express the application processing from the point of view of the individual nodes.

Macro-programming approaches are usually characterized by abstracting away node level interactions which limits the customize-ability by restricting the possibility to deploy application-specific components in the network to leverage in-network processing and extend functionality. On the other hand node-centric approaches allow for the possibility of deploying application-specific components but lack some higher level abstractions to unburden application developers of implementing common use functionality e.g. having to realize service composition. Our approach provides both, high level abstractions e.g. the SMC and node-centric abstractions e.g. the RM and annotated components. This allows our middleware to provide both: high-level functionality e.g. service composition and fine-grained control e.g. component level adaptation.

None of the evaluated approaches [1,4,5,11] provide abstractions to enable modification or extension of reconfiguration actions or introspection into run-time behavior. Our approach provides system strategies that inform reconfiguration actions both at system level and at node level and introspection of component dependencies, allocated requests with their respective parameters. We briefly comment on the more relevant differences in each of these approaches:

TinyCOPS [5] only offers a notion of control information with soft semantics to inform subscriptions but it is not really clear to what extent this could influence run-time reconfiguration and it is left up to the application developer to determine the control information to be used. It offers Service Extension Components (SEC) which can be used to extend functionality to meet application-specific functionality. However these components cannot locally interact with other services, they need to subscribe to receive events of interest with a centralized broker, limiting the possibility of composition.

Servilla [4] offers direct access to services on nodes through the implementation of tasks, these tasks implement application-specific functionality. They offer support for discovery, matching and binding but the application developer is left to realize any needed service compositions. Using ServillaSpec the service developer can enhance service descriptions with semantic information but this is only used during the service matching process not to inform system reconfiguration.

In TinySOA [1] the extension of functionality is possible by implementing new services. However no support for run-time reconfiguration or service composition e.g. sense, aggregate and persist is offered.

TinyDB [11] exposes the WSN as strictly a database. They offer support for service composition which offers the possibility to request sensor data, aggregate or filter it. However the approach does not provide the possibility to add services at runtime or to deploy application-specific components and leverage in-network processing.

7 Conclusion

This paper presented a lightweight, component-based service platform for WSN. We argue in favor of per-instance run-time configuration of QoD attributes and we demonstrate how our approach leverages per-instance QoD configuration and a separation of operational concerns to achieve simpler configuration and improve adaptability and customize-ability of the WSN.

In the short term we plan to record a baseline of resource usage under concurrent use and varying system loads. We will identify key QoS attributes that may be representative of global system state. In the long term we plan to further investigate and thoroughly evaluate system strategies for multi-purpose WSN.

Acknowledgments. The Research for this paper was partially funded by IMEC, the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, and by the Research Fund K.U. Leuven for IWT-SBO-STADIUM [16].

References

1. Rezgui, A., Eltoweissy,M.: Service-oriented sensor–actuator networks: Promises, challenges, and the road ahead, Elsevier, Computer Communications 30 (2007) 2627–2648.
2. Basaran, C., Kang, K.: Quality of Service in Wireless Sensor Networks. In Guide to Wireless Sensor Networks, Computer communications and Networks, Springer-Verlag, 2009.
3. Murphy, A., Heinzelman, W.: MiLAN: Middleware Linking Applications and Networks, TR-795, University of Rochester, Computer Science, Nov. 2002
4. Fok, C., Roman, G., Lu, C.: Enhanced coordination in sensor networks through flexible service provisioning, Springer-Verlag, Coordination 2009, LNCS 5521, 2009, pp.66-85.
5. Hauer,J., Handziski,V., Kopke,W., Wolisz, A.: A Component Framework for Content-based Publish/Subscribe in Sensor Networks, In Proc. 5th EWSN, LNCS, pp. 369-385, 2008.
6. del Cid, P.J., et al.: Middleware for Resource Sharing in Multi-purpose Wireless Sensor Networks, IEEE In press Proc. NESEA10, China,Nov. 2010.
7. Yu, Y., Rittle, L.J., Bhandari, V., LeBrun, J.: Supporting concurrent applications in wireless sensor networks. ACM Proc. of Sensys06, New York, NY, USA, 2006, pp.139–152.
8. Steffan, J., Fiege, L., Cilia, M. Buchmann, A.: Towards multi-purpose wireless sensor networks, in International Conference on Sensor Networks, Aug. 2005.
9. Huygens, C., et al.: Streamlining development for Networked Embedded Systems using multiple paradigms, IEEE Software, v.27, i. 5, pp. 45-52, Sept. 2010.
10. SunSPOT: http://www.sunspotworld.com/, visited July 2010.
11. Madden,S., Hong,W.: TinyDB: An Acquisitional Query Processing System for Sensor Networks, ACM Transactions on Database Systems, V. 30, No. 1, March 2005, pp.122–173.
12. Sykes, D., Heaven, W., Magee, J., Kramer, J.: Exploiting Non-Functional Preferences in Architectural Adaptation for Self-Managed Systems, ACM, SAC'10 March 22-26, 2010.
13. Lachenmann, A., Marron, P.J., Minder, D., Rothennel, K.,: Meeting Lifetime Goals with Energy Levels, ACM Proc. of SenSys07, 2007.
14. Hu, H., Han, Y., Huang, K., Li, G., Zhao, Z.: A Pattern-Based Approach to Facilitating Service Composition, LNCS 3252, Springer-Verlag Berlin Heidelberg, pp. 90–98, 2004.
15. D. Hughes, et al., "LooCI: A Loosely Coupled Component Infrastructure for Embedded Network Eccentric Systems," ACM Proc. of MoMM'09, Kuala Lumpur, 2009.
16. IWT Stadium project 80037: http://distrinet.cs.kuleuven.be/projects/stadium