Evolving Wireless Sensor Network Behavior Through Adaptability Points in Middleware Architectures

Pedro Javier del Cid, Daniel Hughes, Sam Michiels, Wouter Joosen

Abstract—Reflection has been proven to be a powerful mechanism to address software adaptation in middleware architectures; however this concept requires that the middleware be open and that modification of all of its functionality and behavior be possible. This leads to systems which are difficult to understand and may quickly overwhelm developers. Safer and more understandable approaches use modeling and put forth a partial implementation of reflective principles while limiting the possible scope of modification, as with translucent middleware. We consider that given the resource constraints in a Wireless Sensor Network (WSNs) it is preferable to limit reflective features in order to conserve computational cycles and reduce network traffic. Additionally we do not believe all modifications lie within the concerns of the application developer and we introduce a separation of operational concerns that maps different modification responsibilities and levels of abstractions to different operational roles. We introduce a middleware architecture that provides strategy-controlled adaptability points; which are available to modify the behavior of the middleware’s primary functionality. We have evaluated our approach through the implementation of a proof of concept prototype that supports an industrial use case in the logistics domain and a need-for-change scenario in the middleware’s capacity planning functionality. Results demonstrate how changes in business requirements may be effectively supported through the introduction of adaptability points.

Index Terms—Middleware, Reconfiguration, Software Adaptation, Wireless Sensor Networks

I. INTRODUCTION

Wireless sensor networks (WSNs) deployments support the integration of environmental data into applications and are typically long-lived, large in scale, resource constrained, subject to unreliable networking and node mobility. In such environments an application needs to adapt its behaviors and functionalities to cope with changing context and operational conditions, by consequence software evolution and reconfiguration become a necessity [1]. Existing approaches mainly focus on extending application functionality or modifying the underlying platform’s execution parameters based on contextual conditions. The use of middleware is a popular approach to address these issues in WSNs [21]; which separate the application from the underlying execution platforms.

Software evolution of WSN applications has been addressed through a variety of approaches e.g. runtime reconfigurable component models [3] and component frameworks [5]. Finer-grained reconfiguration is introduced either through policy based approaches [4] or allowing modifications to code units smaller than components as in TinyComponent [1].

Middleware to allow modification of the underlying execution platform in WSNs commonly use reflective principles, e.g. [2], [19] or partial reflection support, as in [3]. These commonly focus solely on providing the applications finer-grained control over the underlying platform. We currently focus on evolving the middleware itself, specifically in modifying the behavior or the way in which middleware executes its functionality; as opposed to extending its functionality or modifying execution parameters of the underlying platform.

Middleware for traditional distributed systems implement the principle of “information hiding” [15]; which abstracts away implementation specific low-level details and offers higher level abstractions that are simpler to use and configure. In WSNs, given the operational conditions, more control over middleware functionality and behavior is necessary in order to be able to inspect and adapt middleware behavior in favor of optimizing performance [2]. However managing low level details will incur in higher levels of complexity, as is the case of reflective middleware [16]. Reflective middleware makes the internal representation of the middleware explicit and, thus, accessible to be modified; this opposes the principle of transparency or information hiding and through introspection may achieve adaptation. These approaches usually make all functionality and middleware behavior available for modification; which can rapidly become highly complex and difficult to manage [17].

In high power mobile platforms, this increased complexity has been addressed by restricting possible modifications on the middleware and has been approached by enhancing reflective principles with XML based meta-data [17] or multi-layered models of translucent middleware [16].

We consider that given the resource constraints in Wireless
Sensor Network (WSNs) limiting the scope of modification is the correct approach but the use of computationally intensive models is not energy efficient. Additionally we do not believe all modifications lie within the concerns of the application developer and we introduce a separation of operational concerns and map different modification responsibilities and levels of abstractions to different operational roles. In this paper we contribute with a middleware architecture that provides strategy-controlled adaptability points; which are available to modify the behavior of the middleware’s primary functionality. Modifying a strategy changes the middleware’s behavior thus modifying how it executes its functionality; in this way changes in business requirements may be effectively supported. To evaluate these capabilities we adapt the capacity planning functionality of our middleware; which we presented and evaluated in [6]. We modify the strategy that controls the capacity planning adaptability point in order to support new business requirements and present the prototype implementation and its evaluation. These new business requirements are introduced in the context of a need-for-change scenario.

This paper is structured as follows: Section II motivates the need to modifying the behavior of the capacity planning functionality. We present the use case, operational roles and the need-for-change scenario. Section III presents an overview of our middleware. Section IV discusses our adaptability points. Section V presents our prototype implementation and its evaluation. Section VI concludes the paper and maps the road ahead.

II. MOTIVATION

In WSNs, functionality commonly addressed through middleware may include: selecting a service provider based on current contextual conditions, modifying sensor sampling frequencies based on available battery, resources, etc. In order to implement the service provider selection functionality, a utility function could be used that accounts for different contextual parameters to rank the suitability of potential providers. One can imagine that in the future, modifications to this utility function may be required for many reasons, e.g. additional contextual sources become available or a more efficient utility function is designed. This gives rise to the need of enacting modifications on how the middleware provides a given functionality i.e. its behavior, without any modifications to its structure or control flow.

In order to evaluate the notion of adaptability points in middleware architectures, we have implemented a prototype system, made modifications to one of the offered adaptability points in our architecture and evaluated middleware performance before and after the modifications. Specifically, we have modified the runtime capacity planning functionality offered by our middleware. As discussed in Section I, we have presented and evaluated this functionality in [6].

Capacity planning is the practice of estimating the resources that will be needed over some future period of time and is one of the most critical responsibilities in the management of an infrastructure [7]. It is essential to ensure that adequate resources are planned for and provided. Providing runtime capacity planning in our middleware supports the effective control of resource use and enhances system reliability because required resources to process a service request are reserved. This functionality is controlled by a lightweight on-node resource planner. The behavior of which is controlled through a set of strategies. Each strategy is evaluated at a predetermined location in the middleware architecture. These locations are determined based on the importance of the corresponding functionality and the probability that changes to its behavior may be needed in the future. We refer to these locations as “adaptability points”. Specifically, capacity planning is controlled by a planning strategy which dictates how and when resources are reserved. This strategy is evaluated at the capacity planning adaptability point.

A. Use case

Our middleware is designed to optimize resource use while considering Quality of Data (QoD) and context aware operation for multi-purpose WSN deployments. In these deployments the infrastructure is considered a light-weight service platform that can provide services for multiple concurrent applications. Concurrently running applications share network resources without inter-application coordination and may have conflicting requirements.

Consider a WSN deployed in a corporate warehouse (see Fig.1). Sensor nodes are deployed at locations A, B, C and D. The deployment is shared by multiple stakeholders, each with its own application requirements. The maintenance department periodically gathers sensing information for a Heating Ventilation and Air Conditioning (HVAC) application. The logistics department deploys a tracking application that provides information on package movement and environmental conditions during shipping of goods.

The HVAC application 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 tracking application monitors Shipping and Handling (S&H) conditions. Warehouse temperature and humidity...
readings are recorded. On individual packages, position is also monitored. High value packages require light and accelerometer readings to locally determine package handling and tampering and submit the appropriate alarms when necessary.

Runtime capacity planning in these deployments becomes essential due to the concurrent and uncoordinated use of resources. Consider the common usage pattern in a WSN application, sense-process-react. Successfully supporting this usage pattern requires that the infrastructure is able to provide not only access to the sensor but also provide the memory required during processing, storage and access to the radio to eventually transmit. Additionally one needs to consider that multiple applications compete for limited resources demanding that allocation for these limited resources be done efficiently; thus making the case for runtime capacity planning.

B. Operational roles

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 [13]: application developers, service developers and network administrators. The primary motivation for this separation of operational concerns is based on the fact that managing large scale computational infrastructures across multiple stakeholders is a complicated undertaking. As may be seen from computer networks, web-based service or grid infrastructures. In order to support a large client base and achieve economies of scale in the deployment of such infrastructures a separation of operational concerns is commonly used.

1) Application developers (application owners in [13]) will be concerned with achieving high-level business goals and will undertake the implementation of domain specific business logic.

2) Service developers (component developers in [13]) 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 [14] sensor node i.e. atomic middleware services as later introduced in Section III-A.

3) Network administrators (infrastructure owner in [13]) will be concerned with monitoring network Quality of Service (QoS) and Quality of Data (QoD). They will also configure and maintain common use software services e.g. temperature, aggregation (atomic middleware services as later introduced in Section III-A). 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.

C. Need-for-change scenarios

In this section we put forth two need-for-change scenarios to the capacity planning functionality, in order to exemplify the many situations that may lead to required changes in middleware behavior.

Capacity planning in our architecture is controlled by a planning strategy which dictates how and when resources are allocated and reserved. This strategy dictates how the middleware provides this particular functionality, thus its behavior. Currently this strategy allocates resources on a First Come First Serve (FCFS) basis until the resource’s usage quota is full, after which any additional requests are denied. One may imagine a multitude of situations that would require the modification of this strategy with the intention of changing how and when these resources are allocated. Any of these situations may be regarded as a need-for-change scenario. We elaborate on two scenarios:

1) Prioritizing subscribers: The payment model currently in use for the WSN, is pay per use and does not allow any prioritization of important clients or sensitive data. The FCFS strategy was designed given these considerations. It has been decided that a new payment model will be offered for service usage on the WSN. Different subscription levels will be offered, e.g. elite and standard. Elite subscribers will receive prioritized access to resources and their requests will be processed before standard subscriber requests. Subscriber status should be considered in the planning strategy in order to prioritize resource use. In this scenario elite subscribers are to have priority access to any resource over standard subscribers. Given that the current FCFS planning strategy reserves resources in a first come first serve basis it is not suited to account for subscriber status. This creates the need to modify the behavior of the capacity planning functionality, specifically how the planning strategy allocates resources and what factors are accounted for. Thus a need for change scenario.

2) Compliance with government regulations: New regulations now mandate that all sensor platforms in use in the harbor areas must make their resources available in case of disaster situations, e.g. a fire. In this case sensing and processing data related to the ongoing disaster must make priority over all other allocations. Given that the current FCFS planning strategy does not account for request priorities, this scenario cannot be supported, hence a need to modify middleware behavior. Thus a need for change scenario.

III. MIDDLEWARE OVERVIEW

Our middleware platform is designed to maximize potential resource usage and ensure controlled resource use in multi-purpose WSNs. The workload in this environment is high, concurrent and unpredictable. The middleware actively calculates trade-offs between i) quality requirements associated with service requests and ii) resource capabilities and sensing/actuating alternatives throughout the WSN. Interpretation of these trade-offs enables the middleware to translate service requests to customized component compositions and to instantiate them at well-selected resource providers.

Clients express their requirements through the submission of a service request, in accordance with the service request specification. These requests are parsed and interpreted by a service management layer; which selects the service providers and instantiates a service composition accordingly. We also provide a service framework that defines WSN services and offers mechanism to support concurrency and controlled service use.
A. The service framework

The service framework was designed to present WSN services as a pool of services available to be concurrently used in multiple compositions. It provides support for high loads of concurrent service requests and achieves simpler service composition, fine-grained reconfiguration and higher component reusability. It allows components to be transparently added or removed from any service composition without the need to re-wire existing compositions or interrupt services. Runtime variability in requested QoD can be effectively supported through fine-grained configuration of service compositions. Further discussion on the benefits achieved by the service framework may be found in [9], in the following subsections we provide a high level overview of the framework as is relevant for the context of this paper. The framework defines: 1) service meta-types, 2) service structure, 3) an approach to enable concurrency.

1) Service meta-types: The pool of components available to create service compositions is comprised of basic sensing services and data processing services, these are considered the atomic WSN services. Sensing services (SSC) are components offering typical functionality such as the retrieval of temperature or light readings (see Figure 2). They provide access to the various sensors. Data processing services (DPC) are components implementing post-collection data processing functionality; where the raw sensor data is processed to obtain the desired output.

![Figure 2. Service meta-types](image)

One may use only one DPC or a pipeline composed of multiple DPCs. In this case, DPCs implement processing steps are connected by data flow through the system, the output data of a step is the input of the following step. Each DPC may enrich input data by computing and adding information, refine data by concentrating or extracting, transform data by producing a new representation, etc. Common processing in WSNs involves, averaging, filtering, calculating a utility function, encryption, etc.

It is important to notice that DPCs may be used to address data qualities in the service request or implement some cross-cutting concerns. For example, temporal aggregation can be achieved with an averaging service, data accuracy may be increased with a specialized data filter that may remove anomalous data values that may indicate a faulty sensor from the raw sensor readings. Additionally sensed data may be prepared and stored in external mediums for persistence or confidential information, e.g. patient data, may be encrypted previous to transmission.

2) Service structure: Components that implement any service in the WSN are provided with typed structure; which is inherited from the service meta-types. All services inherit from a meta-type for which all required and provided interfaces are mandatory. Services may not be extended by adding or modifying existing interfaces unless these changes are implemented at the meta-type level. According to their meta-types, all services inherit annotated attributes. These attributes offer the possibility of encoding runtime accessible semantic information in each service. 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 [10]. The annotated attributes selected for runtime modification by the adaptation interface are also enforced by the meta-type. E.g. sampling frequency attribute in SSCs is modified at runtime by the middleware based on battery level.

Additionally all SSCs must implement the Singleton pattern [12]. It is also required that timestamps are included for every sample of raw sensor data to improve data accuracy. Component coordination and interaction patterns are dictated by the underlying component model.

3) Enabling concurrency: Concurrent use of services in our framework is achieved through reuse of component instances. Given the intrinsic resource constraints in WSNs dealing with service contention through the replication of component instances is not an efficient approach; for this reason, we introduce a configuration meta-level on top of components. We separate a component’s functional code from its meta-data and share the same component instance across multiple service compositions (see Fig. 3). This meta-data contains the configuration semantics to be used in each composition in order to support the client required QoD.

Examples of meta-data for SSCs include corresponding request Id, sampling frequency and duration of service. In the DPCs one may use: request Id, parameter and source Id. The parameter is used by the DPC to parameterize its functionality, in the case of the averaging DPC this determines the time window for the average, i.e. average every 60min.

![Figure 3. Component meta-data](image)

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 client specified service request. The configuration semantics include client specified QoD, services involved in each composition and related parameterization.

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 because only one component instances is instantiated per service type for multiple requests.

B. The service request specification

Clients use the service request specification to express their QoD requirements in a per-service instance manner. 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. A client or application using the WSN may have multiple concurrent service-instances, e.g. sense temperature and humidity at warehouse X every 15 minutes for the next 3 days.

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 data processing service Id, which is globally unique identifier for services like averaging or specialized data filters. Every data processing service requested requires a parameter be specified for configuration, 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 data processing services. Optionally a status may be included to allow the middleware to customize parsing of the service request.

Listing 1: Service request format:

```c
serviceRequest#(requestId, serviceId, samplingfrequency,duration,targetLocation,DataProcessServiceId[], parameter[], status);
```

Per-service instance configuration allows multi-purpose WSNs to 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 [8]. 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].

C. Autonomic service composition

Service composition involves the definition of the processing order and configuration of service interaction in accordance to the client specified service request.

Valid service compositions: Service compositions can have only 1 SSC and zero-to-many DPCs (see Fig. 4). Multiple SSC are not allowed and all DPCs must be configured in sequence. Compositions must follow the pipe and filter pattern [11]. We extend the pattern to also allow for batch processing, where a component may consume all the data before producing an output; as opposed to only consuming and delivering data incrementally.

![Figure 4. Valid service compositions](image)

This definition appears rather simple but it is capable of representing a wide range of service compositions in multi-purpose WSNs. For example: i) sense temperature, ii) sense and average humidity iii) sense, average, encrypt light, iv) sense, filter, persist methane, V) sense, encrypt, reliably transmit temperature. As one may see this definition is capable of capturing important functional, data quality and cross-cutting concerns.

However, this definition does not cover the composition of composite services i.e. services that require multiple inputs. For example: assessing risk of fire; where temperature and light readings are used to calculate the probability of a fire starting in a given area. As one may see, this composition violates the definition because it has two source components providing input to a filter. We consider that these services should be addressed with the implementation of application specific components; which are considered as consumers or clients in our model. Logically one may assume that in turn they may be considered as services by other components or applications higher above the abstraction level. As is the case of the S&H tampering component introduced in the use case Section II-A.

The service composition process: it begins with the submission of a service request to the service management layer, which is accessible through a 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/client 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.
adaptation. Capacity planning ensures that only service requests are provided in Section IV-A.

2) Analyze composition: Parameters of each service requested are verified within the context of the requested composition. For example: one cannot average data samples within a time period smaller than the sampling frequency of the raw sensor data, i.e. you need at least two raw data samples per average interval. As one can see the validity of the parameter for the average service varies depending on the other requested services.

3) Evaluate providers: A potential set of candidate nodes is generated based on matching of target location and availability of required services. Service matching is done syntactically: given all services have a globally unique event Id. This event Id is generated according the event type hierarchy presented in [3] and assigned when each service is implemented. These candidates are evaluated given their currently offered data quality properties, battery level, node load, etc.

4) Select Provider: The evaluation made in the previous process guides the node selection strategy for the selection of service provider.

5) Capacity planning: Required indirect resources are calculated based on the configuration parameters in each service request. The corresponding resource reservations and allocations are fulfilled by the capacity planning functionality of our middleware platform. Further details on capacity planning are provided in Section IV-A.

6) Create Composition: The service management layer uses the configuration parameters extracted from the service request to configure the requested services, creating a service composition (see Section III-A). As one may recall, these services may include sensing services and data processing services.

D. Controlling resource use

Physical resources are exposed though the use of services. Services control the invocation of actual sensors, generation of data or use of any other underlying physical resource, e.g. memory, processor, network, etc. These services are guided by and controlled by the middleware. Clients can only submit their service request, where their usage requirements are specified, to the middleware but exert no direct control over any of the resources. For example a client may request temperature sampling every 10 seconds, this request may be accepted or rejected based on the maximum sampling frequency currently offered by the temperature service in the corresponding sensor node but the client has no control over this maximum.

Our middleware platform controls resource use with the use of two mechanisms: capacity planning and localized adaptation. Capacity planning ensures that only service invocations that are within current permissible usage parameters are allocated to be processed. The capacity planning process estimates the resources required to support a service request and checks availability of each required resource. Usage quotas per resource are used to specify how much of a given resource may be allocated for each activity.

Localized adaptation is an autonomic and independent process guided by adaptation strategies. These adaptation strategies are designed to evaluate how often a resource e.g. sensor, may be used under current system conditions and still maintain quality requirements. For instance, given a battery level of 25%, power hungry sensors may only be invoked once every 10 minutes. These strategies are evaluated locally at node level and directly modify component parameters that limit the use of each resource accordingly. As demonstrated in [10] controlling frequency of invocation of high power sensors significantly lengthens node lifetime. Additionally, the implementation of the singleton pattern [12] in all SSCs provides effective support for resource control.

IV. ADAPTABILITY POINTS IN OUR MIDDLEWARE

Proposed design principles for adaptive applications have steered application development to implement functionality in a modularized fashion such as inspired by component based engineering principles. In these approaches, formal interfaces are exposed to allow for component parameterization and the modification of functionality [20]. Of course finding the appropriate extent of modularization and determining its impact on performance are important issues. Furthermore, it is generally considered that modification of any modularized part of the application lies solely in the responsibility of a single operational role and that this single operational role has advanced knowledge of the hardware platform, execution platform, middleware and domain specific application software.

We have implemented our middleware functionality in a modularized fashion in such a way that modifications to these modularized portions may be offered to different operational roles and at different levels of abstraction. It is for this purpose that we separate what the middleware does, i.e. its functionality from how it does it, i.e. its behavior. The functionality is modularized with the use of components. The behavior is separated from functionality and evaluated within strategies. The locations in which strategies are called upon and evaluated within components are called adaptability points.

It is important to notice that the abstraction level at which modifications are made to components and strategies may vary significantly. The implementation and modification of components requires knowledge of the middleware, for example: the component model in use, coordination model and underlying execution platform. The implementation or modification of a strategy requires understanding of the different adaptability points available within the architecture and knowledge of how to express the desired behavior in a strategy. One may use event condition action semantics to express the desired behavior within a strategy.

Essentially, the logic that controls how primary functionality is executed, i.e. its behavior, has been
externalized through the use of strategies. These strategies are called upon during runtime to guide the execution of component functional code. For instance, every time a service request in a sensor node is received, the planning strategy is called upon to evaluate if there are enough resources available to support the request and to allocate resources if necessary. An adaptability point refers to a location where calls to strategies are made within the execution of functionality. The capacity planning component contains the corresponding variability point, where the planning strategy is called upon. We have augmented all the core middleware functionality with strategies to allow network administrators to enact behavioral adaptations without the need of advanced knowledge regarding the component implementation, underlying runtime environment or hardware platform.

It is important to note the clear distinction between the extension of functionality and the adaptation of its behavior. The former refers to adding new functionality, for example if we include support for component deployment in the middleware platform. The later refers to modifying the decision logic that guides the functionality, as per our need-for-change scenarios, where the current planning strategy will need to be changed to account for emergency situations by allocating emergency service invocations to the corresponding memory quota. Hence the decision logic that decides to grant resources or not for an invocation is modified but all the mechanisms that calculate requirements and later reserve them remain unchanged. It is for modification of behavior only that we have included these adaptability points in the architecture. We address extensibility through predefined plug-in locations in the architecture but these are outside the scope of this paper.

In the broader context of software engineering one may find direct resemblance between adaptability points and joint points in Aspect Oriented (AO) approaches. They both may indicate points in component code where an external construct is called upon to aid execution, aspects in AO and strategies in our approach. A similarity that is only relevant from an implementation perspective. At a conceptual level an aspect is fundamentally introduced to deal with cross-cutting concerns like logging or persistence. Their implementation, maintenance and deployments is considered to be undertaken by the same operational role and does not support a separation of operational concerns.

At a conceptual level, our work is more closely related to the levels of abstraction presented in the PERPOS middleware [16], as these too relay different abstraction levels to aid adaptation efforts. The use of strategies is certainly not new, as they are a subset of the broader policy concept. The main contribution of our works lies in that adaptability points are introduced to aid middleware evolution by allowing adaptation to be done at different abstraction levels and by different operational roles.

In this section we provide further details on the adaptability points corresponding to the capacity planning functionality and the corresponding views and different levels of abstraction mapped to their respective operational role. The different operational roles have been previously introduced in Section II, the application developer, the service developer and the network administrator.

A. Capacity planning process flow

Capacity planning functionality was designed to plan for and reserve consequential or indirect resources, thus avoiding consequential contention. Direct contention over a resource is currently managed through the use of low level mechanisms commonly offered by the underlying operating system (OS) or virtual machine (VM). For example, components A and B need access to a single light sensor and corresponding CPU cycles (see Fig. 6).

![Figure 6. Diagram depicting consequential and direct contention for resources.](image)

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.

Capacity planning starts after the service provider has been selected (see Fig. 5) and it consists of two main processes. Mainly, calculating resources needed and reservation of resources (see Fig. 7). These are initiated every time a service invocation is submitted to any service. These invocations for services are accompanied with all the relevant configuration parameters; which are used by the resource planner to calculate indirect resources required. In the case of SSCs these parameters include sampling frequency and duration of service.

![Figure 7. Sequence of processing steps required for capacity planning.](image)

1) Calculate resources: 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.

2) Reservation of resources: The resource reservation process is mainly comprised of two main sub-processes: and offline calculation of each component's resource use and a runtime capacity planning process. The offline process is executed by the service developer every time a new implementation for any service is developed and the runtime process is executed autonomously by the resource planner during system execution. The resource planner 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. The reservation is done on a first come first served basis.

Off-line, a Max usage quota for each memory medium should be defined. The network administrator should define what portion of available dynamic (RAM) memory should be used to support running services and what portion of static (flash) memory should be used to support persistence for the generated data. For example: if the total available amount of RAM is 10 Kb, allocate 50% to support running services. These memory usage quotas represent the 100% of dynamic and static memory that is available for the resource planner to allocate. Every reservation granted subtracts from the memory quota. Every resource that is released after service duration time expires adds back to the amount of memory available from the corresponding quota.

B. Sub-processes of the calculate resource process

This process consists of two sub-processes: generating a baseline of component resource use and runtime capacity planning.

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) Static memory requirements to store component code: record the memory required to store component executable code in flash memory.

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 and the creation and maintenance of the configuration meta-data (see Fig. 4). 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 service 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 Fig. 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).

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 a memory management strategy specifies that when battery level is under 10%, the persistence service stores all data sets from all running services to ensure no data-loss. 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. 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 plus the dynamic storage overhead. The Static memory required is equal to the output data set size plus the 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 plus the dynamic storage overhead plus the output data set size plus the dynamic storage overhead. Required static memory is equal to output data set size plus 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. We multiply the amount of records and the byte size of the data type, this gives
us the required memory for the output dataset.

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.

Figure 9. Runtime capacity planning and resource reservation

<table>
<thead>
<tr>
<th>Component Instance</th>
<th>Per component instance</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SSC</strong></td>
<td><strong>DPC</strong></td>
</tr>
<tr>
<td>runtimeStaticMem = output data set + staticStorageOverhead</td>
<td>runtimeDynamicMem = output data set + dynamicStorageOverhead</td>
</tr>
<tr>
<td>reservedStaticMem = requiredStaticMemory</td>
<td>reservedDynamicMem = requiredDynamicMemory</td>
</tr>
<tr>
<td>reservedDynamicMem = runtimeStaticMem</td>
<td>reservedDynamicMem = runtimeDynamicMem + dynamicMemoryAddService</td>
</tr>
</tbody>
</table>

A) Run-time capacity planning  
B) Resource reservation for SSCs and DPCs

C. Sub-processes of the reservation of resources process

The resource reservation process is comprised of two alternative sub-processes. Mainly, a reservation for each component instance and a calculation for each service request.

**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 memory 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.

D. Adaptability points in the capacity planning functionality

Adaptability mechanisms in our architecture are mapped to the corresponding operational role. We consider that given the scope of each operational role the appropriate level of abstraction varies. Therefore we offer three different levels of abstraction specifically designed for the scope of adaptation corresponding to each operational role (see Fig.10). In order from most to least abstract, these are: the client layer, the adaptation layer and the deployment layer.

1) **Client layer**: Application developers interact with the middleware API only; they are abstracted away from all underlying layers. However, they can use the API to access different hierarchical levels in the network architecture. They have access to the service management layer, which is present in the backend middleware and the cluster heads (see Fig.11). The API also gives them access to services on sensor nodes directly, configuring SSCs and DPCs directly. Even at node level they are abstracted away from the low level details. Even while abstracted away from middleware and platform details, the application developer may considerably customize the WSN and extend it using application specific functionality. The deployment of application specific components into the network allows for higher degrees of interaction, in-network processing and localized actuation. These components can easily exploit the WSN services using the configuration method calls offered for SSCs and DPCs in the node level API.

2) **Adaptation Layer**: Network administrators look at the multi-purpose WSN in terms of processes and available strategies where they may modify the behavior of middleware functionality to meet new business requirements (see Fig.12), as exemplified in Section II. Needless to say there is always an important balance to be maintained between the degree of modifiability an architecture offers and its runtime performance. Therefore adaptability points are only available for the primary functionality of our middleware. As is to be expected, some changes in business requirements will require middleware extension and structural adaptation, which are outside the scope of the network administrator’s responsibility.
3) The deployment layer: At this level of abstraction the service developers deal with platform specific details and implementing WSN services. It is at service development time that adaptability points are implemented. These adaptability points mark locations in the component code, where strategies are called upon and evaluated. Middleware extension and structural adaptation happens at this layer. Behavioral adaptation may also be executed at this layer by modifying all decision logic that is hard coded into the components or by modifying adaptability points.

V. IMPLEMENTATION AND EVALUATION

In this paper we presented a middleware architecture that provides strategy-controlled adaptability points to enable the modification of middleware behavior. We introduced a use case and need-for-change scenarios in Section II. We have discussed how modifying a strategy changes the middleware’s behavior thus modifying how it executes its functionality and, in this way, changes in business requirements may be effectively supported. To evaluate these capabilities we adapt the capacity planning functionality of our middleware; which we presented and evaluated in [6].

1) Previous prototype implementation to support the presented use case: In [6] we implemented and evaluated a prototype of our middleware platform that supports the use case presented in Section II. To provide the reader with some useful information regarding that evaluation, we describe it further.

In that evaluation we submitted a total of 20 service requests in 6 successive and overlapping batches to the SMC as depicted in Fig. 14, 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. 14 as they were submitted and Fig. 15 as they were actually processed. As one can see requests 6, 7, 13, 14 and 15 were rejected by the system.

Figures 16 and 17 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. 16 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. Our results demonstrate that 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.
In Fig. 17 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.

2) Supporting changes in business requirements through adaptability points in middleware architectures: As previously mentioned, in order to evaluate these capabilities we need to modify the planning strategy that controls the capacity planning adaptability point. This strategy was originally designed to allocate resources based on a first come first serve decision logic. In order to support the need-for-change scenario: Compliance with government regulations, the strategy needs to be modified to support resource allocation based on service request priorities.

Previously the planning strategy would reject any new request when the memory quota was full. If an emergency situation where to happen while the quota was full, it would have been rejected as any other request. We have now modified this strategy to allow emergency requests be allocated even after the service quota is full. We have implemented an emergency quota for emergency use reserved only for emergency situations, thus providing support for these even after the normal memory quota is full; in accordance with the need-for-change scenario.

Service requests are now parsed to determine if they are normal requests or emergency requests. The resource planner now proceeds with capacity planning accordingly, based on service request priorities.

Furthermore it is important to notice that as previously discussed, the modification of the planning strategy is within the responsibility of the network administrator. Modifications required to support this need for change scenario are enacted within the adaptation layer and achieved at the level of abstraction suitable to the expertise of a network administrator.

This demonstrates that changes in business requirements may be supported by network administrators at a convenient abstraction level. It also shows that the separation of operational concerns into corresponding roles and mapping adaptation responsibilities to each; is a feasible approach to evolving middleware behavior in the domain of multi-purpose WSNs.

As one may see in Fig. 18 we have submitted a total of 20 service requests in 6 successive and overlapping batches to the SMC, requests from 1 to 20, as we have previously done. Additionally, emergency requests 21 and 22 are submitted. 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. 18 as they were submitted and Fig. 19 as they were actually processed. As one can see, now emergency situations are effectively supported. At t4 in Fig. 19 one can see that even under full load condition emergency request 22 is supported and requests 13,14,15 are rejected.
Figures 20 and 21 illustrate RAM and FLASH usage during the processing of these requests. As one may see that for emergency request 22 at t4, the RAM usage has exceeded the MaxQuota, but since we have allocated additional memory space exclusively for emergency situations, the system may still effectively support the emergency request while rejecting a normal request.

As one may see, the planner still maintains allocation controlled and as before, every request that exceeds availability is rejected, while allowing emergency requests to be processed. In Fig. 21 one may see how the actual FLASH 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 memory management strategy 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 strategy may affect the resource reservation is outside the scope of this paper. All memory measurements provided are obtained by using the memory management facilities offered by the Sunspot API [14].

VI. CONCLUSION

In this paper we have presented a middleware architecture that provides strategy-controlled adaptability points; which are design to enable network administrators to enact behavioral adaptation in the middleware’s functionality in order to support changes in business requirements. We presented three operation roles that represent a separation of operational concerns. We have mapped each operational role to a different abstraction level corresponding to the scope of modification responsibilities of each role. To evaluate these capabilities we introduced a need-for-change scenario, which we have implemented and successfully supported.

In this manner, we have demonstrated that changes in business requirements may be supported by network administrators at a convenient abstraction level. It also shows that the separation of operational concerns into corresponding roles and mapping adaptation responsibilities to each; is a feasible approach to evolving middleware behavior in the domain of multi-purpose WSNs.

In our future work we plan to evaluate this functionality under an extended set of use cases that prove the applicability of the approach beyond logistics scenarios. We will prepare a hybrid testing infrastructure composed of a physical network and simulated environment to evaluate the efficiency of these strategies under high workloads and dynamic conditions.

REFERENCES

[14] SunSPOT: http://www.sunspotworld.com/, visited July 2010
Pedro Javier del Cid received an engineering in electronics degree from Universidad Francisco Marroquin, Guatemala in 2000. He received an advanced masters degree in artificial intelligence from Katholieke Universiteit Leuven in 2001. He is currently a researcher at IBBT-DistriNet and a PhD candidate in computer science at Katholieke Universiteit Leuven. His research interests include adaptive middleware for pervasive systems and architecture for software intensive systems.

Sam Michiels received a MSc in informatics and a PhD in computer science from Katholieke Universiteit Leuven in 1998 and 2003 respectively. He is currently Research Manager (Industrial Research Fellow, K.U.Leuven-IOF) within the DistriNet research group of the K.U.Leuven; this mandate enables him to focus on research, innovation and especially technology transfer in the domain of managing large-scale distributed software systems. His current activities target sensor networks and the design of adaptable middleware.

Wouter Joosen is full professor at the Department of Computer Science of the Katholieke Universiteit Leuven in Belgium, where he teaches courses on software architecture and component-based software engineering, distributed systems and the engineering of secure service platforms. His research interests are in aspect-oriented software development, focusing on software architecture and middleware, and in security aspects of software, including security in component frameworks and security architectures.

Danny Hughes received a Bs in computer science, a MSc in distributed interactive systems and a PhD in adaptive peer-to-peer systems at Lancaster University, England in 2001, 2002 and 2007 respectively. His research interests include: wireless sensor networks, middleware, embedded systems, distributed systems and peer-to-peer systems. He is currently a lecturer and faculty member of Xi’an Jiatong-Liverpool University, Suzhou, China and a free associate with Katholieke Universiteit of Leuven, Belgium.