An Architecture for Dynamic Service-Oriented Computing in Networked Embedded Systems

Kirill Mechitov and Gul Agha

Department of Computer Science
University of Illinois at Urbana-Champaign
201 North Goodwin Avenue Urbana, IL, USA 61801
{mechitov,agha}@illinois.edu

Abstract. Software development in real-time and embedded systems has traditionally focused on stand-alone applications with static models for scheduling and resource allocation. Our goal is to facilitate the development of embedded applications in an open system, where tasks and resources arrive and leave dynamically, and their execution is concurrent. We model such applications as a dynamic composition of network services. This paper presents an enabling framework for dynamic service orchestration in cyber-physical systems, based on a modular, reusable, and extensible service-oriented architecture. By taking advantage of a network-wide programming model, adaptive global resource management, and late binding of tasks to resources, the architecture enables execution of dynamic embedded application workloads in a resource-efficient manner.

Keywords: networked embedded systems, open systems, service-oriented architecture.

1 Introduction

The typical cyber-physical system (CPS) environment is a large-scale distributed system comprising a mix of low-power embedded computing devices, sensing and actuation elements, networked mobile devices, and traditional computing and network platforms. One of the principal challenges of computer science research in networked embedded systems is to find ways of creating scalable, robust, and efficient software capable of operating in this environment.

Current practice considers CPS in the context of a single application, e.g., a system for target tracking, environment monitoring, or structural control. This model of application development, together with the small scale of most experimental networked embedded system deployments, has led to the design of middleware services that are highly efficient but often tightly coupled or customized to a particular application. This practice hampers service portability and reusability, such as when a data aggregation service is designed to work only with a specific routing protocol.

M. Heisel (Ed.): Krämer Festschrift, LNCS 7365, pp. 147–164, 2012.

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 [5]. Even small modifications to existing codebases require significant programming skill and embedded systems experience.

Some recent work has proposed supporting several concurrent applications in a sensor network [19]. As CPS deployments become more numerous and their scale increases, we envision these networked embedded systems becoming an *open computing platform* used concurrently by multiple users and multiple applications for different and uncoordinated activities. For instance, as illustrated in Fig. 1, middleware services should be shared among unrelated applications. In this context, efficient customized middleware services specific to each application are a poor solution, as common functionality is needlessly replicated.

In our view, these requirements imply the need for a software architecture that provides a looser coupling between services and applications, and among the services themselves, in a resource-efficient and context-aware manner. We consider applications that make use of a number of general network-wide middleware services, such as routing, localization, and time synchronization. In order to accommodate the vast collection of services and protocols already developed by the embedded and cyber-physical systems communities, we adopt a very broad definition of a middleware service, concerning ourselves only with their interfaces to applications and other services, and not their internal semantics or implementation method. Specifically, we propose a dynamic service composition-based architecture for networked embedded systems, based on the principle of *self-mediated execution*, with the dual goals of facilitating large-scale application development and enabling global, network-wide optimization, rather than application-focused local optimization of the constituent middleware services. Our approach is based on postponing the binding of applications to specific network resources and implementations of middleware services from design- or compile-time to the runtime. Appropriate service implementations are chosen at runtime and deployed on demand.

By dissociating middleware services from the application context and from each other, we give up some possible performance advantages due to explicit customization and tight coupling. In return, we provide a more scalable software development process, support for multiple concurrent applications, and the possibility of global resource management across applications.

The remainder of the paper is organized as follows. First, we review related work on service-oriented architecture in Section 2. Section 3 states our overall design principles. Sections 4 and 5 then describe a dynamic service composition-based architecture implementing these principles and Section 7 illustrates its use in the context of a target tracking application, and Section 8 discusses the properties of our architecture.

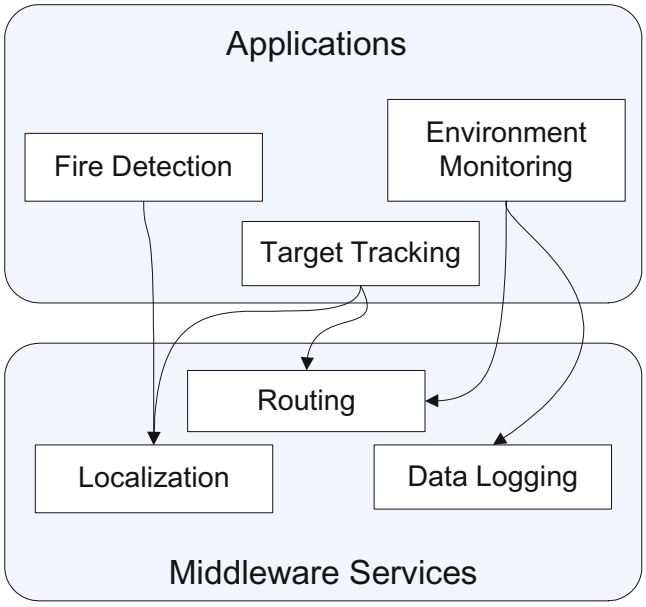

Fig. 1. Middleware services are shared among concurrently executing applications, resulting in a many-to-many relationship between applications and services

2 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 engineering*, 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 [13, 16, 18].

Services, in SOA terminology, are self-describing software components with well-defined interfaces in an open distributed system. The description of a service, called an interface or a contract, lists its inputs and outputs, explains the provided functionality, and describes non-functional aspects of execution. Service interfaces, unlike those of components, are generally location- and implementation-agnostic. Krämer has proposed a merger of component- and service-oriented software development as a *service component architecture* (SCA) [8]. We follow a similar approach in our work, defining applications as a composition of service components.

Different applications can be built from the same set of services depending on how they are linked and on the execution context [7]. This approach makes for dynamic, highly adaptive applications without the need to revisit and adapt the implementation of each service for a particular application context.

2.1 SOA in Cyber-Physical Systems

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 [11, 12]. 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. 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.

2.2 Dynamic Execution in Cyber-Physical Systems

Implementing service-oriented architecture requires as a basis the ability to dynamically reconfigure and adjust the behavior of a running system. A dynamic code execution framework is thus necessary to accomplish this. Traditionally, real-time and embedded systems favored static, compiled models due to their deterministic properties. In recent years, however, several platforms for dynamic execution in networked embedded systems have been proposed.

The Melete system provides a method for concurrently executing uncoordinated applications in a wireless sensor network (WSN) [19]. Melete applications are written in the TinyScript language and executed by a virtual machine on an arbitrary subset of nodes in the network. In contrast to this approach, we propose a more comprehensive method of executing concurrent applications, which allows global resource management and a higher level of optimization. In fact, Melete may be used as part of our architecture, acting as the code deployment method for service instances.

The Tenet architecture enables service composition for multi-tiered applications incorporating WSNs [6]. Most of the coordination and processing functionality is relegated to more powerful tiers, while the WSN nodes are used primarily to retrieve sensor data. Our approach differs in that we treat the system as a collaborative distributed computing platform. By associating asynchronously interacting, autonomous actors to service instances on sensor nodes we make possible *in situ* collaborative problem solving.

The SONGS architecture and programming model considers sensor network applications as a composition of semantic services [11]. Semantic services are a type of semantic data transformation functions, and do not correspond to what we call services in this paper. We are interested in facilitating composition of less structured infrastructure and middleware services, a vast quantity of which has already been developed for cyber-physical systems.

ActorNet [9] is a mobile agent platform for sensor networks, designed to support multiagent applications on these resource-limited, real-time systems. ActorNet agents, are called actors, and are based on the actor model of computation [1]: concurrent active objects communicating via asynchronous message passing. They are specified in a relatively high-level interpreted language based on Scheme, which allows highly dynamic, mobile code to be executed across multiple sensor nodes. The ActorNet runtime environment provides all the underlying functionality necessary to support the interpreter for mobile code on sensor network platforms: memory management, scheduling, communication and migration. These features enable the execution of complex, highly dynamic mobile agent applications in the severely resource-constrained environment of wireless sensor networks. We employ ActorNet as the code deployment framework for dynamic macroprogramming, since Scheme-like code is very easy to generate automatically based on a behavior template.

Agilla [4] is another WSN mobile agent platform, and is in many respects similar to ActorNet. The principal difference is in the trade-off between power and expressiveness of the actor language versus efficiency. Agilla agents are based on virtual machine code, which is considerably more compact than ActorNet's Scheme representation. For the same reason, however, Agilla agents are not as flexible or capable as their ActorNet counterparts.

3 Design Principles

Service- and component-based architectures are widely used, providing greater ease and scalability to the software design and implementation process. We aim to apply the same approach to the cyber-physical systems domain, adapting to its unique limitations and requirements. We identify the following key principles for the design of scalable, resource-efficient WSN applications as a composition of middleware services:

1. *Network-wide Programming Model.* The networked embedded system is treated as a collaborative distributed computing platform. Applications are specified as a collection of network-wide tasks and not as a unique program image per embedded node.
2. *Sharing and Reuse.* Multiple uncoordinated applications and middleware services need to coexist in the network without prior knowledge of each other. Therefore, both network resources requiring exclusive access (sensors, actuators, etc.) and middleware services are shared among several applications. Resource management cannot be relegated to each application individually, it must be performed globally.
3. *Late Binding.* Application specification is sufficiently flexible to allow runtime adaptivity in selecting the services and resources to be used. We do not know in advance which services or resources will be used by which application, or when. Postponing the choice of which service or resource best fits the application opens up more opportunities for optimization.

In the following section we present a service composition-based software architecture that follows from these design principles.

4 Architecture Overview

Our architecture leverages the concept of dynamic service composition to support application 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 assumptions about 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 call to a service a *service request*. A repository of available services for a given CPS or application domain is provided.

To facilitate the use of a large number of pre-existing middleware services 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, we fix a model for their interaction. We use the Actor model of computation [1] 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 messaging. We distinguish between the actors representing the service itself from *meta-actors*, which are control threads supervising deployment and execution of the services.

Responsibilities of the meta-actor include controlling the lifecycle of a service (deploying, starting, stopping and disposing of the service) and interaction with other services. Note that once the appropriate services are deployed, they may choose to interact directly, rather than through their corresponding

meta-actors. 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.

The last requirement is a fine-grained dynamic code execution method, such as a mobile agent system like ActorNet [9] or Agilla [4].

Consider how a typical localization service request is represented in our architecture. To be self-sufficient, the contract includes a reference its execution method, e.g., a compiled library implementation of a localization algorithm, the type of sensors used, such as distance measuring or angle of arrival, and data types for the output (locations and error intervals). To be minimally constrained, it must not specify a deployment location (node addresses or coordinates) or method (a specific range measurement service), referring instead to the contracts in the repository. Execution-specific information is filled in at run-time based on the specified constraints.

5 Architecture Components

Given an application comprising a composition of middleware service requests represented in such manner, its execution consists of a self-decomposition and self-deployment 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 deployment 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. 2 highlights the governing behavior of a meta-actor in processing service requests. These meta-objects are dynamic, they have the capability to observe the application objects 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. We now explain the function of each component of this architecture and their interactions.

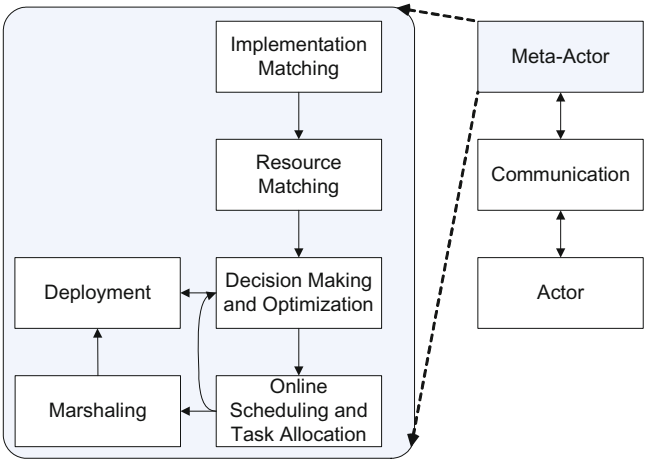

Fig. 2. Self-mediated execution architecture for middleware services

5.1 Choice of Implementation

Deciding *how* to execute the service request involves matching a particular service implementation to an interface from the service contract repository, and then finding the network resources required by that implementation.

Implementation Matching. This component finds all implementations that match the constraints of a given service request. For example, we might search for implementations of a ranging service with a MeasureDistance method that also satisfy a maximum distance constraint. This is done by querying the contract repository and filtering the results according to the constraints specified in the service request. Pattern matching or a linear constraint solver may be used to filter the available service implementations.

Resource Matching. Likewise, the resource matching component finds all suitable resources for a given service implementation. 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. For instance, if tuple spaces are used, sensor nodes entering the system can publish their resource descriptions in the tuple space, and the resource matching component performs a search in the form of pattern matching [2]. 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, avoiding the need to flood the network even in cases when cached information is unavailable.

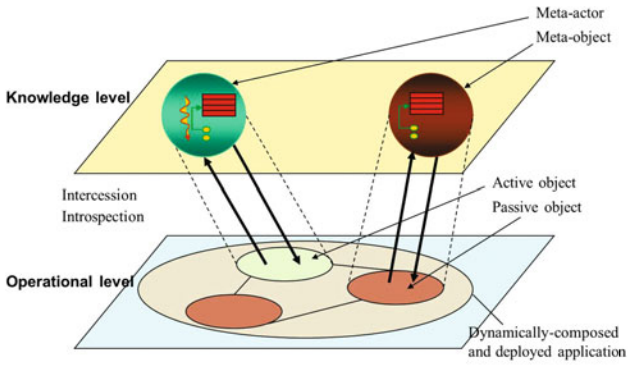

Fig. 3. Two-level architecture for controlling active objects

Location and Deployment. Second, the meta-actor needs to decide *where* to execute the service request. For the sake of efficiency, deployment and invocation are treated separately. As such, code deployment starts as soon as possible, while the invocation is delayed by the scheduling component until the necessary resources become available.

Decision Making and Optimization. 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.

Deployment. This component is responsible for transporting the executable code segment to the destination platform, thereby making the service available to other services and applications. If an implementation of the service is already available at the destination platform, the code deployment step is skipped entirely, and the service request is sent to the deployed service.

Scheduling. Third, the meta-actor decides *when* to execute the service request. This is accomplished by the scheduling and task allocation component.

Online Scheduling and Task Allocation. 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.

Invocation and Execution. Finally, the service request is ready to be deployed and executed on the target platform. This step includes marshaling and remote invocation.

Marshaling. The marshaling component packages the service request for transport and deployment on the destination platform, using the deployment component. The method is platform-dependent. In our system, this involves wrapping the service invocation code in a mobile agent, which can move to the destination node without relying on an external routing service.

The service request is then handed off directly to the run-time environment to launch or query the selected implementation of the service. From this point onward, the service instance interfaces via its actor interface with its meta-actor and with other services in the CPS by means of asynchronous message passing, implemented by the communication component (Figure 4). 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.

6 Mobile Code Deployment

Finally, we consider the operational level of the architecture, where the actual interaction with the cyber-physical system takes place. We employ ActorNet [9], a mobile agent platform for wireless sensor networks, as the mobile code deployment platform. There are several reasons why ActorNet fits well into this role.

Cyber-physical systems are well-suited to the multiagent approach: agent autonomy reduces the need for communication, saving precious energy. Mobile agents are also an intuitive technique for remotely reprogramming sensors deployed in the field. However, implementing agent programs directly on a CPS is complicated by the many limitations of sensor nodes, including limited memory, slow processors, low bandwidth and finite energy. ActorNet eases development by providing an abstract environment for lightweight concurrent object-oriented mobile code on WSNs. As such, it enables a wide range of dynamic applications, including fully customizable queries and aggregation functions, in-network interactive debugging and high-level concurrent programming on the inherently parallel sensor network platform. Moreover, ActorNet cleanly integrates all of these features into a fine-tuned, multi-threaded embedded Scheme interpreter that supports compact, maintainable programs—a significant advantage over primitive stack-based virtual machines.

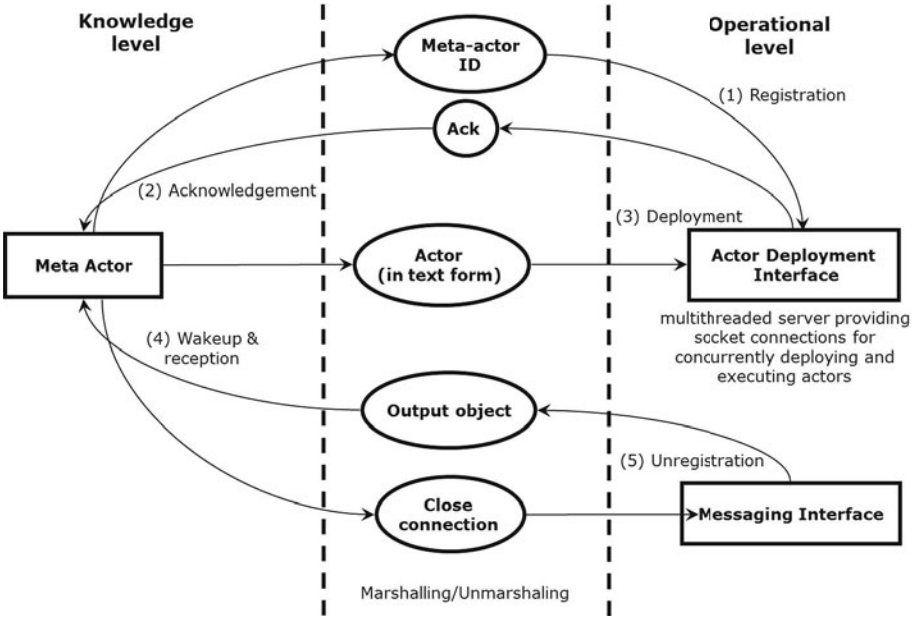

Fig. 4. Communication protocol between actors and meta-actors

6.1 ActorNet Platform

ActorNet is a distributed actor platform targeted primarily at resource-constrained wireless sensor networks. Each ActorNet node is a multi-threaded interpreter for a high-level actor language (Figure 5).

An ActorNet network can span several Internet-connected sensor networks, as well as PCs. It consists of three types of nodes: sensor nodes, which directly execute actor code; repeaters, which distribute actor messages in local sensor networks; and forwarders, which bridge ActorNet systems across the Internet and provide an external access point to the actor system. It currently executes on TinyOS, a popular operating system for embedded sensors, as well as Windows- and Linux-based PCs. Like other virtual machines, the ActorNet platform provides a uniform computing environment for all actors, regardless of hardware or operating system differences; actors can seamlessly migrate between all these hardware platforms.

Figure 5 depicts the layered architecture of an ambient node ActorNet platform. Actors only use the interpreter module directly; thus implementation details are hidden from actor programs. Lower-level services are necessary to reconcile the desired properties of simplicity and platform-independence in the high-level language with the specifics of the wireless sensor network environment. The layered architecture alleviates some of the complexity of application development for sensor networks, which currently involves a significant amount of low-level programming due to the tight coupling between the application and

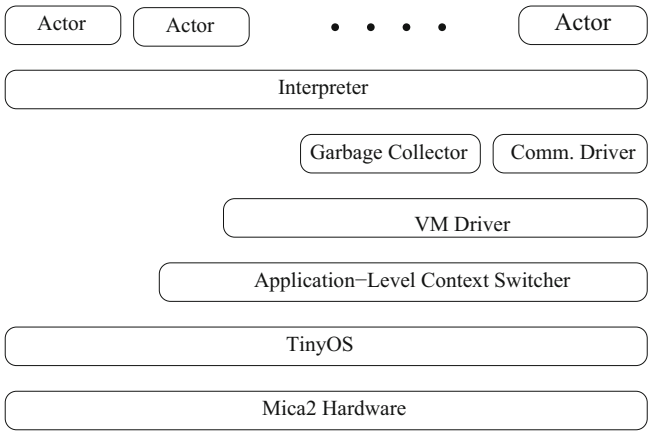

Fig. 5. ActorNet mobile agent platform for networked embedded systems

the operating system. ActorNet aims to provide a stricter decoupling of applications from the operating system, enabling a sensor node to safely load and execute multiple agents, while at the same time simplifying application development.

The ActorNet virtual machine features several services necessary to meet the challenges presented by the CPS environment. These platform services allow efficient memory management and blocking I/O operations for the actor language.

1. *Virtual Memory.* Since the small amounts of RAM available on most embedded devices is insufficient for many applications, ActorNet provides a virtual memory subsystem. It builds a page structure on the permanent storage (e.g., serial data flash) and uses an inverted page table to access pages stored in an LRU cache in RAM.
2. *Application-Level Context Switching.* ActorNet provides an application-level context switching mechanism that allows efficient blocking I/O on top of the strict non-blocking model of TinyOS. This mechanism eases development of maintainable applications. To preserve portability and modularity, the context switching mechanism is implemented purely as an application-level service; it does not modify the underlying OS scheduler.
3. *Multi-phase Garbage Collector.* The ActorNet platform provides a mark-and-sweep garbage collection mechanism. System-level support for garbage collection has many benefits: it eases application development, eliminates the chance of memory leaks, protects other applications from misbehaving actors, and reduces the actor code size. In order to prevent long-running garbage collection tasks from blocking other applications, we divide the sweep step into many short phases. Combined with the context switching functionality, this greatly reduces the impact of garbage collection on application performance.

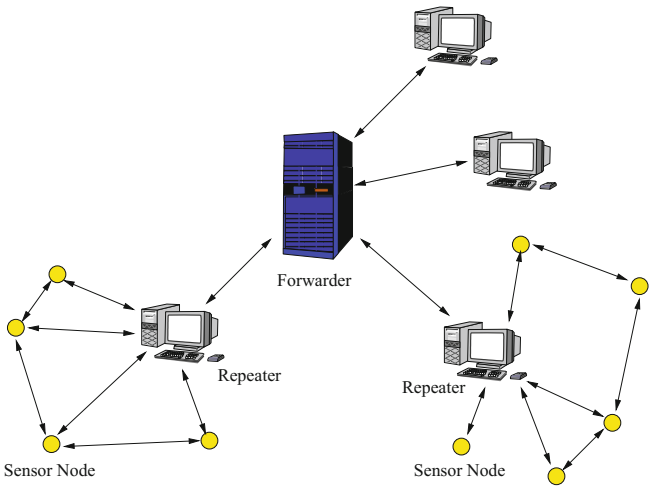

Fig. 6. The network structure of ActorNet

7 Illustration

We now demonstrate how a sensor network application can be executed by our self-mediated execution architecture. As an example, we consider a distributed target tracking service similar to one proposed by Liu *et al.* [10].

Distributed target tracking is one of the canonical problems in sensor networks. Target tracking algorithms typically consist of detecting a signal emitted by the target, identifying or classifying the target by its type or signature, and once detected and classified, keeping track of its position as it changes over time. We assume that the tracking application is provided to us as a composition of Signal Detection, Target Classification and Track Maintenance application-level services, along with Localization, Time Synchronization, Routing and Group Formation middleware services, whose dependency graph is shown in Fig. 7. In this figure, Forever Do and For All Nodes Do are special control constructs, which are executed entirely by meta-actors.

Let us look at how this composite service is deployed and executed. In response to a request, the self-mediated execution architecture creates a meta-actor for the composite service, and recursively for its individual subcomponents, made possible due to the functional nature of the service composition (see Section 4).

Consider a request to the Signal Detection service, which is the first service instance ready to execute, due to having no dependencies. The target tracking service meta-actor requests to deploy a Signal Detection service on all nodes in the network. The Signal Detection service contract specifies that it needs a certain type of sensor, say a magnetometer, to detect the target. The resource and implementation matching components will locate a suitable implementation by pattern matching the request with service and resource descriptions.

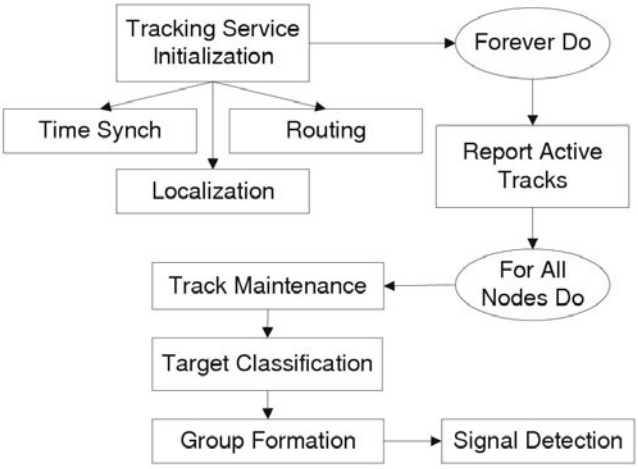

Fig. 7. Composite target tracking application represented as a service dependency graph

At this point we have an executable code segment that is ready to be transported to the destination node. After the scheduling process is completed, the service request is also marshaled and transported. This in effect creates a platform-specific relocatable executable.

The only resource used by the Signal Detector service is the magnetometer; however, since multiple uncoordinated applications may be concurrently executing on the WSN, the magnetometer at the target node may currently be in use by another service. It is the responsibility of the scheduling component of the architecture to control its invocation time, such that the required resource is available prior to request deployment. This means that the Signal Detector service request may be blocked from deployment until the magnetometer at its destination node becomes available.

Now consider a scenario where after the target tracking service starts executing, an intrusion detection application enters the system, ready to be executed. It is also represented as a composition of services, and happens to rely on the same target tracking service in its computation. However, its specification contains additional constraints on the Target Classification service, e.g., requiring a higher confidence threshold before a target is positively identified.

Due to our design choices (dynamicity and late binding), we have an opportunity for run-time optimization. When this new application starts the self-mediated execution process, the implementation matching service lists the instances of the already-deployed services as matching the requested service contracts. This is suitable for Signal Detection and Track Management services, but the Target Classification service will fail a constraint check. With negligible incremental deployment cost, the former two service instances will be reused by the system and linked to a newly instantiated Target Classification service instance meeting the more stringent requirements of the new application.

8 Discussion

To summarize our approach, applications represented as a functional composition of services with well-defined interfaces are executed in a concurrent and distributed manner by the self-mediated execution architecture. Service implementations fitting application requirements are found and deployed on demand, sharing or reusing already-deployed implementations whenever possible. Invocation requests to these services are also generated on demand. Let us first address the benefits of taking this approach to building WSN applications.

8.1 Benefits

Late binding of service implementations and network resources is a key distinguishing feature of our architecture. By postponing the explicit identification of methods and resources until the point when they are actually used, we avoid the problem of *overspecification*. Overspecification occurs when the programmer implicitly or explicitly supplies constraints on execution beyond what is strictly necessary to specify the desired behavior. Sampling a sensor at a *given* node within a region of interest, where sampling a sensor at *any* node within that region would have been sufficient is an example of overspecification. This leads to inefficient use of limited shared resources within the networked embedded system, since the scheduler or optimizer is subjected to unnecessary constraints imposed by the programmer. With late binding, we postpone the decision-making process as to which method or resource to employ from design-time to run-time, thus allowing the scheduler or optimizer components more freedom.

We also argue that service abstraction, a reusable service composition machinery, and fine-grained code deployment and execution allow creating more dynamic, maintainable and customizable applications for WSNs. Code mobility also enables predictive behavior or system-directed load balancing: a service may decide to move from one node to another to better achieve its goal, or to do so more efficiently.

8.2 Requirements

Our self-mediated execution architecture requires the application specification to be provided in the form of a composition of service descriptions. This specification may or may not be immediately executable, as not all elements are fully specified. For example, the composition may not contain a reference to a specific Target Classification service implementation, but rather to a Target Classification service contract. It is up to the mediated execution architecture to identify an appropriate implementation or resources matching the contract.

We require all composable services to conform to such a contract specification. This translates to a substantial amount of work on the service designer's part to supply a sufficiently rich service contract to turn an existing middleware service into a composable service usable by our architecture. Fortunately, the transition process can be facilitated by starting with a very rigid constraint on the interface

(e.g., it is only usable by the service it was originally designed for) and gradually relaxing it as a more comprehensive service contract is constructed.

The dynamic service deployment and execution process relies on the availability of a fine-grained code deployment method for the WSN, meaning that it should be possible to deploy a service to a single node or to a subset of nodes in the network at runtime.

8.3 Implementation Issues

We have a prototype implementation of an architecture supporting a subset of the described functionality in the context of dynamic application deployment for Ambient Intelligence applications, called *Ambiance* [14]. This system reuses Dart [15] at its knowledge level, which is an example of a service composition framework in alignment with our design principles, for both representing applications and supporting the self-mediated execution process. Additionally, Dart supports creating intuitive Web interfaces for interactive specification of applications by multiple uncoordinated end-users at run-time. At the operational level, Ambiance deploys the ActorNet mobile agent platform. The interactions between these two levels conform to the logical architecture described in prior sections.

Our architecture makes use of a service composition framework, online resource scheduling and task allocation algorithms, fine-grained runtime code deployment, and implementation- and resource-matching methods. Several approaches to these tasks have been proposed:

An extensive body of distributed resource scheduling and task allocation research is available from the real-time and parallel processing communities, and may be applied to the WSN domain given allowances for limited bandwidth, memory and processing capabilities and high likelihood of failures of typical sensor nodes.

Mobile agent platforms such as ActorNet and Agilla [4, 9] or virtual machine-based code migration systems such as Melete [19] satisfy our requirement for a fine-grained run-time code deployment method.

We consider the Decision Making and Optimization component to be one of the most challenging aspects in the implementation of our architecture. While a simple heuristic-based approach is sufficient for a prototype implementation, achieving efficient resource utilization is vital to making WSNs a suitable platform for deploying large, concurrent applications. Developing novel algorithms for this task is an important direction for future research. We believe that the clean separation of request processing and execution aspects in our architecture facilitates the integration of these components.

8.4 Potential Applications

We see a number of application opportunities for this architecture. In [14], we have described a possible application to a query processing engine for end-user defined concurrent queries integrating with sensor networks.

Another promising possibility is sensor-rich business processes, where sensors are attached to "smart items," and the interactions between these items is modeled within the business process. The goal then consists of enabling the execution on the sensor nodes of that part of the business logic. For example, in a safety process, smart chemical containers collaboratively ensure continuous compliance with certain storage regulations. Any violation of these rules results in local alerts, as well as reporting to the back-end systems [3].

Such processes are considered to increase visibility, enable real-time decision making and business process adjustment, and thus allow responding to situations more efficiently, with a higher degree of quality and end-user satisfaction. They also allow for management by exception, where the relocated processes only notify the back-end system of extraordinary situations, increasing scalability and speed of detecting situations that require action (avoiding latency of control loop), and does not require a constant connection to the back-end [17].

9 Conclusion

Our research aims to improve programmability of complex cyber-physical systems by separating context-independent application logic, known at design-time, from the low-level execution context considerations that are often unavailable until run-time, by means of a dynamic service-oriented architecture. An expanded version of the current architecture prototype, taking advantage of real-time operating system features for scheduling, control and synchronization, is under development. The primary focus is on incorporating the high-level decision making and optimization components into the existing scheduling model of an embedded RTOS. This includes a study of which aspects of low-level service optimization and control decision can be externalized.

We believe that the design principles and architecture defined in this paper have wider implications beyond the adaptive execution of composite middleware services in cyber-physical systems. We are specifically interested in coordination behaviors within the CPS as well as their relation to outside platforms and applications. We are thus investigating the scalability of our architecture in the context of complex hierarchical processes running in a pervasive computing environment, which also includes cyber-physical components. We consider system-wide optimization of independent concurrent applications in a shared CPS environment to be a major open research topic.

Acknowledgments. The authors gratefully acknowledge the support of this research by the National Science Foundation, under grants CMS 06-00433 and CNS 10-35773.

References

1. Agha, G.: Actors: A Model of Concurrent Computation in Distributed Systems. MIT Press (1986)
2. Carriero, N., Gelernter, D.: Linda in context. Communications of the ACM 32(4), 444–458 (1989)

3. Decker, C., Spiess, P., sa de Souza, L.M., Beigl, M., Nochta., Z.: Coupling enterprise systems with wireless sensor nodes: Analysis, implementation, experiences and guidelines. In: Pervasive Technology Applied @ PERVASIVE (May 2006)
4. Fok, C.L., Roman, G.-C., Lu, C.: Mobile agent middleware for sensor networks: An application case study. In: 4th International Conference on Information Processing in Sensor Networks, pp. 382–387 (April 2005)
5. Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., Culler, D.: The nesC language: A holistic approach to networked embedded systems. ACM SIGPLAN Notices 38(5), 1–11 (2003)
6. Gnawali, O., Greenstein, B., Jang, K.Y., Joki, A., Paek, J., Vieira, M., Estrin, D., Govindan, R., Kohler, E.: The TENET architecture for tiered sensor networks. In: ACM Conference on Embedded Networked Sensor Systems (November 2006)
7. Gu, T., Pung, H., Zhang, D.: A service-oriented middleware for building context-aware services. J. Network and Computer Applications 28(1), 1–18 (2005)
8. Krämer, B.J.: Component meets service: what does the mongrel look like? ISSE 4(4), 385–394 (2008)
9. Kwon, Y., Sundresh, S., Mechitov, K., Agha, G.: ActorNet: An actor platform for wireless sensor networks. In: International Conference on Agents and Multiagent Systems (2006)
10. Liu, J., Liu, J., Reich, J., Cheung, P., Zhao, F.: Distributed Group Management for Track Initiation and Maintenance in Target Localization Applications. In: Zhao, F., Guibas, L.J. (eds.) IPSN 2003. LNCS, vol. 2634, pp. 113–128. Springer, Heidelberg (2003)
11. Liu, J., Zhao, F.: Towards semantic services for sensor-rich information systems. In: International Workshop on Broadband Advanced Sensor Networks (October 2005)
12. Mechitov, K., Razavi, R., Agha, G.: Architecture design principles to support adaptive service orchestration in WSN applications. ACM SIGBED Review 4(3) (2007)
13. Papazoglou, M.P., Traverso, P., Dustdar, S., Leymann, F., Krämer, B.J.: Service-oriented computing: A research roadmap. In: Cubera, F., Krämer, B.J., Papazoglou, M.P. (eds.) Service Oriented Computing (2006)
14. Razavi, R., Mechitov, K., Agha, G., Perrot, J.-F.: Dynamic macroprogramming of wireless sensor networks with mobile agents. In: 2nd Workshop on Artificial Intelligence Techniques for Ambient Intelligence (January 2007)
15. Razavi, R., Perrot, J.F., Johnson, R.: Dart: A meta-level object-oriented framework for task-specific, artifact-driven behavior modeling. In: Proceedings of DSM 2006, pp. 43–55 (2006)
16. Singh, M.P., Huhns, M.N.: Service-Oriented Computing: Semantics, Processes, Agents. John Wiley and Sons (2005)
17. Spiess, P., Vogt, H., Jutting, H.: Integrating sensor networks with business processes. In: Real-World Sensor Networks Workshop at ACM MobiSys (June 2006)
18. Tsai, W.T.: Service-oriented system engineering: A new paradigm. In: Proc. IEEE International Workshop on Service-Oriented Systems Engineering, pp. 3–8 (2005)
19. Yu, Y., Rittle, L.J., Bhandari, V., LeBrun, J.B.: Supporting concurrent applications in wireless sensor networks. In: 4th International Conference on Embedded Networked Sensor Systems, pp. 139–152 (2006)