Report Number: WUCSE-2008-24

2008-01-01

Flexible Service Provisioning for Heterogeneous Sensor Networks

Chien-Liang Fok, Gruia-Catalin Roman, and Chenyang Lu

This paper presents Servilla, a highly flexible service provisioning framework for heterogeneous wireless sensor networks. Its service-oriented programming model and middleware enable developers to construct platform-independent applications over a dynamic set of devices with diverse computational resources and sensors. A salient feature of Servilla is its support for dynamic discovery and binding to local and remote services, which enables flexible and energy-efficient in-network collaboration among heterogeneous devices. Furthermore, Servilla provides a modular middleware architecture that can be easily tailored for devices with a wide range of resources, allowing even resource-limited devices to provide services and leverage resource-rich devices for... Read complete abstract on page 2.

Flexible Service Provisioning for Heterogeneous Sensor Networks

Chien-Liang Fok, Gruia-Catalin Roman, and Chenyang Lu

Complete Abstract:

This paper presents Servilla, a highly flexible service provisioning framework for heterogeneous wireless sensor networks. Its service-oriented programming model and middleware enable developers to construct platform-independent applications over a dynamic set of devices with diverse computational resources and sensors. A salient feature of Servilla is its support for dynamic discovery and binding to local and remote services, which enables flexible and energy-efficient in-network collaboration among heterogeneous devices. Furthermore, Servilla provides a modular middleware architecture that can be easily tailored for devices with a wide range of resources, allowing even resource-limited devices to provide services and leverage resource-rich devices for in-network processing. Microbenchmarks demonstrate the efficiency of Servilla's middleware, and an application case study for structural health monitoring on a heterogeneous testbed consisting of TelosB and Imote2 nodes demonstrates the efficacy of its programming model.This paper is replaced by tech report WUCSE-2009-2.

2008-24

Flexible Service Provisioning for Heterogeneous Sensor Networks

Authors: Chien-Liang Fok, Gruia-Catalin Roman, and Chenyang Lu

Corresponding Author: liangfok@wustl.edu

Web Page: http://www.cse.wustl.edu/wsn/index.php?title=Servilla

Abstract: This paper presents Servilla, a highly flexible service provisioning framework for heterogeneous wireless sensor networks. Its service-oriented programming model and middleware enable developers to construct platform-independent applications over a dynamic set of devices with diverse computational resources and sensors. A salient feature of Servilla is its support for dynamic discovery and binding to local and remote services, which enables flexible and energy-efficient in-network collaboration among heterogeneous devices. Furthermore, Servilla provides a modular middleware architecture that can be easily tailored for devices with a wide range of resources, allowing even resource-limited devices to provide services and leverage resource-rich devices for in-network processing. Microbenchmarks demonstrate the efficiency of Servilla's middleware, and an application case study for structural health monitoring on a heterogeneous testbed consisting of TelosB and Imote2 nodes demonstrates the efficacy of its programming model.

Notes:
This paper is replaced by tech report WUCSE-2009-2.

Type of Report: Other

Flexible Service Provisioning for Heterogeneous Sensor Networks

Chien-Liang Fok, Gruia-Catalin Roman, and Chenyang Lu
Dept. of Computer Science and Engineering
Washington University in St. Louis
Saint Louis, MO, 63105, USA
[liang, roman, lu]@cse.wustl.edu

ABSTRACT

This paper presents Servilla, a highly flexible service provisioning framework for heterogeneous wireless sensor networks. Its service-oriented programming model and middleware enable developers to construct platform-independent applications over a dynamic set of devices with diverse computational resources and sensors. A salient feature of Servilla is its support for dynamic discovery and binding to local and remote services, which enables flexible and energy-efficient in-network collaboration among heterogeneous devices. Furthermore, Servilla provides a modular middleware architecture that can be easily tailored for devices with a wide range of resources, allowing even resource-limited devices to provide services and leverage resource-rich devices for in-network processing. Microbenchmarks demonstrate the efficiency of Servilla's middleware, and an application case study for structural health monitoring on a heterogeneous testbed consisting of TelosB and Imote2 nodes demonstrates the efficacy of its programming model.

1. INTRODUCTION

Wireless sensor networks (WSNs) [7] are becoming increasingly heterogeneous with nodes that span a wide range of memory, processing, and sensing capabilities [15]. Furthermore, WSNs tend to be highly dynamic, existing nodes fail and new nodes are continuously developed and deployed. Network heterogeneity and dynamics give rise to many challenges like having to create multiple versions of an application tailored to each hardware platform, and re-designing the implementation each time a new platform is introduced into the network. Solving these challenges is critical to enable WSNs to evolve into a ubiquitous, permanent, and continuously evolving sensing infrastructure [13]. The creation of such a WSN is important because it makes new types of integrated sensing systems such as urban sensing [30, 37] and building automation. New programming models and frameworks are needed to allow applications to make use of whatever resources are available at a certain point in time, and adapt to changing hardware resources due to the evolving WSN infrastructure.

Service-oriented computing (SOC) [33] is a promising programming model for handling node heterogeneity in the Internet. Recently SOC has been explored in the context of WSNs. For example, two systems that use SOC with WSNs are Tiny Web Services (TWS) [35] and PhyNet^{TM} [2]. TWS provides an HTTP server on each WSN node and enables applications to access services using HTTP requests. PhyNet^{TM}provides a gateway that exposes WSN capabilities as web services. Both systems treat WSNs mainly as data sources for applications rather than as a computing infrastructure for executing the application. While these systems effectively enhance the interoperability among heterogeneous nodes, in both systems applications execute outside of the WSN and are bound to services at a central base station, limiting their ability to support in-network collaboration among heterogeneous nodes.

This paper presents *Servilla*, a highly flexible service provisioning framework for heterogeneous wireless sensor networks. Using Servilla, an application can dynamically discover and bind to local and remote services, facilitating in-network collaboration between heterogeneous WSN nodes and achieving higher levels of efficiency and flexibility. Servilla structures applications in terms of platform-independent tasks and expose platform-specific capabilities as services. Tasks search for services that match their requirements, and dynamically bind to them when they are available. A specialized service description language is introduced that enables tasks to selectively bind to services that exploit the capabilities of whatever hardware is available at a particular time and place. Criteria such as distance and energy efficiency level can be used in making service binding

decisions.

Unlike prior work on SOC for sensor networks [35, 2], Servilla takes the SOC programming model *inside* a WSN to address challenges that stem from network heterogeneity and dynamics. The loose coupling between service consumers and providers can be used to separate application-level platform-independent logic from the low-level software components that perform platform-specific capabilities. Furthermore, by allowing application logic to execute inside a WSN, higher levels of efficiency are obtainable via in-network collaboration. For example, in a structural health monitoring application, a low-power node may detect potentially damage-inducing shocks and activate more powerful nodes that collaborate to localize the damage. Or, in a surveillance application, low-power nodes may sense vibrations from an intruder and activate more powerful nodes with cameras to take pictures of the intruder [19]. The ability to support collaboration among heterogeneous devices is a key distinguishing feature that separates this work from existing SOC frameworks for WSN.

This paper makes the following primary contributions.

- We propose a specialized *service-oriented* programming model that enables flexible and energy-efficient collaboration among heterogeneous nodes via dynamic binding to local or remote services;
- We design and implement a *modular middleware architecture* that can be easily tailored for devices with a wide range of resources, which allows extremely resource-constrained devices to provide services and leverage resource-rich devices for in-network processing;
- We present microbenchmarks that demonstrate the feasibility and efficiency of Servilla on two representative hardware platforms (TelosB and Imote2) with significantly different resources;
- We provide an application case study on structural health monitoring that demonstrates the efficacy of the Servilla programming model.

The remainder of the paper is organized as follows. Section 2 presents Servilla's programming model. Section 3 presents Servilla's programming languages. Section 4 presents Servilla's middleware architecture. Section 5 presents a prototype implementation. Section 6 presents an evaluation on a heterogeneous WSN. Section 7 presents a structural health monitoring application that is implemented using Servilla. Section 8 presents related work. The paper ends with conclusions in section 9.

2. PROGRAMMING MODEL

An overview of a heterogeneous sensor network running the Servilla middleware is shown in Figure 1. It consists

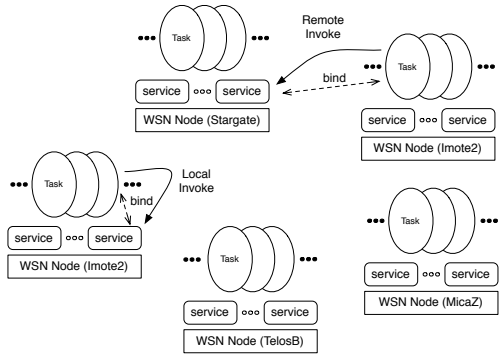


Figure 1: Servilla targets heterogeneous and dynamic WSNs in which nodes provide services that expose platform-specific capabilities, while platform-independent application tasks use these services either locally or remotely.

of a heterogeneous mix of WSN nodes, services that expose platform-specific functionality, and tasks that perform application-specific operations in a platform-independent fashion. Note that Servilla is meant for heterogeneous networks that integrate fine-grained and low-power sensing of resource-constrained nodes (like the TelosB [34]) and the computational resources of resource-rich nodes (like the Imote2 [6]). It is not meant for flat WSNs composed entirely of resource-poor nodes.

When a task needs to perform a platform-specific operation, it searches for a service that is able to carry out the operation. The service may be provided by the node on which the task resides or on a remote node. A key feature and novelty of Servilla lies in its support for service provisioning inside the network, including local and neighborhood service discovery, matching, and dynamic binding. This service provisioning model enhances the system flexibility and enables applications to effectively exploit services inside the network.

2.1 Tasks

A task contains an application's code, execution state, and list of *service specifications*, as shown in Figure 2. Note that tasks are platform-independent while the service specifications allow the SOC framework to handle platform heterogeneity and changes *transparently* from applications. The service specifications describe the services that are needed by the task. In addition to describing the service's interface, the specification also includes desired properties of the service, like the maximum amount of power that the service may consume. This enables tasks to better distinguish between similar services by, for example, binding to the ones that are the most energy efficient.

2.2 Services

Services expose platform-specific capabilities and are dy-

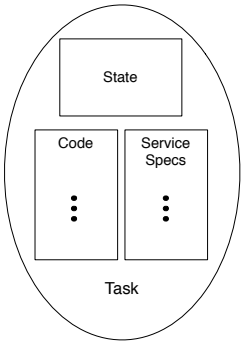


Figure 2: A task consists of code, service specifications, and execution state.

namically discovered and bound to tasks. Since they are platform-specific, they can be fine-tuned for maximum efficiency. Servilla services are able to maintain state, provide multiple methods, and have their own thread of control. This enables more powerful services that can, for example, provide access to large memory spaces that are only available on certain hardware platforms. It also enables services to run concurrently with tasks, like continuously monitoring a sensor in the background.

A task can discover and bind to both local services and remote services in its neighborhood that match its service specifications. Remote services are accessed wirelessly from another node. They increase flexibility by allowing a task to run on nodes that do not have all of the services it requires, by exploiting services available on neighboring nodes. This has benefits beyond increasing the number of nodes that a task can execute on. For example, remote binding may increase energy efficiency by exploiting low-power nodes in the neighborhood, or enable tasks to perform in-network data fusion of sensor data collected from the neighborhood.

2.3 Service Binding

Service binding consists of a three-step process: service discovery, matching, and informing the task of the chosen service. Service discovery consists of finding services that are available to a given task. In many traditional SOC frameworks, it involves querying a service directory located at the base station. While this is sufficient when applications reside outside of the WSN and must access the WSN via the base station anyway, it is not appropriate for Servilla because (1) any WSN node may initiate the discovery process and forcing nodes to access a potentially distant directory is not efficient, and (2) keeping a centralized directory up to date is difficult in a dynamic environment. Instead, Servilla assumes that tasks prefer a nearby service over one that is further away to save energy. For this reason, Servilla distributes the service directory. Specifically, every node that provides services maintains a service registry that contains the specifications of the services that it provides. This registry serves as a directory that is searchable locally and remotely among neighbors, enabling tasks to dynamically discover services that are available.

Service matching involves finding a service that fulfills a task's requirements. Recall that the service directories are distributed across WSN nodes, making the location of the service explicit. Servilla's service matching process exploits this fact by first querying the local service directory before searching neighboring ones, ensuring that a task is always bound to a local service whenever possible, reducing network overhead.

Once a satisfactory service is found, the binding process is completed by informing the task of the chosen service. This is done by giving the address of the node that provides the service to the task. Using this address, the task is able to invoke the service. Note that this address is hidden from the application developer, who is able to invoke the service based on its name, a process that is described next.

2.4 Service Invocation

Service invocations are analogous to remote procedure calls (RPCs) [8]. The invocation must include the name of the service, the method within the service to execute, and the input parameters needed by the method. A service may provide multiple methods because it can maintain state. For example, a service that provides access to a storage space may provide methods "read" and "write", and a service that continuously monitors ambient vibrations may provide methods "start" and "stop". As with all RPCs, the process is prone to failure. To account for this, Servilla provides an error flag indicating why the invocation failed. This is essential because service invocations may fail in many ways depending on whether the service is local or remote, and tasks may want to handle various error conditions differently. For example, local invocations may fail because the service is busy, in which case the task may wait and try again later, while remote invocations may fail due to disconnection, in which case the task may want to abort.

3. PROGRAMMING LANGUAGE

Servilla provides two specialized programming languages. The first, called *ServillaSpec*, is used to create service specifications that enable flexible matching between applications and services. The second, called *ServillaScript*, is used to create application tasks. Servilla services are implemented in NesC [14] and compiled into efficient TinyOS [20] bytecode. Each of Servilla's specialized languages are now described.

3.1 ServillaSpec

The purpose of ServillaSpec is to enable the specification of a service used to match services needed by tasks with services provided by the nodes. To support resource-constrained nodes, the service specification language must

```
NAME = fft
METHOD = fft-real
INPUT = {int dir, int numSamples, float[] data}
OUTPUT = float[]
ATTRIBUTE Version = 5.0
ATTRIBUTE MaxSamples = 5000
ATTRIBUTE Power = 10
```

Figure 3: A specification describing a FFT service

```
1.  uses Temperature; // declare required service
2.
3.  void main() {
4.      int count = 0; float temp;
5.      bind(Temperature, 2); // bind service within 2 hops
6.      while(count++ < 10) {
7.          temp = invoke(Temperature, "get"); // invoke service
8.          send(temp);
9.      }
10.     unbind(Temperature);
11. }
```

Figure 4: A task that invokes a temperature sensing service 10 times

be compact and should not require an overly complex matching algorithm. As such, we avoid standard specification languages like WSDL [38] and provide ServillaSpec instead. ServillaSpec avoids verbose syntax and limits the types of properties in a service specification. An example is shown in Figure 3. The first line specifies the name of the service. It is followed by three-line segments each specifying the name, input parameters, and output results of a method provided by the service. The remainder of the specification is a list of attributes. These properties enable flexibility in matching by defining a name, relation, and value. Using attributes, an application developer can, for example, select a floating point FFT service that consumes *at most* 50mW, which would match a service whose specification is shown in Figure 3.

By limiting the property types to be the five shown in Figure 3 (i.e., NAME, METHOD, INPUT, OUTPUT, and ATTRIBUTE), and arranging them to always be in the same order, the size of the specification can be greatly compressed. For example, since the service's NAME property always appears first, the property's identifier, NAME, can be ommitted. Thus, the NAME property in the specification shown in Figure 3 can be compressed to just 4 bytes, "fft" followed by a null terminator. This compression saves memory and enables greater matching efficiency.

3.2 ServillaScript

ServillaScript is used to create application tasks. Its syntax is similar to other high level languages like JavaScript [11], but with key extensions for service provisioning. An example, shown Figure 4, implements an application that periodically takes the temperature and sends the reading to the base station. It declares the name of the file that contains the specification of the service that it needs on line 1. In this case, this file contains a specification of a temperature sensing service. The specification is located in a separate file to increase the modularity of the code. The task initiates the process of finding a service within two hops on line 5. The task then loops ten times, each time invoking the service on line 7 and sending the temperature to the base station on line 9. The task ends by disconnecting from the service on line 10.

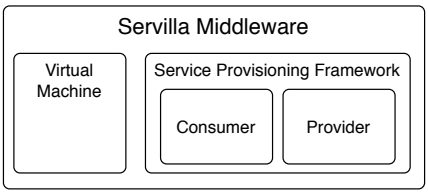


Figure 5: Servilla's middleware consists of a virtual machine and a service provisioning framework (SPF). The SPF consists of a consumer and provider.

The example above illustrates how ServillaScript enables tasks to 1) indicate which services are needed, 2) initiate the service discovery process, 3) invoke services, and 4) disconnect from services. Aspects not shown for brevity include a way to enable a task to determine whether a service is bound, and, if so, how many hops away the service is located. This will allow the task to throttle how often it invokes the service based on its distance. Another aspect not shown is error handling code. If an error occurs due to a service becoming unavailable, the invocation will return an error indicating the cause, as discussed in Section 2.

4. MIDDLEWARE

An overview of Servilla's middleware architecture is shown in Figure 5. It consists of a virtual machine (VM) and a service provisioning framework (SPF). The VM is responsible for executing application tasks. The SPF consists of a consumer (SPF-Consumer) that discovers and accesses services, and provider (SPF-Provider) that advertises and executes services.

A virtual machine (VM) is used because current WSN nodes span a wide range of processors with different instruction sets, and it is not known whether any one of these will prevail. Application tasks are compiled into a static instruction set provided by the VM, which is uniform across all hardware platforms, enabling tasks to be platform-independent. Furthermore, the VM provides the dynamic deployment of application tasks, justifying the need for dynamic service binding. The VM is based on the one provided by Agilla [12] though with major extensions to support service specifications and an interface with the SPF. Specifically, whenever a task performs an operation involving a service, the VM passes the task to the SPF-Consumer, which is described next.

4.1 SPF-Consumer

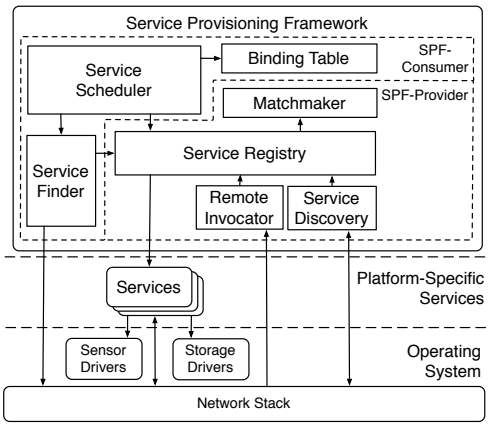


Figure 6: The detailed architecture of the Service Provisioning Framework.

The SPF-Consumer is responsible for discovering, matching, and invoking services. Its architecture, shown in Figure 6, consists of a Service Scheduler, Binding Table, and Service Finder. When a task executes a bind operation, the SPF-Consumer saves the specification and range for service discovery (number of hops) in the Binding Table and notifies the Service Finder to start searching for a match. To save energy, local services are preferred over remote ones, and closer ones are preferred over those farther away — all things being equal. More often the application can specify explicitly the energy usage it expects from a service with the service meeting the criteria being selected. Once a match is found, the address of the node providing the service is recorded in the Binding Table, enabling the task to access the service.

To maximize concurrency, the Service Finder runs in the background allowing tasks to continue executing while a service is being discovered. It starts by searching the local node followed by each neighbor with increasing distance until either a match is found or all candidate neighbors are checked without finding a match. Although the Service Finder runs asynchronously with the task, invocations are performed synchronously. That is, if a task attempts to invoke a service before the Service Finder finishes, the task is blocked until either a match is found, or the Service Finder fails to find a match. Synchronous invocations simplify application code by avoiding call-back functions and possible race conditions.

The Service Scheduler carries out the actual invocation. It takes the input parameters provided by the task, sends them to the node hosting the service, and waits for the results to arrive. Once the results arrive, it passes them to the task which can then resume executing. If the results do not arrive within a certain timeframe, the Service Scheduler aborts the operation and notifies the task of the error.

4.2 SPF-Provider

The SPF-Provider advertises and executes services. Its architecture, shown in Figure 6, contains a main component called the Service Registry which records the specifications of local services. When the SPF-Consumer tries to find a service, the SPF-Provider consults its Service Registry, which uses the Matchmaker to determine whether a match exists. The results are sent back to the SPF-Consumer, which saves the information in the Binding Table.

In the current Servilla Middleware, the SPF-Provider is responsible for determining whether a match exists, meaning the task's specification must be sent to the SPF-Provider. Alternatively, the SPF-Consumer could perform the match. This would reduce the footprint of the SPF-Provider, thereby decreasing the minimum system requirements, but requires that the SPF-Provider send the SPF-Consumer all of its specifications, which may incur higher communication cost.

4.3 Middleware Modularity

WSNs are becoming extremely diverse with resources that differ by several orders of magnitude [34, 6]. Even as hardware improves, cost considerations ensure that there will always be nodes that do not have enough resources to implement the entire Servilla middleware. The modularity of Servilla's middleware enables these nodes to participate. Recall that it is modularized into three basic components: a VM, SPF-Consumer, and SPF-Provider. By exploiting the decoupled nature of SOC, the Servilla middleware may be configured to include a subset of components, while allowing nodes with different configurations to collaborate through service provisioning. Specifically, the Servilla middleware supports the following configurations:

- **VM + SPF:** The full Servilla framework.
- **VM + SPF-Consumer:** Executes tasks and provides access to remote services only.
- **SPF-Provider:** Provides services for neighboring tasks to use.
- **VM:** Can serve as a forwarder of application tasks.

A detailed analysis of the memory consumed by each of these configurations is given in Section 6.1. The configuration containing only the SPF-Provider is particularly interesting because it allows resource-weak but energy efficient nodes to provide services to more powerful nodes. This can result in greater overall energy efficiency and, assuming the weak nodes are less costly and more prevalent, decrease the cost of achieving dense sensing and sensing coverage among the more powerful nodes by allowing them to exploit the sensing capabilities of the weak nodes surrounding them.

5. IMPLEMENTATION

	MicaZ	TelosB	Imote2
Release Date	2004	2005	2007
Processor	7.4MHz 8-bit Atmel ATmega128L	8MHz 16-bit TI MSP430	13-416MHz 32-bit Intel PXA271 XScale
Radio	IEEE 802.15.4	IEEE 802.15.4	IEEE 802.15.4
Memory	128KB Code, 4KB Data	48KB Code, 10KB Data	32MB Shared
Flash	512KB	1MB	32MB
Price	$99	$69	$299

Table 1: WSN nodes vary widely in computational resources.

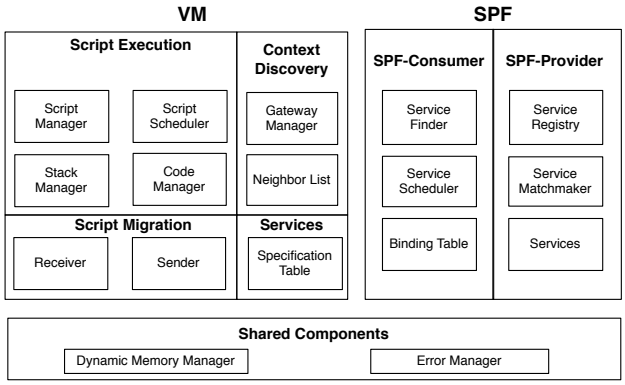


Figure 7: Servilla's middleware components.

An implementation of Servilla is available on Servilla's website [36]. It is built on TinyOS 1.x [1] which supports numerous WSN platforms including those shown in Table 1. While these platforms are different, they all communicate over IEEE 802.15.4 [21], an industry-standard wireless interface.

Servilla's implementation is divided into two levels as shown in Figure 7: a lower level that provides essential shared services, and a higher level consisting of Servilla's VM and SPF. This section first discusses the lower level followed by the SPF in the upper level (the VM's implementation was already discussed in Section 4). It ends with a discussion of the implementation of Servilla's programming languages.

5.1 Shared Components

The shared components reflect the particular properties of the platform on which Servilla is implemented, in this case TinyOS. They include a dynamic memory manager, error manager, and reliable network interface.

Servilla provides a dynamic memory manager to make efficient use of memory on a WSN node. This is important because Servilla has many components that require varying amounts of memory over time. The dynamic memory is divided into 10-byte blocks that are arranged as a linked list within the dynamic memory manager. The dynamic memory manager is shared by most components in Servilla's middleware, maximizing the flexibility of memory allocation.

To aid in debugging, Servilla provides an error manager that detects errors and sends a summary of the problem to the base station. The error manager is shared by all other components in Servilla's middleware.

5.2 SPF Implementation

The SPF is implemented from scratch in NesC. It is divided into two modules, the SPF-Consumer and SPF-Provider, as shown in Figure 7. The Service Scheduler within the SPF-Provider serializes service invocations performed by different tasks to simplify the implementation and prevent saturating the wireless channel. The Service Registry in the SPF-Provider implements an 8-bit parameterized interface for attaching services, meaning it can support up to 256 local services.

5.3 Servilla Compiler

We developed the Servilla compiler that can compile application tasks (written in ServillaScript) and service specifications (written in ServillaSpec) into a compact binary format. For example, the task shown in Figure 4 is compiled into 181 bytes of code and 30 bytes of specifications, and the specification shown in Figure 3 is compiled into just 64 bytes.

6. EVALUATION

Evaluating Servilla requires a heterogeneous WSN consisting of nodes with a wide range of resources. To achieve this, a WSN consisting of TelosB [34] and Imote2 [6] nodes is used. The computational resources vary widely between these two platforms, as shown in Table 1. The evaluation consists of two parts. First, the memory footprint of different middleware configurations is measured. This determines how flexible the framework is in terms of accommodating nodes with different amounts of memory. Second, the runtime efficiency of service discovery and invocation across heterogeneous WSNs is evaluated.

6.1 Memory Footprint

Each Imote2 node has 32MB of code and data memory, which is sufficient to hold the entire Servilla middleware plus many services. The combined size of the entire Servilla middleware without services is a mere 318KB on the Imote2, or only about 1% of the total, leaving plenty for services.

[1]TinyOS has a two-tiered concurrancy model one of which consists of tasks. TinyOS tasks, which are low-priority background processes, should not be confused with Servilla tasks, which are platform-independent application processes.

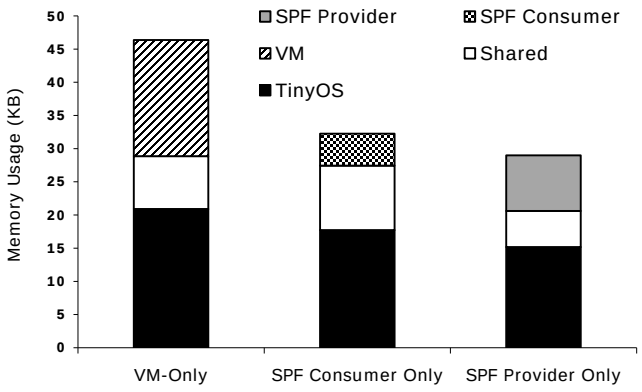


Figure 8: The code memory footprint of various Servilla configurations on the TelosB platform.

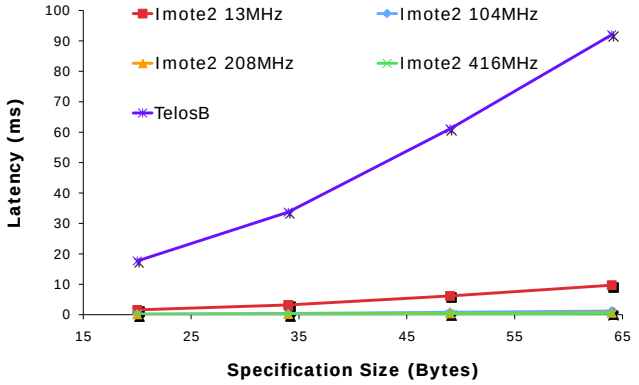


Figure 9: The latency of comparing a specification vs. its size.

In contrast, TelosB nodes have only 48KB of code memory and 10KB of data memory. This is not even enough to fit the VM and SPF-User at once, as shown in Figure 8. The figure shows the amount of code memory consumed by three different configurations of Servilla. For each configuration, the memory footprint of each individual component is indicated. Of the three configurations shown, only two are valid. The valid configurations are the ones labels "VM Only" and "SPF Provider only." The third configuration, labeled "SPF Consumer Only," is included to illustrate why TelosB nodes cannot include both the VM and the SPF-Consumer. The figure shows that the VM consumes about 46KB of code memory, and the SPF-Consumer consumes 32KB, of which at least 5KB is unique to the SPF-Consumer. This additional 5KB on top of the 46KB consumed by the VM exceeds the 48KB available on the TelosB node.

The results indicate that resource-poor nodes like the TelosB may serve one of two functions in a Servilla network. The first function is to implement just the SPF-Provider and provide services to other nodes that are able to host a SPF-Consumer. This may prove highly beneficial if the network integrates low-power resource-constrained nodes like the TelosB and the computational resources of resource-rich nodes like the Imote2. In this case, the energy efficiency of the TelosB can be exploited by the more power-hungry Imote2 to increase overall energy efficiency. Furthermore, the lower cost of the TelosB may enable more of them to be deployed. Allowing the Imote2 nodes to access the sensors on the TelosB nodes surrounding it may help it achieve dense sensing or sensing coverage at lower cost. The second function is to implement just the VM and serve as a forwarder of application tasks. Either option enables TelosB nodes to contribute to a WSN application.

Our results clearly show how Servilla's modular architecture increases the scope of hardware devices that can participate in a Servilla network. It reduces the minimum system requirements by allowing nodes to be integrated into an application through the service provisioning framework. This is made possible by exploiting the decoupled nature of SOC, and enables Servilla to function across a wider spectrum of WSN platforms.

6.2 Efficiency of Service Binding

Service binding consists of three parts: 1) service discovery, 2) service matching, and 3) updating the Binding Table. This study focuses on local service binding because remote binding is dominated by wireless communication, which is not the focus of this study. As such, the latency of service discovery is negligible since it involves accessing the local Service Registry. Updating the Binding Table is also negligible since it involves writing only five bytes into memory. Thus, this section focuses on service discovery, specifically the latency of determining a match.

To evaluate the efficiency of service binding, the Matchmaker is used to compare two copies of specification `FFT`, shown in Figure 3. This incurs the worst-case latency since every property within the specification must be considered. Each experiment is repeated twenty times on both TelosB and Imote2 platforms running at all possible CPU speeds and the average latency is calculated. The latency is measured by toggling a general I/O pin before and after the comparison operation, and capturing the time between toggles using an oscilloscope. The results are shown in Table 2. Since the experiment runs locally, the measurements exhibit very low variance and the confidence intervals are omitted.

The results indicate that service binding is highly efficient. The latencies are small compared to the execution times of certain VM instructions. The total latency consists of the latencies of comparing the signature and each attribute within the specification. The column labeled "other" is the overhead incurred by the Matchmaker between comparing properties. As expected, the latency of comparing two specifications depends on the speed and architecture of the processor, and is mostly inversely proportional to the CPU speed reflecting the CPU-bound nature of the comparison.

To determine how the latency is affected by the specifica-

Node	CPU Speed	Bus Speed	Sig.	Attr. 1	Attr. 2	Attr. 3	Other	Total	Units
TelosB	8MHz	8MHz	18	14	24	29	8	92	ms
Imote2	13MHz	13MHz	1569	1421	2642	3272	784	9688	μs
Imote2	104MHz	104MHz	198	180	330	408	94	1209	μs
Imote2	208MHz	208MHz	99	89	165	204	47	604	μs
Imote2	416MHz	208MHz	71	62	113	136	31	413	μs

Table 2: The latency of service matching when comparing two FFT-real service specifications

Spec.	Name	# of Properties	Size (Bytes)
1	`FFT-real`	3	64
2	`light-tsr`	2	46
3	`accel-3d`	5	85
4	`flash_mem`	1	34

Table 3: The sizes of the specifications used to evaluate service invocation

Spec.	1	2	3	4	Units
TelosB	2	13	25	45	ms
13MHz Imote2	206	1571	3251	6785	μs
104MHz Imote2	26	196	406	849	μs
208MHz Imote2	13	98	203	424	μs
416MHz Imote2	9	67	133	265	μs

Table 4: The latency of obtaining the a service's binding state

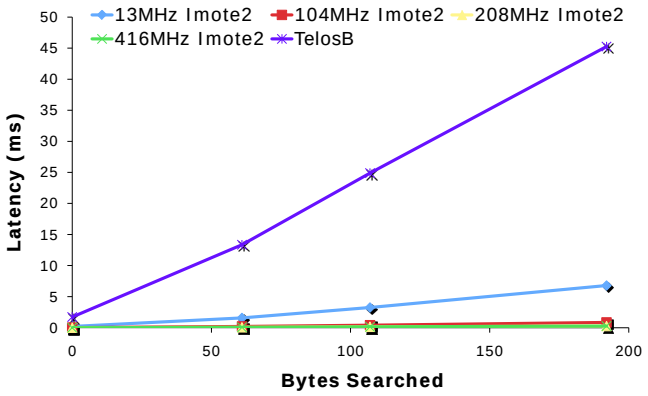


Figure 10: The latency of obtaining a service's binding state vs. number of bytes searched

tion's size, `FFT` is compared to versions of itself with one, two, and all three of its attributes removed. The latencies of comparing these specifications is plotted against their sizes and the results are shown in Figure 9. The results indicate that the latency is roughly proportional to its size. It is not exactly proportional because of the additional overhead incurred with the addition of each attribute, as indicated by the "other" column in Table 2.

6.3 Efficiency of Service Invocation

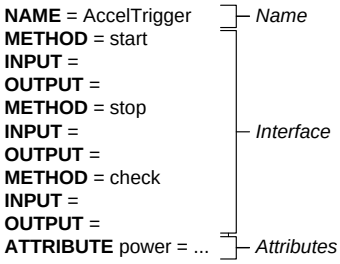
The efficiency of service invocation depends on the latency of obtaining the service specification's binding state. To determine this, the Specification and Binding Tables are loaded with `FFT` followed by three specifications whose properties are summarized in Table 3. The latency of obtaining the binding information of each specification is measured using the same technique described in Section 6.2. The results, shown in Table 4, indicate that the latency depends on the sizes of the specifications that occur before it in the table. This makes sense since all these specifications must also be analyzed in the current implementation. Figure 10 shows the linear relationship between latency versus the total size of the specifications that are located before it. In the future, the implementation can be improved by using a hash function to achieve constant-time access to the binding state.

7. APPLICATION CASE STUDY

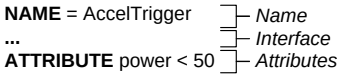
This section evaluates the efficacy of the Servilla programming model through a case study on a structural health monitoring application designed to localize damage in structures (e.g., a bridge). This application enables real-time evaluation of a structure's integrity, thereby reducing manual inspection costs while increasing safety. WSNs have recently been used to successfully localize damage to experimental structures using a homogeneous network of Imote2 nodes [17]. In this case, the algorithm, called Damage Localization Assurance Criterion (DLAC), was written using NesC specifically for the Imote2. The Servilla implementation generalizes and improves upon the original implementation by making it platform-independent and increasing its overall energy efficiency by exploiting network heterogeneity.

The heterogeneous WSN used in this study consists of TelosB and Imote2 nodes. DLAC can only run on the Imote2 due to insufficient memory on the TelosB. However, Imote2 nodes consume more energy than TelosB nodes. Our new structural health monitoring application combines the advantages of both platforms by keeping the Imote2 nodes idle as much as possible, and using the TelosB nodes to monitor the ambient vibration levels. The Imote2 nodes are only activated when the TelosB nodes detect that the ambient vibration levels exceed a certain threshold, at which time they perform the DLAC algorithm. The dual-level nature of this configuration is common to other applications that run over heterogeneous WSNs like vehicle tracking [18], and is essential for conserving energy and ensuring network longevity.

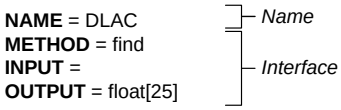
The Servilla implementation relies on two services: `AccelTrigger` and `DLAC`. Ambient vibrations are monitored by `AccelTrigger`, which sets a flag when they exceed a threshold. Its specification is shown in Figure 11(a). The service has three methods: `start`, `stop`, and `check`. Methods `start` and `stop` control when the service mon-



(a) The specification of service `AccelTrigger` provided by Imote2 and TelosB nodes. The power attribute specifies the amount of power the service consumes. It is 145mW on the Imote2, and 9mW on the TelosB.



(b) The specification of a low-power version of service `AccelTrigger`, which is provided by the application task. Its interface is omitted since it is the same as the one in Figure 11(a). A high-power version has attribute power $\geq$ 50 mW.



(c) The specification of service `DLAC` provided by Imote2 nodes.

Figure 11: The services used by the damage localization application

itors the local accelerometer. The status of the flag is obtained by invoking `check`. Both the Imote2 and TelosB nodes provide `AccelTrigger`. They differ in their power attribute, since the Imote2 consumes more power than the TelosB (145mW vs. 9mW).

The specification of service `DLAC` is shown in Figure 11(c). It contains a single method, `find`, that takes no parameters and returns an array of floating-point numbers that are used to localize damage to the bridge [17].

The application's task is shown in Figure 12. The first three lines specify the names of the files containing the required service specifications. The content of `AccelTriggerLP` is shown in Figure 11(b), and the content of `DLAC` is shown in Figure 11(c). Notice that `AccelTriggerLP` matches the TelosB version of the `AccelTrigger` service shown in Figure 11(a) because its power attribute is less than 50mW. `AccelTriggerHP` contains the same specification as `AccelTriggerLP` except its power attribute is $\geq$ 50 mW, which matches the service provided by the Imote2.

The application attempts to reduce energy consumption by preferentially binding to an `Acceltrigger` service that consumes less power. It does this by first attempting to bind using the specification within `AccelTriggerLP` on line 6, before using the specification within `AccelTriggerHP` on line 13. Once an `AccelTrigger` service is bound, the

```
1.  uses AccelTiggerHP;
2.  uses AccelTiggerLP;
3.  uses DLAC;
4.
5.  void main() {
6.      bind(DLAC, 0); // bind DLAC service
7.      if(!isBound(DLAC)) exit(); // failed to bind DLAC
8.      bind(AccelTriggerLP, 1); // bind low-power AccelTrigger service
9.      if(isBound(AccelTriggerLP)) {
10.         invoke(AccelTriggerLP, "start");
11.         waitForTrigger(1);
12.     } else {
13.         bind(AccelTriggerHP);
14.         if(isBound(AccelTriggerHP)) {
15.             invoke(AccelTriggerHP, "start");
16.             waitForTrigger(0);
17.         }
18.     }
19. }
20.
21. void waitForTrigger(int useLowPower) {
22.     while(true) {
23.         int vibration;
24.         if (useLowPower)
25.             vibration = invoke(AccelTriggerLP, "check");
26.         else
27.             vibration = invoke(AccelTriggerHP, "check");
28.         if (vibration == 1) {
29.             if (useLowPower)
30.                 invoke(AccelTriggerLP, "stop");
31.             else
32.                 invoke(AccelTriggerHP, "stop");
33.             doDLAC();
34.         }
35.         sleep(1024*60*5); // sleep for 5 minutes
36.     }
37. }
38.
39. void doDLAC() {
40.     float[25] dlac_data;
41.     dlac_data = invoke(DLAC, "find");
42.     send(dlac_data); // send DLAC data to base station
43. }
```

Figure 12: The damage localization application task

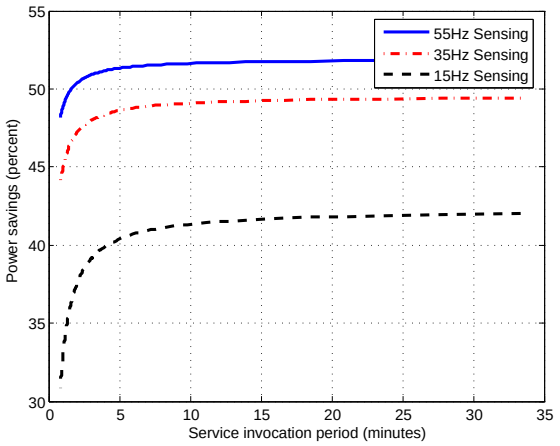


Figure 13: Percent power savings of heterogeneous vs. homogeneous WSN.

task periodically queries it to determine if the acceleration readings are above a certain threshold (lines 21-37). If it is, `DLAC` is invoked and the results are sent to the base station (lines 41-42).

To evaluate the benefit of exploiting network heterogeneity on Servilla, the task shown in Figure 12 is injected into two WSNs: a homogeneous network consisting of only Imote2 devices, and a heterogeneous network consisting of both Imote2 and TelosB devices. Since the application is written using Servilla, it is able to run on both types of networks without modification. In both cases, `DLAC` is executed by the Imote2, meaning the power consumption of performing damage localization is constant. However, the power consumption of `AccelTrigger` does vary. This is because Servilla's service provisioning framework enables an application to exploit more energy-efficient services when possible in a platform-independent and declarative fashion. Specifically, if TelosB nodes are present, the service will be executed on a TelosB node since its `AccelTrigger` service consumes lower power and hence matches the first service specified by the application task. Otherwise it will be executed on the Imote2. We compare the power consumption of invoking `AccelTrigger` in different network configurations.

Since invoking `AccelTrigger` on the TelosB requires a remote invocation, the amount of energy saved by the heterogeneous implementation depends on the invocation and sensing frequencies. If the service is invoked too often, more energy will be spent on wireless communication than is saved. Likewise, if the sensor is accessed very infrequently, the benefits of using the TelosB is diminished since the nodes will remain asleep a larger percentage of the time. To determine how much energy savings is possible, an oscilloscope is used to measure the time each platform spends computing, communicating wirelessly, and sensing, in both a homogeneous and heterogeneous network. The sensing frequency is varied between 15Hz and 55Hz (the maximum sampling frequency of the TelosB), and the service invocation frequency is varied between 50 seconds to 35 minutes. The percent savings of using a heterogeneous network relative to a homogeneous network is then calculated and the results are shown in Figure 13.

The results show that invoking the service too frequently will reduce the amount of power saved since doing so incurs more network overhead, while increasing the sensing frequency results in more power savings since the TelosB is able to sense while consuming less power. The results also show that there is a limit to the amount of energy that can be saved as the service invocation period increases. This is because as the invocation period increases, the energy savings becomes the difference between the sensing energy consumed by the Imote2 versus the TelosB.

This case study demonstrates how Servilla enables the development of platform-independent applications that operate over a heterogeneous WSN, and how Servilla facilitates in-network collaboration between different types of nodes that leads to higher energy efficiency. Moreover, it demonstrates that Servilla enables an application to bind to a more energy-efficient service through service specification.

8. RELATED WORK

Servilla is related to various WSN scripting systems in its use of a VM for interpreting application tasks. Scripting has been used in WSNs, though for different reasons. Some scripting systems, including Maté [25], ASVM [26], SwissQM [31], and Agilla [12], enable reprogramming. Other systems, including Melete [40] and SensorWare [4], enable multiple applications to share the same WSN. All of these systems come with different scripting languages including TinyScript [24], SCript [10], Mottle [26], SNACK [16], and variants of SQL [28, 39]. Servilla differs from these systems in that it focuses on how to address challenges that arise due to network heterogeneity and dynamics, by allowing scripts to dynamically find and access platform-specific services. Unlike applications written in other scripting systems, Servilla applications are able to perform platform-specific operations and adapt to changing hardware capabilities, while remaining platform-independent.

One scripting system, DVM [3], explores the idea of integrating platform-independent scripts with native services. This system features a dynamically extensible virtual machine in which native services can register extensions. While this enables fine-tuning the boundary between interpreted code and native code, DVM does not support the SOC model that enables the flexible integration of heterogeneous nodes.

SOC has long been used in traditional networks to enable applications written by different organizations to interoperate. There are many SOC systems including SLP [22], Jini [23], CORBA [32], Salutation [5], and Web Services [1]. They are made possible by numerous technologies that make

language-independent communication possible, which is essential for interoperability. Some of these technologies include SOAP [9], RPC [8], DCOM [9], and WCF [29]. Servilla differs from these systems by focusing on how service-provisioning can be made lightweight and yet remain flexible to allow slightly different specifications to match. This is necessary due to the limited resources available on some WSN nodes. Moreover, its modular middleware can be tailored to nodes with a wide range of computation resources.

There are some efforts to port traditional SOC technologies into the WSN domain. They include Tiny Web Services [35] and Arch Rock's PhyNet™ [2]. Both systems heavily optimize traditional Internet protocols to enable them to function under the severe resource constraints of WSNs. Unlike Servilla, they do not provide a mechanism for service discovery or the flexible matching between service users and consumers *within* the WSN. Instead, they focus on how to enable language-independent communication between services inside the WSN and applications residing outside of the WSN. Servilla is complementary to these efforts; Servilla may be extended to leverage off their communication services to expose WSN services to applications external to the WSN, while these systems may rely on Servilla to bring the full capabilities of SOC inside the WSN itself.

In addition to scripting and SOC, Servilla also introduces the idea of a modular and configurable platform in which extremely resource-poor nodes only implement a fraction of the entire framework. This enables a hierarchy in which weak nodes serve more powerful nodes. The idea of having a hierarchy within a WSN is not new. Tenet [15] promotes this idea by creating a two-tired WSN in which the lower tier consists of resource-poor nodes that can accept tasks from higher-tier nodes. It differs from Servilla in that it does not support SOC which enables flexible discovery and binding between different nodes.

SONGS [27] is an architecture for WSNs that allows users to issue queries that are automatically decomposed into graphs of services which are mapped onto actual nodes. SONG does not provide flexible service binding among heterogeneous nodes.

9. CONCLUSIONS

The complexity of developing applications for increasingly heterogeneous and dynamic WSNs demands a new programming and middleware framework. Servilla meets this demand by enabling platform-independent applications to be created that can adapt to a dynamic and heterogeneous hardware platform. This is achieved by adopting a flexible service-oriented programming model in which applications access platform-specific services. A specialized service description language is introduced that enables flexible matching between applications and services, which may reside on different nodes. Servilla provides a modular middleware architecture to enable resource-poor nodes to contribute services, facilitating in-network collaboration among a wide range of devices. The efficiency of Servilla's implementation is established via microbenchmarks on two representative classes of hardware platforms (the TelosB and Imote2). The benefits and effectiveness of Servilla's programming model is demonstrated by a structural health monitoring application case study.

10. REFERENCES

[1] ALONSO, G., CASATI, F., KUNO, H., AND MACHIRAJU, V. *Web Services*. Springer, 2003.

[2] ARCH ROCK. Arch Rock PhyNet™. `http://www.archrock.com/product/`.

[3] BALANI, R., HAN, C.-C., RENGASWAMY, R. K., TSIGKOGIANNIS, I., AND SRIVASTAVA, M. Multi-level software reconfiguration for sensor networks. In *EMSOFT'06* (New York, NY, USA, 2006), ACM Press, pp. 112–121.

[4] BOULIS, A., HAN, C.-C., AND SRIVASTAVA, M. Design and implementation of a framework for efficient and programmable sensor networks. In *MobiSys'03* (May 2003), USENIX, pp. 187–200.

[5] CHAKRABORTY, D., AND CHEN, H. Service discovery in the future for mobile commerce. *Crossroads 7*, 2 (2000), 18–24.

[6] CROSSBOW TECHNOLOGIES. Imote2 datasheet. `http://tinyurl.com/5jrw85`.

[7] CULLER, D., ESTRIN, D., AND SRIVASTAVA, M. Overview of sensor networks. *IEEE Computer 37*, 8 (2004), 41–49.

[8] DAVE MARSHALL. Remote procedure calls (rpc). `http://www.cs.cf.ac.uk/Dave/C/node33.html`.

[9] DAVIS, A., AND ZHANG, D. A comparative study of soap and dcom. *J. Syst. Softw. 76*, 2 (2005), 157–169.

[10] DUNKELS, A. A low-overhead script language for tiny networked embedded systems. Tech. Rep. T2006:15, Swedish Institute of Computer Science, Sept. 2006.

[11] FLANAGAN, D. *JavaScript: The Definitive Guide, 4th Ed.* O'REILLY, Inc., 2001.

[12] FOK, C.-L., ROMAN, G.-C., AND LU, C. Rapid development and flexible deployment of adaptive wireless sensor network applications. In *ICDCS'05* (June 2005), IEEE, pp. 653–662.

[13] FOK, C.-L., ROMAN, G.-C., AND LU, C. Towards a flexible global sensing infrastructure. *SIGBED Rev. 4*, 3 (2007), 1–6.

[14] GAY, D., LEVIS, P., VON BEHREN, R., WELSH, M., BREWER, E., AND CULLER, D. The nesc language: A holistic approach to networked embedded systems. In *PLDI'03* (New York, NY, USA, 2003), ACM, pp. 1–11.

[15] GNAWALI, O., JANG, K.-Y., PAEK, J., VIEIRA, M., GOVINDAN, R., GREENSTEIN, B., JOKI, A., ESTRIN, D., AND KOHLER, E. The tenet architecture for tiered sensor networks. In *SenSys'06* (New York, NY, USA, 2006), ACM Press, pp. 153–166.

[16] GREENSTEIN, B., KOHLER, E., AND ESTRIN, D. A sensor network application construction kit (snack). In *SenSys'04* (New York, NY, USA, 2004), ACM, pp. 69–80.

[17] HACKMANN, G., SUN, F., CASTANEDA, N., LU, C., AND DYKE, S. A holistic approach to decentralized structural damage localization using wireless sensor networks. In *RTSS'08* (11 2008), IEEE.

[18] HE, T., KRISHNAMURTHY, S., LUO, L., YAN, T., GU, L., STOLERU, R., ZHOU, G., CAO, Q., VICAIRE, P., STANKOVIC, J. A., ABDELZAHER, T. F., HUI, J., AND KROGH, B. Vigilnet: An integrated sensor network system for energy-efficient surveillance. *ACM Trans. Sen. Netw. 2*, 1 (2006), 1–38.

[19] HE, T., KRISHNAMURTHY, S., STANKOVIC, J. A., ABDELZAHER, T., LUO, L., STOLERU, R., YAN, T., GU, L., ZHOU, G., HUI, J., AND KROGH, B. Vigilnet:an integrated sensor network system for energy-efficient surveillance. *ACM Transactions on Sensor Networks (under submission)* (2004).

[20] HILL, J., SZEWCZYK, R., WOO, A., HOLLAR, S., CULLER, D., AND PISTER, K. System architecture directions for networked sensors. In *Architectural Support for Programming Languages and Operating Systems* (2000), pp. 93–104.

[21] IEEE 802.15 WORKING GROUP FOR WPAN. IEEE 802.15.4 website. `http://www.ieee802.org/15/`.

[22] KEMPF, J., AND PIERRE, P. S. *Service location protocol for enterprise networks: implementing and deploying a dynamic service finder*. John Wiley & Sons, Inc., New York, NY, USA, 1999.

[23] KUMARAN, I., AND KUMARAN, S. I. *Jini Technology: An Overview*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[24] LEVIS, P. The TinyScript Manual. http://tinyurl.com/57kycj, July 2004.

[25] LEVIS, P., AND CULLER, D. Maté: a tiny virtual machine for sensor networks. In *ASPLOS'02* (New York, NY, USA, 2002), ACM Press, pp. 85–95.

[26] LEVIS, P., GAY, D., AND CULLER, D. Active sensor networks. In *NSDI'05* (May 2005).

[27] LIU, J., AND ZHAO, F. Towards semantic services for sensor-rich information systems. In *2nd Int. Conf. on Broadband Networks* (2005), pp. 44–51.

[28] MADDEN, S., FRANKLIN, M. J., HELLERSTEIN, J. M., AND HONG, W. Tag: a tiny aggregation service for ad-hoc sensor networks. *SIGOPS Oper. Syst. Rev. 36*, SI (2002), 131–146.

[29] MICROSOFT. Windows communication foundation. `http://msdn2.microsoft.com/en-us/library/ms735119.aspx`.

[30] MURTY, R., GOSAIN, A., TIERNEY, M., BRODY, A., FAHAD, A., BERS, J., AND WELSH, M. Citysense: A vision for an urban-scale wireless networking testbed. Tech. Rep. 13-07, Harvard University, 2007.

[31] MÃIJLLER, R., ALONSO, G., AND KOSSMANN, D. A virtual machine for sensor networks. In *EuroSys 2007* (March 2007).

[32] OBJECT MANAGEMENT GROUP. Corba basics. `http://www.omg.org/gettingstarted/corbafaq.htm`.

[33] PAPAZOGLOU, M. P., TRAVERSO, P., DUSTDAR, S., AND LEYMANN, F. Service-oriented computing: State of the art and research challenges. *Computer 40*, 11 (2007), 38–45.

[34] POLASTRE, J., SZEWCZYK, R., AND CULLER, D. Telos: enabling ultra-low power wireless research. In *IPSN'05* (Piscataway, NJ, USA, 2005), IEEE Press, p. 48.

[35] PRIYANTHA, N., KANSAL, A., GORACZKO, M., AND ZHAO, F. Design and implementation of an evolutionary sensor network. In *SenSys'08* (New York, NY, USA, 2008), ACM.

[36] SERVILLA. Website. `http://www.cse.wustl.edu/wsn/index.php?title=Servilla`.

[37] STREELINE. Parking management. `http://www.streetlinenetworks.com`.

[38] W3C. Web services description language (wsdl). `http://www.w3.org/TR/wsdl`.

[39] YAO, Y., AND GEHRKE, J. The cougar approach to in-network query processing in sensor networks. *SIGMOD Rec. 31*, 3 (2002), 9–18.

[40] YU, Y., RITTLE, L. J., BHANDARI, V., AND LEBRUN, J. B. Supporting concurrent applications in wireless sensor networks. In *SenSys'06* (New York, NY, USA, 2006), ACM Press, pp. 139–152.