Middleware for Resource Sharing in Multi-purpose Wireless Sensor Networks

Pedro Javier del Cid, Sam Michiels, Wouter Joosen
IBBT-Distrinet Research Group
Katholieke Universiteit Leuven
Leuven, Belgium
{name.lastname}@cs.kuleuven.be

Danny Hughes
Xi'an Jiaotong-Liverpool University
Suzhou, Jiangsu, 215123, China
daniel.hughes@xjtlu.edu.cn

Abstract—**In order to improve application reaction times and decrease overall transmission overhead, Wireless Sensor Network (WSN) applications are being developed to push intelligence into the network. In multi-purpose enterprise deployments of WSNs the infrastructure is considered a light-weight service platform that can provide services for multiple concurrent distributed applications. In this context our middleware focuses on efficiently managing shared resources while considering Quality of Data (QoD) and context aware operation. In this paper we address the issue of how concurrent use of WSN services may lead to consequential contention over a sensor node's resources. We contribute by introducing share-able components that minimize the consequential resources needed and a resource planner that effectively reserves these resources. A prototype implementation and evaluation are provided.**

Keywords- resource management; wireless sensor network; service composition; optimization; adaptive middleware

I. INTRODUCTION

Wireless sensor networks (WSNs) in enterprise deployments support the integration of environmental data into applications, from mobile devices to back-end enterprise infrastructure. WSNs are evolving towards interconnected, sensing, processing and actuating infrastructures that are expected to provide services for multiple concurrent clients. In a multi-purpose WSN, concurrently running distributed applications share network resources and each may have varying Quality of Data (QoD) requirements. QoD refers to application-specific data quality properties [16], such as reliability and resolution. 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.

WSN deployments are typically large in scale, heterogeneous, subject to unreliable networking, node mobility, resource constraints and are expected to operate unattended for long periods. In the context of enterprise deployments, WSN infrastructures are a reusable asset [1]. In such scenarios, resource sharing is vital to reduce the deployment and administrative costs, thus increasing the return on investment and usefulness of the network [2,3].

To support shared infrastructure, multi-application scenarios one first needs to decouple the applications from the network and expose network resources as a reusable set of services. One requires appropriate service discovery, service selection and mechanisms to deal with resource contention due to concurrent use. Some of these issues are separately addressed in current state of the art in the WSN domain through a variety of approaches, including application driven e.g [4,5] and component-based [7,8,10] among others. Sensor node resource scheduling and allocation e.g. CPU, sensors and memory are commonly left to low level mechanisms provided by the operating system e.g. tinyOS [11], Contiki[12] or virtual machines such as Squawk [13] and DAVIM [15].

These distributed applications run independently of each other with varying QoD requirements, have no predetermined global goal or inter-application coordination and thus compete over network resources. The common core functionality is to sense the environment, perform in-network processing and eventually transmit the data to the interested party. This leads to the main shortcoming of previous approaches: *concurrent use of services leads to consequential contention over scarce node resources*. In this context the use of a service e.g. temperature sensing, is specified through a service request and entails more than just access to the sensor. To successfully support a service request to its completion, one would also require processing, storing and eventually transmitting of the resulting data.

Direct contention over a resource is currently managed through the use of low level mechanisms. For example, components A and B need access to a single light sensor and corresponding CPU cycles. Consequential resource contention is not currently addressed. A consequential resource refers to any non-direct resource needed to support an allocated request e.g. when using a sensor you will need not only access to the sensor itself but dynamic memory for processing and static memory to store the data or access to the radio to transmit. We define consequential contention as the moment when two or more running services require a limited consequential resource e.g. memory, to accomplish their tasks and there is not enough availability to serve these requests appropriately. In this paper we focus on two consequential contention problems:

- **Proliferation of components with functional overlap**: In the current state of the art, for every service request with different QoD requirements, a new service composition is used. Each service composition requires the instantiation of parameterized components. Serving multiple requests requires multiple

IWT STADiUM project No.80037 and IMEC

compositions and multiple component instances that implement the same functionality. This considerably increases the amount of resources needed to support additional request. This will quickly overload the nodes capacity and lead to consequential contention problem over dynamic and static memory on sensor nodes.

- **Unpredictable memory requirements**: Services being used to support service requests will generate data to be stored (memory is then a consequential resource). The amount of services that will be required and the target locations is unknown before runtime. This means that concurrently running applications generate unpredictable amounts of data that will lead to consequential contention over dynamic and static memory as well.

In this paper we propose a distributed WSN management middleware that leverages on share-able components that may be concurrently used in several compositions, each with varying QoD parameters. This directly decreases the amount of consequential resources needed to allocated additional requests. We also offer a light-weight per node resource planner that calculates required capacity and reserves all consequential resources needed to fulfill allocated services.

II. MOTIVATION

One of the main objectives of shared enterprise deployments is to maximize the return on the investment in the WSN infrastructure. To this end we attempt to exploit the networks potential to the fullest extent and to achieve this we present the WSN as a light weight service infrastructure. We focus on maximizing the amount of concurrent and varying QoD-aware requests the network can successfully support. To accomplish this, we try to minimize the amount of resources needed to support any additional request and ensure that the consequential resources needed to support the allocated requests will be available. Capacity planning is one of the most critical responsibilities in the management of an infrastructure [14] to ensure that adequate resources are planned for and provided. However, in our context predicting future workloads is not feasible, making the case for a run-time approach.

Consider a WSN deployed in a corporate warehouse (see Figure 1). Sensor nodes are deployed at locations A, B, C and D. The deployment is shared by multiple departments, each with its own applications and having different QoD requirements. The maintenance department periodically gathers sensing information for a HVAC application. The logistics department deploys a tracking application that provides information on package movement and environmental conditions.

The HVAC application has a back-end component that periodically requests temperature and light measurements throughout the warehouse to determine general AC or heating requirements. Additionally it 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 back-end tracking application submits request for temperature and position of packages it is tracking. Additionally a specialized component is deployed to high value packages which use light and accelerometer readings to locally determine package handling and tampering and submit the appropriate alarms when necessary.

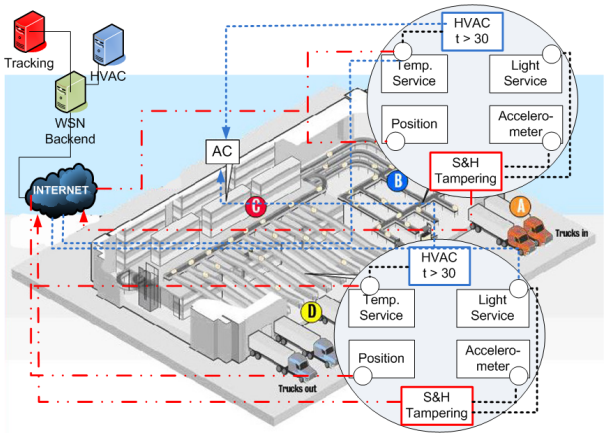


Figure 1. Deployment scenario

The HVAC application requests temperature to be sensed in the warehouse with a sampling frequency of 5 minutes and these should be averaged every hour during the next 30 days (see Figure 2). The aggregated data is to be stored. Current approaches would address these requirements by instantiating a composition involving a component to retrieve temperature readings every 5 min., another component to average the sensed data and a final component to persist the data. The tracking application requests light readings every 10 minutes for the next 15 days. These readings are to be averaged every 2 hours and the results are required to persist. To fulfill these requirements a second composition would be created requiring 2 additional components be instantiated that implement repeated functionality. This will substantially increase the requirements on static and dynamic memory for every request supported. As one can imagine this would rapidly overload sensor resources.

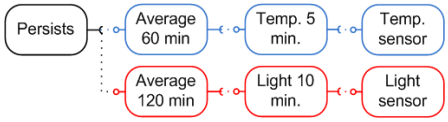


Figure 2. Component based service compositions

In the scenario above the allocated services would start generating data and consequently require dynamic memory while processing and afterwards static memory for persistence. The resources to initially serve a request may be available, the resources to support it through the expected service duration may not be.

III. REQUIREMENTS TO SUPPORT CONSEQUENTIAL RESOURCE CONTENTION

In order to maximize the amount of concurrent and varying QoD-aware requests the network can successfully support and to minimize the amount of resources required we need to systematically address consequential contention. In this

context, applications run un-aware of each other, there is no inter-application coordination and requirements may change constantly. In the previous section, we discussed the primary reasons for consequential contention; from these the following middleware requirements to mitigate consequential resource contention may be derived:

- Minimize the amount of consequential resources required to support additional service requests.
- Estimate and reserve the consequential resources to be used by each allocated service request.
- Distribute workload efficiently throughout the network thus avoiding the overload of any resource.

In this paper we discuss our solutions for requirements 1 and 2 in the context of consequential usage of static and dynamic memory. Workload distribution strategies that maximizing service allocation capacity for the WSN is part of our future work.

IV. MIDDLEWARE OVERVIEW

We propose a middleware approach based on configurable components that may be used in multiple concurrently running compositions and allow different QoD to be parameterized for each composition. Figure 3 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 as specified. The service request specification is used to express the desired functionality and data qualities for every service instance.

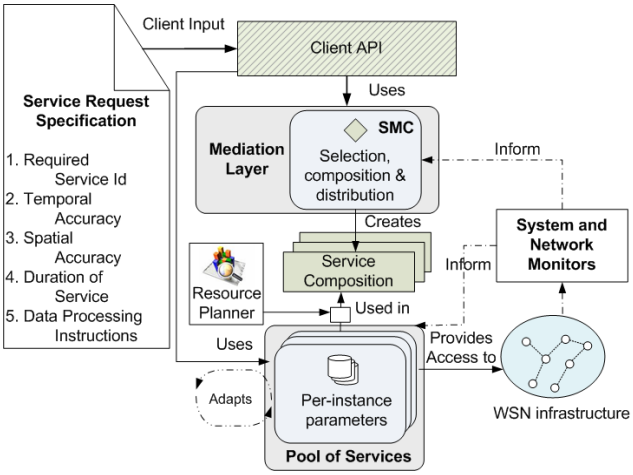


Figure 3. Middleware overview.

A. The Mediation Layer

The mediation layer is running in the backend and in cluster heads in the WSN and 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. In this way, requirements from different users are handled independently, thus avoiding potential conflicts due to resource competition or varying QoD requirements.

B. Service Requests

Clients use the service request specification to express their QoD requirements in a per service instance manner. In the specification one expresses 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. 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 i.e. 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,samplingfre
quency,duration,targetLocation,DataProcessServi
ceId[],parameter[]);
```

C. WSN Services

The pool of components available to create service compositions is comprised of basic sensing services and data processing services (see Figure 4A). 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: an SSC reads, timestamps and stores temperature readings which an averaging component processes and finally a persistence component stores to static memory. Each component has parameters that are set separately for every service composition the component is used in. These parameters are used by a per node resource planner that calculates required capacity and reserves all consequential resources, more details on this are presented in later sections.

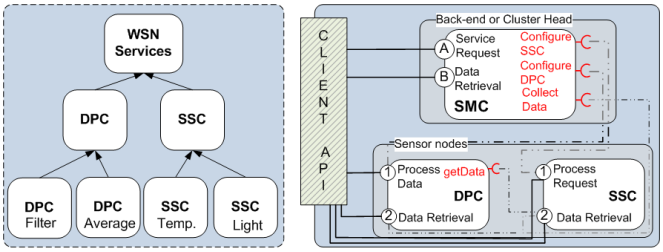


Figure 4. WSN services and client API.

D. *The Client API*

Clients interact with the middleware through a distributed API (see Figure 4B). Clients located at the back-end or on a cluster head node interact with the mediation layer through interfaces A and B. The former interface is used to submit service requests and the latter to retrieve processed data. Clients that are located on sensor nodes i.e. specialized components that implement application specific functionality, interact with the middleware 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);
```

E. *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 sensing service (SSC) and zero-to-many post-collection data processing services (DPCs) (see figure 5, each composition is denoted with a different type of line). Dotted lines depict a composition that only involves an SSC, the regular lines depict a composition that uses one SSC and one DPC and the dashed dot lines depict a composition that involves an SSC and 2 DPCs.

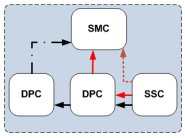


Figure 5. Valid service compositions.

V. SUPPORT FOR CONSEQUENTIAL RESOURCE CONTENTION

In this section we discuss how we minimize the amount of consequential resources required to support additional service requests and how we estimate and reserve the consequential resources to be used by each allocated service request.

A. *Minimizing Consequential Resources for additional requests*

As we described in Sections 1 and 2, in the current state of the art each service composition requires the instantiation of parameterized components. Serving multiple requests requires multiple compositions and multiple component instances that implement the same functionality (see Figure 2). This considerably increases the amount of resources needed to support additional request. This will quickly overload the nodes capacity and lead to consequential contention problem over dynamic and static memory on sensor nodes. To address this problem we have designed and implemented our SSCs and DPCs so they may be concurrently used in multiple service compositions thus avoiding the need to instantiate components that implement repeated functionality, as explained in the next paragraph.

Share-able Components: We consider that introducing a new component for every service instance that requires different parameterization is not efficient. Components may be used as depicted in Figure 5, where the color lines represent concurrently running compositions. We separate the functional code from the meta-data and share the same component instance across multiple service compositions (see Figure 6). 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.

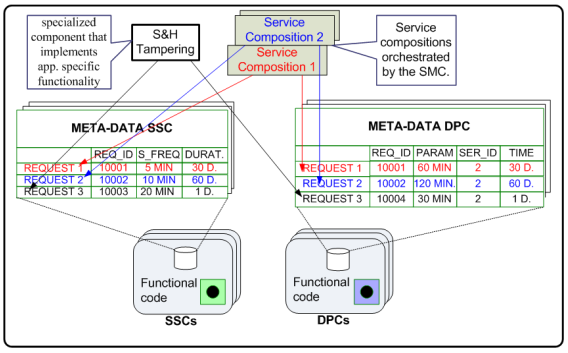


Figure 6. Configuration semantics for components.

Configuration semantics for each service composition are extracted from the client specified service request. The configuration semantics include client specified QoD, services involved in each composition and related parameterization. The order in which requested services are operated is expressed through composition strategies in the SMC and not specific to every service request. The SMC parses the service request and extracts configuration parameters which it autonomously submits to the corresponding SSC or DPCs (denoted in dash-dotted lines in Figure 4B). Clients deployed on a sensor node i.e. specialized components that implements application specific functionality, may also submit these configuration parameters (see Figure 6) directly to the SSC or DPC using ProcessRequest@SSC or ProcessData@DPC as can be seen in the client API in Figure 4B.

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.

Configuration of SSCs: Request Id, sampling frequency and service duration is the meta-data used to configure the SSC sensing functionality. Once the meta-data used to process each service request is available processing may begin as one can see in Figure 7A. In the SSC the sensed value is retrieved at the specified sampling frequency from the sensor driver, a corresponding timestamp is included and stored with the corresponding request Id as the key value in a table. In an interleaved manner, the same process is carried out for other requests, storing these values each with its corresponding request Id to facilitate data retrieval. The process is repeated for the specified duration interval in each service request.

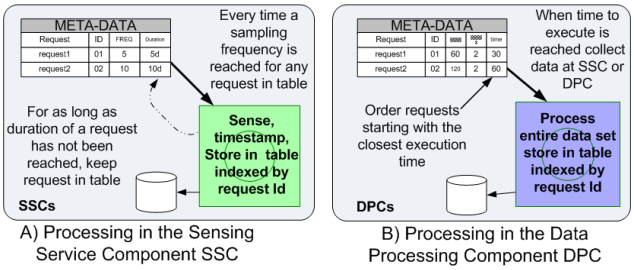


Figure 7. Processing in SSCs and DPCs.

Configuration of DPCs: In the DPCs for each service request the following meta-data is used: the execution times, the corresponding SSC or previous DPC for data retrieval and a parameter. Data is retrieved and processed in a sequentially ordered manner based on specified execution. E.g. averaged or filter according to a set threshold (see Figure 7B). The entire collection of data samples for a request is processed and stored in a table with the corresponding request Id to facilitate data retrieval. This process is repeated for every service request received. The parameter varies depending on the function being implemented e.g. in the averaging component it is the length of time to average, in the data filter it is threshold value.

B. Effectively Calculating Consequential Resources Needed

In order to effectively calculate the amount of static and dynamic memory that will be used by the middleware we use an off-line process to establish a memory baseline and a run-time process to establish run-time memory requirements. Each node has a light-weight resource planner that is able to estimate specific amounts of memory needed to fulfill each request for both SSCs and DPCs. The resource planner also reserves these required resources. This guarantees that every service request will have the needed consequential resources e.g. memory, to be processed successfully through the service duration.

Generating a baseline for component's resource use: A baseline of resource consumption is recorded for each component type on each platform i.e. hardware and runtime environment, to be used. This process is done off-line. The baseline includes:

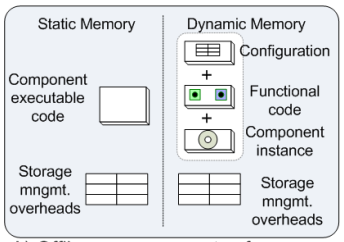
1) S*tatic memory requirements to store component code*: record the memory required to store component executable code.

2) *Dynamic memory requirements:* Each component requires dynamic memory to be instantiated. It also requires varying amounts of memory during the execution of its functional code (see Fig. 7) and the creation and maintenance of the configuration meta-data (see Fig. 6). At different moments in time during these processes, the amount of required memory varies. To account for this, we collect memory usage information at various points in the processing cycle. This provides the max peak of memory consumption per complete processing cycle. We consider these max peaks of required memory will be the same for any request processed in a particular component type. Each specific SSC or DPC must be measured accordingly.

During the processing of multiple service requests we measure the dynamic memory required to serve additional service requests in a component instance and any additional memory required due to housekeeping overheads such as lagging garbage collection.

Additionally, random measurements are taken while storing records of varying sizes from 32 byte to 32Kb. These are indexed with a request Id, in both dynamic and static memory. In this manner we ascertain the amount of memory overhead generated by platform specific data storage and related housekeeping.

The total required bytes of static and dynamic memory (see F. 8A) are recorded and assumed to be constant for a particular implementation of each specific service type. Every component has these amounts recorded as values available and introspect-able as annotated component attributes (see Fig. 8B).



Attribute	Units
requiredStaticMemory	bytes
requiredDynamicMemory	bytes
staticStorageOverhead	Bytes/ record
dynamicStorageOverhead	Bytes/ record
dynamicMemAddService	Bytes

B) Annotated component attributes

Figure 8. Baseline memory requirements and component attributes.

Run-time capacity planning: Each service composition specifies which components will be used to serve a specific service request. For every component to be used, we calculate both dynamic and static memory requirements, even though it is possible that the static memory will only be used after the last component has finished processing the data sets. We do this because one of the adaptation strategies specifies that when battery level is under 10%, the persistence component stores all data sets from running components. For this reason we need to assure that enough static memory is available for any data set being processed.

The on-node resource planner performs run-time calculations to determine exact amounts of memory required to process each service request according to the meta-data provided for configuration. During processing in an SSC or DPC the entire record set is kept in dynamic memory and transferred to static memory by the persistence component only after the data set has been processed. Sensed or processed data available in

dynamic memory is only erased after persisted to static memory or through a specific delete instruction available in the data retrieval interfaces of the SMC, DPCs or SSCs (see Figure 4B). Calculations for SSC and DPC are different and described below.

1) *Calculations for the SSC*: Dynamic memory required is equal to the output data set size + dynamic storage overhead. The Static memory required is equal to the output data set size + static storage overheads. The calculation of output data set size must be done at runtime for each request because the amount of records produced will vary depending on sampling frequency and service duration. Given the amount of records and the data type we calculate the output data set size. Storage overheads are known from the off-line baseline.

2) *Calculation for the DPC*: Required dynamic memory is equal to the input data set size + dynamic storage overhead + output data set size + dynamic storage overhead. Required static memory is equal to output data set size + the static storage overhead. To calculate the size of the input dataset: record count is obtained from the data and multiplied by the data type size. The size of the output data set varies depending on the mathematical function applied to the samples and the specific configuration parameters. In the case of the averaging component and other time-series analysis based components: i) time-span incurred in the time samples is calculated based on the timestamps on the data. ii) [time-span] / [user specified parameter] = amount of records the output data set will have. Given this amount * byte size of the data type required memory for the output data set is calculated.

In the case of filtering components we assume that in the worst case the output data set will have the same size as the input data set. This is done because there is no way to predict how many data samples will be filtered out, but we know no records will be added, so estimating equal input and output data sets is a safe assumption. To these values we add corresponding housekeeping overheads.

SSC
runtimeStaticMem = output data set + staticStorageOverhead
runtimeDynamicMem= output data set + dynamicStorageOverhead
DPC
runtimeStaticMem = output data set + staticStorageOverhead
runtimeDynamicMem = input data set + dynamicStorageOverhead + output data set + dynamicStorageOverhead

A) Run-time capacity planning

Per component instance
reservedStatMem = requiredStaticMemory
reservedDynamMem = requiredDynamicMemory
Per Service instance
reservedStatMem = runtimeStaticMem
reservedDynamMem = runtimeDynamicMem + dynamicMemAddService

B) Resource reservation for SSCs and DPCs

Figure 9. Run-time capacity planning and resource reservation.

Reservation of resources: The planner estimates the amount of dynamic memory and static memory required for every component instantiated on a node and every time a new service request is received. The reservation is done on a first come first served basis. Off-line, a Max usage quota is defined by the network admin e.g. from 10 Kb of dynamic memory allocate 50% to support running services. These usage quotas represent the 100% of dynamic and static memory that is available. Every reservation granted subtracts from the memory available. Every resource that is released after service duration time expires adds back to the available amount.

For each component instance: These calculations are updated in two scenarios: i) A new component is going to be instantiated. ii) A component is removed from the node. As one may see in Fig. 9B, the reserved static memory is equal to the required static memory as calculated in the offline procedure (see Fig. 8A). The reserved dynamic memory is equal to the required dynamic memory as calculated in the off-line procedure (see Fig.8A).

For every service request: The reserved static memory is equal to the run-time static calculated during run-time capacity planning (see Fig 9B). The reserved dynamic memory is equal to the run-time dynamic memory calculated during run-time capacity planning and the dynamic memory required for additional services as calculated in the off-line process (see Fig 9B). These may be updated two scenarios: i) every time the service duration from a previously allocated service expires. ii) every time a data set is deleted either from dynamic or static memory. SSCs and DPCs alert the planner every time they have executed a data delete.

The memory reservation is valid for the middleware layer only. This means that service requests are only allocated to components when the consequential resources required are available. The OS or VM are not aware of this memory reservations and do not enforce them.

VI. MIDDLEWARE IMPLEMENTATION

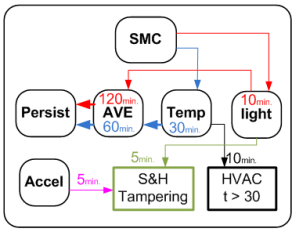
We have implemented a prototype that supports the application scenario as described in Section 2. The implementation was done in Java ME CLDC1.1 configuration on the SunSPOT platform [13].

A. *Implemented Scenario.*

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. 10 (as is not directly relevant to our scenario, we omit the specification of target location).

```
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

Figure 10. Implemented scenario

The HVAC specialized component that determines if the AC needs to be adjusted 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 (see Fig.10). 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 and are depicted in Fig. 10.

VII. EVALUATION

We evaluate our system with respect to the middleware requirements presented in Section 3: i) to determine the amount of consequential resources needed to serve additional service request we measure static and dynamic resources required to instantiate the service compositions as depicted in Fig. 10B. We compare our results to the resources required to instantiate equivalent service compositions in a component based approach designed for sensor networks. ii) to evaluate how effectively our resource planner calculates and reserves resources, we compare the amount of resources reserved by our resource planner vrs. the amount of resources actually used by the middleware during the execution of the requests.

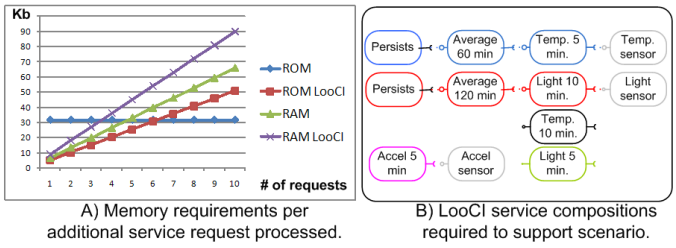
A. Minimizing Consequential Resources Needed to Support Additional Service Requests.

Using our approach, as depicted in Fig. 10B one needs to instantiate **5 service components** and the SMC. These five components are used to serve 5 concurrent requests which are fulfilled with 5 independent service compositions. As the number of served request increases the amount of instantiated components remains constant, effectively supporting the first middleware requirement. 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**. Each additional request that runs in a DPC consumes about **800 bytes** of additional dynamic memory.

As a comparison component based approach we selected LooCI [10]. We selected this approach because of its very loosely coupled component interaction and published subscribe functionality provides effective mechanisms to implement multi-purpose WSNs. To implement the functionality depicted in Fig 10 it was necessary to instantiate **9 LooCI micro-components** as depicted in Fig.11B Each LooCI component requires **3 Kb** of dynamic memory and **1.7Kb** of static memory.

Memory requirements to support service requests: In Fig.11A we plotted the amount of dynamic memory (RAM) and static memory (ROM) required to instantiate the components needed to process concurrent requests. We varied the amount of concurrent requests processed from **n**=1 to **n**=10. Each of these requests is equivalent in functionality as serviceRequest1. Results effectively demonstrate: *our approach requires less consequential resources per additional service request processed.*

One can see in Fig.11A in our approach ROM stays fixed at 31.6Kb (labeled as ROM in the graph) and for LooCI at n=1 ROM=5.1Kb and for n=10 ROM=51Kb. We plotted RAM measurements for our approach (labeled as RAM in the graph), one can see that for n=1 RAM=6.6Kb, n=10 RAM 66Kb. For the LooCI version RAM requirements at n=1 RAM=9Kb and for n=10 RAM=90Kb.



A) Memory requirements per additional service request processed.

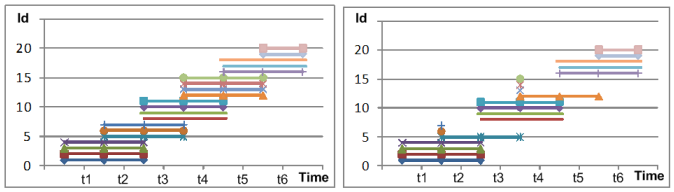
B) LooCI service compositions required to support scenario.

Figure 11. Memory required and implemented scenario with LooCI

The overhead in ROM requirements generated by our system is surpassed by LooCI after request 6, which indicates that our approach is better suited for higher load multi-purpose WSNs.

B. Effectively Calculating Consequential Resources Needed.

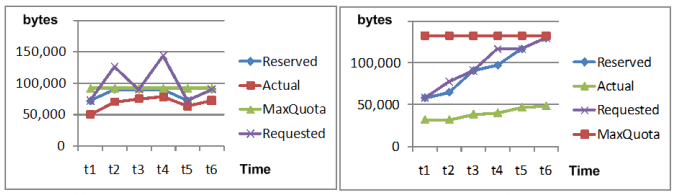
We have submitted a total of 20 service requests in 6 successive and overlapping batches to the SMC as depicted in Fig. 12A, requests from 1 to 20. The x-axis depicts elapsed time and the y-axis depicts the request Ids. One can see the submission times and durations of all requests, in Fig. 12A as they were submitted and Fig. 12B as they were actually processed. As one can see requests 6,7,13,14 and 15 were rejected by the system. All requests are equal to serviceRequest1 depicted in Fig.10.



A) Requests as submitted to the system

B) Requests as actually processed

Figure 12. Service requests submitted to the SMC.



A) Dynamic Memory

B) Static Memory

Figure 13. Runtime measurements for dynamic and static memory.

Figures 13A and 13B show dynamic and static memory measurements at times t1 through t6. In both figures one can see the requested, reserved and actual memory readings. MaxQuotas represent the max amount of memory that may be allocated by the resource planner to service requests. This amount is set by the network administrator. In Fig. 13A peaks in requested dynamic memory that exceed the MaxQuota may be seen at t2 and t4. The system rejects requests 6,7,13,14,15

hence effectively maintaining used dynamic memory under the specified MaxQuota. In Fig. 13B one may see how the actual static memory used is very low compared to the requested and reserved amounts. When instantiating a composition static memory is requested for all components, in this case the temperature and averaging components have static memory reserved but only the persistence component actually uses it. This is because there is a system policy that requires all components processing a service to have enough static memory available in case battery levels drops under 10%. Further elaboration on how modifying this system policy may affect the resource reservation is outside the scope of this paper.

Our results demonstrate: *our resource planner can reserve the consequential resources needed to support concurrent service requests.* Memory is reserved to guarantee all allocated services are supported and released after it is no longer needed, effectively supporting requirement 2.

VIII. RELATED WORK

In the current state of the art there are several efforts that aim at efficiently managing and using the resources of a sensor node such as Levels [6], Eon [9], and Pixie [17]. Levels [6] and Eon [9] both focus mainly on monitoring energy level and adapting system behavior. At run- time Levels monitors the remaining battery capacity and selects an energy level that allows the application to achieve its target lifetime. Eon's automatic energy management can dynamically adapt system states to current and predicted energy levels. The Pixie OS is based on a dataflow programming model based on the concept of resource tickets, which represents resource availability and reservations. Pixie allows the application to control resource allocation, which is not well suited to deployments where resource competition exists. Sensor management frameworks e.g. TinyCubus [18] have been proposed. TinyCubus allows multiple applications to share a WSN. It has a data management framework that selects the best suited set of components based on current system parameters, application requirements, and optimization parameters. It however does not plan for or reserve consequential resources needed.

Other resource approaches focus on managing resource use in sensor networks e.g. energy efficient routing protocols [19], in-network processing [20]. They mainly focus on increasing the lifetime of the whole sensor network not on consequential resources needed to fulfill allocated services.

IX. CONCLUSION AND FUTURE WORK

In this paper, we addressed the issue of how concurrent use of WSN services may lead to consequential contention over a sensor node's resources. We proposed solutions for consequential contention of dynamic and static memory. We demonstrated: 1) How our shareable components require less consequential resources to serve additional service requests 2) How our resource planner can effectively calculate and reserve the consequential resources needed to successfully support each service request allocated. In the short term we plan to extend our approach to cover Maximizing service allocation capacity for the WSN through the use of workload distribution strategies. In the long term we plan to extend system strategies to provide efficient system adaptation.

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 [21].

REFERENCES

[1] C. Huygens and W. Joosen, "Federated and shared use of sensor networks through security middleware," IEEE Proc. of ITNG'09, pp. 1005-1011, Nevada, USA, 2009.

[2] Y. Yu, L. J. Rittle, V. Bhandari, and J.B. LeBrun, "Supporting concurrent applications in wireless sensor networks," ACM Proc. of SenSys 06, ACM Press G ,pp. 139-152, New York, NY, USA, 2006.

[3] J. Steffan, L. Fiege, M. Cilia, and A. Buchmann, "Towards multi-purpose wireless sensor networks," Systems Communications, 2005. Proceedings , pp. 336- 341, Aug. 2005.

[4] A. Murphy and W. Heinzelman, "MiLAN: Middleware Linking Applications and Networks," TR-795, University of Rochester, Computer Science, Nov. 2002.

[5] C. Fok, G. Roman and C. Lu, " Enhanced coordination in sensor networks through flexible service provisioning," Springer-Verlag Berlin Heidelberg, Coordination 2009, LNCS 5521, pp.66-85, 2009.

[6] A Lachenmann, P. J. Marron, D. Minder, and K. Rothennel, "Meeting Lifetime Goals with Energy Levels," ACM Proc. of SenSys07, 2007.

[7] L. Mottola, G. Picco and A. Sheikh, "FiGaRo:Fine-Grained Software Reconfiguration for Wireless Sensor Networks," LNCS, Volume 4913/2008, pp. 286-304, 2008.

[8] P. Costa, et al, "The RUNES Middleware for Networked Embedded Systems and its Application in a Disaster Management Scenario," IEEE Proc. PerCom'07, White Plains, New York, pp. 69 – 78, 2007.

[9] J. Sorber, et al. "Eon: A Language and Runtime System for Perpetual Systems", ACM Proc. Of SenSys07, 2007.

[10] D. Hughes, et al., "LooCI: A Loosely Coupled Component Infrastructure for Embedded Network Eccentric Systems," ACM Proc. of MoMM'09, Kuala Lumpur, 2009.

[11] TinyOS: http://www.tinyos.net/, visited august 2010.

[12] Contiki: http://www.sics.se/contiki/, visited July 2010.

[13] SunSPOT: http://www.sunspotworld.com/, visited July 2010

[14] Oracle Corporation,: Capacity planning guide, white paper, may 2009.

[15] S. Michiels, W. Horre, W. Joosen, P. Verbaeten, "DAVIM: A Dynamically Adaptable Virtual Machine for Sensor Networks," ACM Proc. Of MidSens 06, ACM Press, pp. 7–12,2006.

[16] C. Basaran and K. Kang, "Quality of Service in Wireless Sensor Networks," In Computer communications and Networks, Springer-Verlag, London Limited, pp.305-321, 2009.

[17] K. Lorincz, B. Chen, J. Watennan, G. W. Allen, and M. Welsh, "Resource Aware Programming in the Pixie OS," ACM Proc. of SenSys07, 2007.

[18] P.J. Marron, D. Minder, A. Lachenmann, and K. Rothermel, "TinyCubus: An adaptive cross-layer framework for sensor networks," Proc. of the 2nd European Workshop on Wireless Sensor Networks, pp. 278- 289, Feb. 2005.

[19] C.F. Huang, L.C. Lo, Y.C. Tseng, and W.T. Chen, "De-centralized energy-conserving and coverage-preserving protocols for wireless sensor networks," ACM Trans. Sen. Netw., 2(2):182–187, 2006.

[20] Y. Yao and J.Gehrke, "The cougar approach to in-network query processing in sensor networks," SIGMOND Record, vol. 31, No. 3, sept. 2002.

[21] IWT Stadium project 80037: http://distrinet.cs.kuleuven.be