Building Portable Middleware Services for Heterogeneous Cyber-Physical Systems

Kirill Mechitov
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL, USA
mechitov@illinois.edu

Gul Agha
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL, USA
agha@illinois.edu

***Abstract*—Software development in wireless sensor networks has traditionally focused on stand-alone applications statically linked with the operating system code, and relying on fixed models for scheduling, synchronization, and resource allocation. Middleware services and network protocols, are usually considered to be part of the operating system. As the number of available WSN platforms and operating systems grows, and the emergence of cyber-physical systems results in the creation of networks of hetrogeneous devices (sensor nodes, microcontrollers, mobile devices, etc.), portability and interoperation emerge as major considerations in the software development process. We propose breaking the tight integration between middleware services and the operating system. We demonstrate how adopting a service-oriented computing approach to WSN middleware services improves portability and enables the creation of heterogeneous sensor networks and cyber-physical systems. The adoption of a service-oriented architecture does not necessarily translate into a significant loss of performance. An extremely light-weight and flexible method for local and remote service interaction is proposed.**

***Keywords*-middleware, wireless sensor networks, service-oriented architecture**

I. Introduction

Current practice considers wireless sensor networks (WSNs) and cyber-physical systems (CPS) in the context of a single static application, e.g., a network for target tracking, environment monitoring, or structural control. This model of software development, together with the small scale of most experimental WSN deployments, has led to the design of middleware services that are highly efficient but often tightly coupled and customized to a particular application or operating system. This practice hampers service portability and reuse, such as when a data aggregation service is designed to work only with a specific routing protocol or a time synchronization serivce is tied to a particular radio driver.

Moreover, most WSN research has focused on networks made up of homogeneous devices—those using the same hardware platform and operating system. Unlike traditional wireless sensor networks, the CPS environment can be a large-scale distributed system comprising a mix of low-power embedded computing devices, sensing and actuation elements, networked mobile devices, and general-purpose computing and network platforms. One of the principal challenges of computer science research in cyber-physical systems is to find ways of creating scalable, robust, and efficient software capable of operating in this environment.

Application development is particularly challenging due to the lack of software engineering tools and programming languages commonly used in modern large-scale software development. Due to resource constraints and efficiency requirements, low-level C programming remains the dominant application development method in this domain [1]. Even small modifications to existing codebases currently require significant embedded software development skills and familiarity with the inner workings of the operating system to which the service is coupled.

As WSN and CPS deployments become more numerous and their scale increases, we envision these systems becoming an *open computing platform* used concurrently by multiple applications, or performing different and uncoordinated activities within the context of a single, complex application, while sharing a network of heterogeneous devices. Highly efficient middleware services customized to each application or integrated with the low-level device drivers become less attractive, as common functionality is needlessly replicated, and incompatibilities between different versions of the same service can arise. In our view, these requirements imply the need for a software architecture that provides a looser coupling between middleware, OS, and applications, and among the services themselves. This must still be accomplished in a resource-efficient manner suitable to low-power embedded devices.

In this paper, we consider applications that make use of a number of general network-wide middleware services such as routing, localization, and time synchronization, in a cyber-physical system comprised of heterogeneous embedded devices. In order to accommodate the vast collection of services and protocols already developed by the embedded systems and WSN communities, we adopt a very broad definition of what constitutes a middleware service, concerning ourselves only with their interfaces to applications, operating systems, and other services, and not their internal semantics or implementation method. Specifically, we propose a service composition-based architecture for networked embedded systems, based on the principles of

978-1-4673-1793-1/12/$31.00 © 2012 IEEE

SESENA 2012, Zurich, Switzerland

service-oriented computing (SOC), with the goals of facilitating large-scale application development, fostering greater portability and reuse, and enabling global, network-wide optimization rather than application- and OS-specific local optimization, of portable middleware services.

By dissociating middleware services from the application context and from the underlying operating system, we give up some possible performance advantages due to explicit customization and tight coupling, although by applying certain design and implementation techiques, the overhead costs can be minimial. In return, we provide a more scalable software development process, support for multiple concurrent applications, and the possibility of developing portable middleware and applications across a range of embedded computing platforms.

This research is motivated in large part by the authors' experience with the development of a modular application framework for the Illinois Structural Health Monitoring Project [2], as well as the installation and maintenance of over 100 sensors in the course of a three year long structural health monitoring (SHM) deployment [3]. The same codebase was later adapted to real-time monitoring and structural control. Due to the size and complexity of the application and middleware code in this project, issues of programmability, adaptivity, and fault tolerance of dominated the development process, as opposed to the more traditional WSN considerations of energy efficiency and low-level performance optimizaiton.

The remainder of the paper is organized as follows. First, we motivate the need for this software development approach with a concrete example in Section II. Next, we briefly review related work on service-oriented architecture in Section III, and propose a service-oriented architecture for portable WSN middleware in Sections IV. Section V then describes a dynamic service composition-based architecture implementing these principles. Section VI concludes the paper.

II. Motivation

Let us identify precisely the issues with the current approach to developing a portable middleware service for a network of heterogeneous devices, and examine an alternative approach. Consider as an example an implementation of a multi-modal data acquisition service in a heterogeneous cyber-physical system consisting of three types of devices: 1) low-power sensors, 2) more powerful and fully-featured sensor nodes, and 3) sensor-actuator hybrid nodes acting as embedded controllers. A specific instance of such a CPS would be a wireless network for structural health monitoring and control of civil infrastructure such as buildings and bridges (Figure 1). Its operation involves periodic collection of data from a multitude of sensors of different modalities and at different timescales. Once collected, the data can be processed and transferred among the nodes for a variety

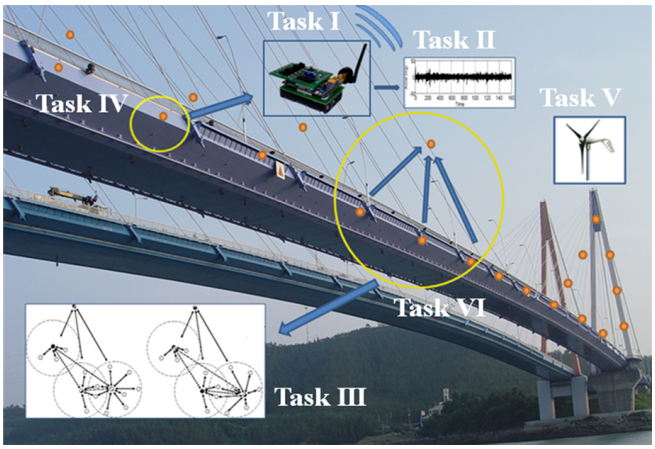

Figure 1. A heterogeneous CPS for structural monitoring and control, with several concurrently executing independent tasks: sensing, distributed processing, hierarchical data aggregation, and energy harvesting.

of purposes: long-term monitoring, damage detection, and active damping (structural control).

As stand-alone components, low-power sensors used for acquiring data at low sampling rates (e.g., temperature, humidity, strain) are typically running on "bare metal" microcontrollers, without the support of an operating system. They are highly tailored to their specific functionality and can be extremely efficient. Fully-featured general purpose sensor platforms, such as Imote2 or Telos motes, most often make use of a WSN-specific, event driven operating system such as TinyOS, SOS, or Contiki. Embedded sensor-actuator devices, due to the hard real-time requirements of control algorithms, normally rely on an embedded RTOS such as FreeRTOS or Keil RTX.

Thanks to the standardization of low-power radios on the IEEE 802.15.4 specification, all of these devices can be part of the same wireless network and exchange packets with each other with little difficulty. However, at the the task scheduling and device driver level, these three systems differ greatly, and would share little code in common.

When middleware services, such as the data acquisition service, are tightly integrated with the operating system, as is commonly the case today, porting them between platforms with different operating systems is a significant undertaking. On the other hand, if the core logic of the service is implemented to well-defined application- and OS-level interfaces, only comparatively small changes to the driver interface are necessary to port the service between these platforms.

In our example, the data acquisition service serves two purposes: to collect sensor data and possibly some meta-information (timestamps, sensor channel or modality, sampling frequency, scaling factors, etc.), and to make the data available upon request at any later time to any node in the network. If the data acquisition service is tightly integrated

```
typedef struct {
    uint32_t size;      // number of elements in 'data'
    float scale;        // ADC scale
    float offset;       // ADC offset
    float freq;         // sampling frequency
    int16_t *data;      // raw ADC data
} channel_data_t;

typedef struct {
    uint32_t size;      // number of elements in 'data'
    uint32_t block;     // samples per timestamped block
    uint32_t high32;    // high 32 bits of 1st timestamp
    uint32_t *low32;    // low 32 bits of timestamps
} timestamp_data_t;

typedef struct {
    uint32_t node;
    channel_data_t channels[MAX_SENSOR_CHANNELS];
    timestamp_data_t timestamps;
} sensor_data_t;
```

Figure 2. Example of a portable, self-describing data structure in C for storing multi-modal sensor data. The data structure contains descriptive meta-data (origin node, sampling rate, scaling factors) alongside the sensor values and timestamps.

with the operating system, the former functionality is likely to be a part of the sensor driver interface for specific sensors. If it is tightly integrated with the application, the latter functionality is likely to be mixed with application logic.

Let us consider an alternative implementation of the data acquisition service, with well-defined interfaces for interaction with the application and the OS. The design of the service centers around a portable and space-efficient data structure for storing and transporting the sensor data and meta-information (Figure 2). This data structure must store the sensor data itself as well as additional semantic data describing where and how the sensor readings were acquired.

An example of the extra information contained in the data structure would be a flexible and efficient timestamping method: storing a timestamp for every k samples. If the sampling rate is known to be constant, only the first timestamp is necessary to fully define the data acquisition timing, $k = 1$. If the sampling rate is highly irregular, each sample must be timestamped $k = N$, where N is the total number of samples. If the sampling rate (or the clock) is subject to drift periodic timestamps every several samples are needed to compensate for this, $1 < k < N$.

The external interface to the service then comprises four methods:

1) Setting up data acquisition: start time, number of samples, frequency, sensor channel, etc.
2) Low-level sensor access.
3) Notification of data availability.
4) Local or remote access to the acquired data.

Only item 2 is likely to be highly platform-dependent, as sensor interfaces vary a great deal between sensor platforms, operating systems, and sensor types. An event-driven OS and an RTOS with support for task priorities and blocking would have very different implementations of a driver for the same sensor. Additionally, sensor interfaces can differ a great deal from each other. For instance, some analog-to-digital converters (ADCs) require manually requesting each sample, such that the CPU and the operating system timers are responsible for the sampling rate. Others have internal clocks that control sample acquisition timing, and only trigger an interrupt for the OS to handle when a sample or a block of samples is available. This part of the service would have to be re-implemented specifically for each platform. However, the other parts of the interface are largely or completely platform-independent. Applications and other middleware services can be written to the semantics of these interfaces, even if the specific syntax of service invocation differs from platform to platform. The sensor data structure acts as an abstraction layer between the two phases of the service: data generation (filling in the structure) and data access.

We consider *well-defined interfaces* and *self-describing data structures* containing semantic metadata to be the keys to the creation of highly portable middleware in wireless sensor networks. In our experience, service interaction through such interfaces and data structures does not constitute a significant fraction of the system's overall resource usage, even on low-power sensor nodes. Efficiency of low-level device drivers and application-level data processing algorithms dominate performance and energy consumption of the system. With this in mind, service-oriented architecture becomes a sensible option for CPS software development.

In the next section, briefly review the principles of service-oriented architecture, and then propose a lightweight SOA for portable WSN middleware services, which can provide this type of functionality with minimal overhead.

III. Service-Oriented Architecture

With the exponential growth in available computing power over the last 50 years, the complexity of computer software has likewise increased dramatically. Advances in the fields of programming language design and software engineering allow application developers to deal with this complexity by dividing the software system into smaller, manageable parts. Notably, *object-oriented programming*, which encapsulates data together with the methods used to operate on it, and *component-based software architecture*, which proposes building applications as a composition of self-contained computing components, have been instrumental to the design and development of large-scale software systems. Expanding on this idea, *service-oriented architecture* (SOA) has recently been proposed as a way to bring this design philosophy to building dynamic, heterogeneous distributed applications spanning the Internet [4], [5], [6].

Services, in SOA terminology, are self-describing software components in an open distributed system. The description of a service, called a contract or an interface, lists its inputs and outputs, explains the provided functionality, and describes non-functional aspects of execution [7]. Different applications can be built from the same set of

services depending on how they are linked and on the execution context [8]. This approach makes for dynamic, highly adaptive services and applications that can be ported between platforms with relative ease, without the need to revisit and adapt the logic of each service for a particular application/OS combination.

SOA design principles apply in the cyber-physical systems context as well as on the Internet. Such systems often consist of numerous independent nodes, each an embedded computing platform with a processor, memory, and a radio transmitter. As such, CPS applications are by definition distributed and thus require communication and coordination for parts of the application running on different nodes. SOA has been proposed to address the inherent problems in designing complex and dynamic CPS applications [9]. Building an application from a set of well-defined services moves much of the complexity associated with embedded distributed computing to the underlying middleware. This approach also fosters reuse and adaptability, as services for a given application domain can be employed by a multitude of applications.

Perhaps more importantly, SOA provides for a *separation of concerns* in application development. That is, application designers can focus on the high-level logic of their application, service programmers can concentrate on the implementation of the services in their application domain, and systems programmers can provide middleware services (reliable communication, time synchronization, data aggregation, etc.) that enable the services to interact for a particular platform. In cyber-physical systems, which are often tailored to application- and context-specific requirements, it especially important for the high-level design of the application and the domain-specific algorithms used by the services to be separated from the low-level infrastructure necessary to make the system work.

IV. Architecture Overview

Our proposed architecture leverages the concept of dynamic service composition to support portable application and middleware development for open WSN systems. We adopt a two-level architecture, separating the two major concerns: that of controlling the execution process, including strategic decision making and adaptation, and that of the execution itself. First, we restate our formulation of the problem more formally.

We consider applications specified in terms of a composition of calls to middleware service interfaces, and we refer to the service interface specification as a *contract* and each invocation of a service a *service request*. To facilitate the use of a large number of pre-existing middleware services and network protocols within our architecture, we choose not to constrain the model of a service's behavior, e.g., whether it is distributed, centralized, single-threaded, etc. Since services and applications need to interact and coordinate, however,

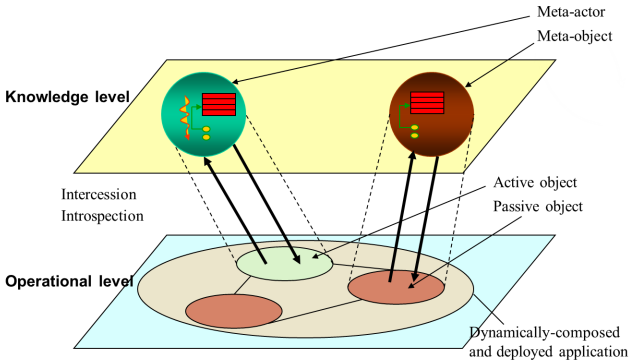

Figure 3. Two-level architecture for controlling active objects. Execution control, coordination, and quality of service considerations are separated from base-level operational functionality.

we fix a model for their interaction. We use the Actor model of computation [10] to represent service interfaces connecting services to each other and to the application. Thus, services are used by our system as if they were implemented as *actors*: concurrent active objects interacting via asynchronous message passing. We distinguish between the actors representing the service itself from *meta-actors*, which supervise deployment and execution of the services, providing for service configuration, adaptation, and non-functional requirements (Figure 3).

Responsibilities of the meta-actor include controlling the lifecycle of a service (instantiation, starting, stopping and disposing of the service) and interaction with other services. Note that once the appropriate services are instantiated, certain interactions can happen directly between base-level actors, rather than through their corresponding meta-actors. Such interaction then occurs through the actor interface specified in the service contract. Only interactions through actor interfaces are mediated by our architecture; any side effects are not captured by this model.

We further assume the existence of a functional service composition language, where service requests are *self-sufficient* and *minimally constrained*. The service composition language is functional in that (1) the control flow between service requests is partially ordered and driven by data dependencies, and (2) it allows for a recursive graph traversal to autonomously process each service request in the specification. Self-sufficiency refers to the fact that each individual service request is provided with the required knowledge about the arguments, resources, context and method required for its execution. Minimally constrained refers to delaying as long as possible placing constraints necessary to execute a specific instance of the service, in other words, the service instance does not refer to information that can be computed or supplied to it at run-time.

We strive for an extremely lightweight, low overhead implementation of SOA. For local interactions, the cost of

a service invocation should not be significantly more than just one function call. For remote interactions, the remote method invocation should be implemented to piggyback on normal data transfers that would occur in any case to provide the base functionality, adding only minimal overhead to the transmission. It should also be noted that most services we consider for this framework provide high level distributed middleware or application-level functionality (e.g., data aggregation, time synchronization, multi-hop routing, etc.), with more low-level OS tasks (timers and interrupts, memory management, scheduling) being left to the underlying system for the sake of efficiency.

V. SOA Implementation

We propose a dynamic service composition framework as an extension of an existing static component-based system. A collection of cutomizable middleware services and data processing components has been developed as part of the Illinois SHM Project [3], [11] for the structural health monitoring application domain. This customizable framework for building SHM has attracted a community of computer scientists and civil engineers from over 70 research groups [2], resulting in the development of numerous SHM algorithms, middleware components, and network protocols. Complete applications can be assembled by customizing and statically linking these parametrized components via their nesC interfaces [1]. In order to replace the static linking of the components with a meta-actor based SOA implementation, we need to introduce a flexible and lightweight service invocation mechnism.

Given an application comprising a composition of middleware service requests represented as outlined in the previous section, its execution consists of a mutually recursive service instantiation and invocation process. This results in a system of distributed interacting meta-actors responsible for handling the interaction among the services. Execution proceeds concurrently and asynchronously as the preconditions for the invocation of each service request are satisfied. We call this process *self-mediated execution*.

Let us now focus on the role of the meta-actors in this process. Fig. 4 highlights the governing behavior of a meta-actor in processing service requests. These meta-objects are dynamic, they have the capability to observe the base-level actors and the environment (*introspection*), and to customize their own behavior by analyzing these observations (*intercession*), as seen in Figure 3.

Due to service request self-sufficiency, each meta-actor can decide *how*, *where* and *when* to execute its associated service functionality. We now explain the function of each component of this architecture and their interactions.

Deciding *how* to execute the service request involves matching a service instance to a suitable service interface, and then finding the resources required by that service.

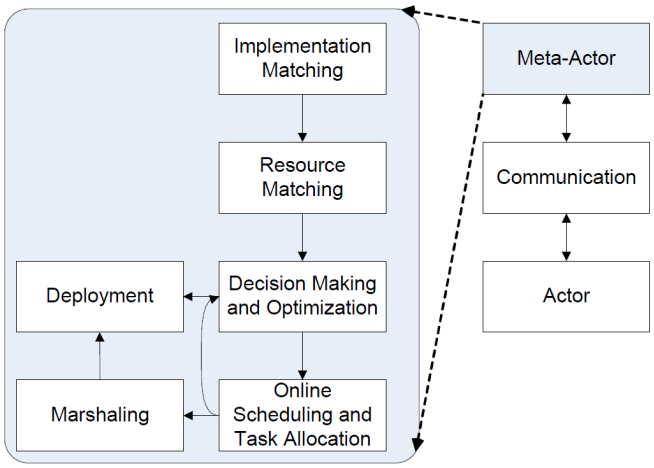

Figure 4. Self-mediated execution architecture for middleware services. The meta-actor acts as a remote governor of the base-level actor.

This component finds all service instances that match the constraints of a given service request. This is done by querying the contract repository and filtering the results according to the constraints specified in the service request.

Likewise, the resource matching component finds all suitable resources (sensors, data stores, etc.) for a given service instance. Matching algorithms used by this service depend on the resource description language employed by the system. Several methods are available for indexing a dynamic set of geographically distributed resources, including a yellow pages service, tuple spaces and actor spaces. Caching and prefetching techniques can make the process more efficient, eliminating the need to scour the network for each query. Due to the location-dependent nature of most WSN computations, we expect most queries to be limited geographically, in most cases to the same node, avoiding the need to flood the network even in cases when cached information is unavailable.

Instantiation and Invocation: The meta-actor also needs to decide *where* to execute the service request. For the sake of efficiency, instantiation and invocation are treated separately. As such, service instantiation—potentially a long process—can start early, while the invocation is delayed by the scheduling component until the necessary resources become available. Given a list of possible resources and implementations, this component chooses which implementation/resource combination best fits the application requirements or system performance considerations. The output of this service is a platform-specific executable code segment, along with a list of its required resources, which dictate where in the WSN the service must be located. This component comprises the core of the self-mediated execution approach. Choosing an appropriate option from a list of resources and service implementations is critical to efficiently executing composite service-based applications.

Scheduling: Next the meta-actor decides *when* to execute the service request. This is accomplished by the scheduling and task allocation component. The goal of this component is to decide when the service instance can be deployed and executed. If the resources required by the service instance are not immediately available, its execution is postponed, along with all services that depend on it. Shared resources requiring exclusive access, *e.g.*, certain types of sensors and actuators, must be scheduled globally, since service implementations may not be aware of each other. An up-to-date resource use schedule is provided to the decision making and optimization component to facilitate the selection of less-utilized resources whenever possible, and a repository of active services is maintained to keep track of all service instances currently deployed in the system. This is also used by the implementation matching component to check if an already-deployed component may satisfy a service request.

Argument and Output Handling: Finally, the service request is ready to be invoked on the target service. This step includes marshaling and remote invocation. The marshaling component packages the service request for transport and execution on the destination platform, using the invocation component. The method is platform-dependent. In our system, this involves wrapping the service invocation parameters together with the raw argument data to be transferred. In the case of service requests with no or small-sized arguments, the entire request data can fit into a single network packet. The service request is then handed off directly to the base level actor interface to invoke the selected service instance. From this point onward, the service instance interfaces via its actor interface with its meta-actor and with other services in the system by means of asynchronous message passing. Asynchronous messaging is used both to deliver computation results and error notifications from the executing services and to deliver control messages from the meta-actor. In the case of both sender and receiver residing on the same node, the message passing can be short-circuited as simply a function call by means of an optimization techinque such as a continuation passing style (CPS) transform.

VI. Conclusion

Our research aims to improve programmability of complex cyber-physical systems by separating context-independent application logic and platform-independent functionality of middleware services from the low-level implementation details that are specific to a particular platform or operating system. The means to accomplish this is a dynamic and lightweight service-oriented architecture for portable WSN middleware services. We believe that the design principles and architecture defined in this paper will lead to greater portability and code reuse in many sensor network and cyber-physical system services and protocols. The success of this approach depends on the possibility to efficiently implement the service interfaces and data interchange formats on low-power embedded systems. We consider performance optimization of the service-oriented architecture for WSN middleware to be an important open topic for future research.

Acknowledgment

The authors gratefully acknowledge the support of this research by the National Science Foundation, under grants CMS 06-00433 and CNS 10-35773. This research was conducted as part of the Illinois Structural Health Monitoring Project, led by Professor Gul Agha (Department of Computer Science) and Professor Bill Spencer (Department of Civil and Environmental Engineering) of the University of Illinois at Urbana-Champaign.

References

[1] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler, "The nesC language: A holistic approach to networked embedded systems," *ACM SIGPLAN Notices*, vol. 38, no. 5, pp. 1–11, 2003.

[2] Illinois Structural Health Monitoring Project, http://shm.cs.illinois.edu/.

[3] J. Rice, K. Mechitov, S.-H. Sim, B. F. Spencer, and G. Agha, "Enabling framework for structural health monitoring using smart sensors," *Structural Control and Health Monitoring*, 2010. [Online]. Available: http://dx.doi.org/10.1002/stc.386

[4] M. Singh and M. Huhns, *Service-Oriented Computing: Semantics, Processes, Agents*. John Wiley and Sons, 2005.

[5] W. Tsai, "Service-oriented system engineering: A new paradigm," in *IEEE International Workshop on Service-Oriented Systems Engineering*, 2005, pp. 3–8.

[6] M. P. Papazoglou, P. Traverso, S. Dustdar, F. Leymann, and B. J. Krämer, "Service-oriented computing: A research roadmap," in *Service Oriented Computing (SOC)*, F. Cubera, B. J. Krämer, and M. P. Papazoglou, Eds., no. 05462, 2006.

[7] B. J. Krämer, "Component meets service: what does the mongrel look like?" *ISSE*, vol. 4, no. 4, pp. 385–394, 2008.

[8] T. Gu, H. Pung, and D. Zhang, "A service-oriented middleware for building context-aware services," *Journal of Network and Computer Applications*, vol. 28, no. 1, pp. 1–18, 2005.

[9] K. Mechitov, R. Razavi, and G. Agha, "Architecture design principles to support adaptive service orchestration in WSN applications," *ACM SIGBED Review*, vol. 4, no. 3, 2007.

[10] G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[11] T. Nagayama, B. F. Spencer, K. Mechitov, and G. Agha, "Middleware services for structural health monitoring using smart sensors," *Smart Structures and Systems*, vol. 5, no. 2, 2008.