Sensor Network Middleware

DAVIM: Adaptable Middleware for Sensor Networks

Wouter Horré, Sam Michiels, Wouter Joosen, and Pierre Verbaeten • Katholieke Universiteit Leuven

Middleware services facilitate sensor-network application development. DAVIM is adaptable middleware that enables dynamic management of services and isolation between simultaneously running applications.

Developing and managing end-to-end sensor network applications (those that span both sensor networks and traditional business infrastructures) can be complex and costly. Middleware that allows easy composition of reusable functionality (sensor networks as lightweight service platforms) can reduce this complexity. Businesses typically deploy sensor networks for substantial periods of time. A sensor network acts as a reusable asset that generates value for several business processes. So, several users will likely use the network simultaneously. In addition, network applications can change during the network’s potentially long lifetime. Middleware can dynamically add services to such a network and isolate applications from one another.

A middleware solution for such applications must meet three key requirements:

- Services should be separate from the applications using them.
- Managing available services should be dynamic and easy.
- Multiple applications running on the same sensor network should be isolated.

DAVIM (the DistriNet Adaptable Virtual Machine) meets these requirements and is the first step toward a lightweight service middleware solution. The DAVIM architecture provides in-sensor network support for the dynamic integration of middleware services. We have implemented a prototype to evaluate this architecture. This evaluation provides promising indications that implementing a lightweight service platform on state-of-the-art sensor network hardware is feasible.

Motivating example

Consider a chemical storage facility (see figure 1). The sensor network’s nodes are permanently deployed in the building (gateways), yet attached to containers stored inside the warehouse. Various applications use this dynamic, heterogeneous sensor network to gather information about the warehouse and the products stored inside.
A facility management application uses the sensor network to control the temperature in rooms where inflammable substances are stored. At the same time, the network monitors incoming and outgoing products for a stock-management application. In addition, a safety application simultaneously uses the network to monitor the presence of people near toxic products or to detect when incompatible chemicals are too close together.

These three applications rely on various functional blocks that are implemented on the sensor network, such as localization, detection of people and products, and temperature monitoring. Yet, these applications also require components that execute in a traditional infrastructure. For example, the business logic for stock management probably runs on an application server. A safety-monitoring console might run on the safety officer’s portable device or smart phone. A technician might use a PDA to diagnose the air conditioning. In addition, access control on the gateways might be in place to ensure that only the safety officer can access the sensor network’s safety component.

Owing to sensor networks’ specific characteristics, developing such end-to-end applications can be difficult. However, implementing recurring functionality (localization, people detection, and so on) as reusable middleware services can reduce this complexity. Ideally, building an application should be as simple as choosing the right services and composing them into an application. We call a sensor network that allows this easy composition of middleware services a lightweight service platform. Such a platform is useful only if its services are integrated in an overall middleware architecture that exposes service APIs to the applications consistently.

Moreover, development of the services and the applications probably occurs independently. For instance, a third party that specializes in localization technology probably developed the localization service in the example shown in figure 1. The application component that uses the location information to discover which area of the warehouse is exceeding the temperature threshold is part of the facility-management application. Presumably, the facility management’s infrastructure supplier developed this application. Thus, the middleware platform should actively support this separation of concerns by allowing separate development of services and facilitating exposure of their interfaces to application developers.
The middleware must also let the sensor network’s owner dynamically manage the available services. Given the uncertainty about which services will be needed during the network’s lifetime, restricting service deployment to static deployment is infeasible. Suppose, for instance, that the stock-management application in our example is installed a year after the deployment of the sensor network. The application requires a product-detection service to operate properly. If this service is not yet present on the sensor network, the middleware platform must enable dynamic deployment of this service. In addition, it should be possible to remove services that are no longer needed (for example, because the facility-management application is replaced with a new system). It should also be possible to update existing services (for example, to fix bugs or replace an algorithm with a more suitable variant).

A final concern that the middleware platform must tackle is how to isolate applications. In our example, an air-conditioning technician obviously shouldn’t know which products enter or leave the warehouse. This information should be accessible only to the stock-management application. Therefore, the applications running on the same sensor network must be isolated from one another. Ideally, each application should have the illusion of exclusive access to the sensor network. On the other hand, sharing services should be possible when appropriate. The facility-management and stock-management applications in our example both need a localization service. Owing to the scarce resources available on the sensor network, duplicating this service isn’t desirable.

**Main features of DAVIM**

Existing virtual-machine technology provides a promising way to realize the separation of services and applications but lacks support for the dynamic management of services and isolation of multiple applications. The DAVIM architecture (shown in figure 2) leverages state-of-the-art sensor network virtual-machine technology to meet these requirements.¹ (The “State-of-the-Art Virtual-Machine Technology” sidebar discusses this recent virtual-machine technology and some of the operating systems that DAVIM builds on.)

![Figure 2. High-level overview of the DAVIM architecture. DAVIM requires an operating system that supports dynamic memory and dynamic loading of code.](image-url)
State-of-the-Art Virtual-Machine Technology

**DAVIM** provides basic virtual-machine support, dynamic deployment of operational libraries, support for multiple virtual machines, and flexible composition of virtual-machine instruction sets.

**DAVIM** leverages state-of-the-art sensor-network virtual-machine technology. An example of a state-of-the-art virtual machine is Maté, which was designed for the TinyOS operating system. Maté was later extended to a framework for building application-specific virtual machines. This framework lets a user build a virtual machine with a customized instruction set. However, this customization is possible only at the time the virtual machine is built. When an update is necessary after deployment, Maté disseminates a complete TinyOS image with a modified virtual machine in the network. Maté accomplishes this using Deluge, an over-the-air reprogramming mechanism for TinyOS. Melete enhances Maté’s support of concurrent applications. Melete introduces the concept of multiple applications and protects the execution space of an application from access by another application. However, Melete doesn’t consider dynamic updates to the virtual machine.

Rahul Balani and his colleagues developed Dynamic Virtual Machine, a virtual machine for the Sensor Operating System. DVM focuses on multilevel reconfiguration support, so it doesn’t limit the virtual machine’s use. The authors have presented various usage scenarios of the virtual machine and the SOS module-loading mechanism. One scenario uses the virtual machine as a scripting and reconfiguration engine for SOS applications implemented as modules. In contrast, **DAVIM** uses the concept of a virtual machine to separate reusable services from the applications using them.

Tenet is an architecture for tiered sensor networks. The sensor nodes make up the lower tier, or mote tier. Atop this tier sits an overlay network of high-end devices, called the master tier. The mote tier consists of a tasklet library and an engine that supports task executions (compositions of tasklets). This is similar to the separation of services and applications that **DAVIM** proposes. However, for the moment, the Tenet tasklet library can’t be updated at runtime.

**DAVIM** uses a combination of native code for services, and virtual-machine bytecode for applications. Adam Dunkels and his colleagues have investigated the energy efficiency of various reprogramming techniques for wireless sensor networks. They conclude that a combination of virtual-machine code and native code is beneficial for energy efficiency. Ingwar Wirjawan and others also argue for hybrid execution of virtual-machine code and native code.

**DAVIM** relies on the underlying operating system to provide dynamic memory and a mechanism to load native code at runtime. SOS and Contiki both provide these two features. SOS consists of a static kernel with minimal functionality and loadable modules that implement most of the functionality (sensor drivers, routing, and so on). Building an application for SOS involves implementing one or more modules that use the kernel API or the API exported by other modules. Contiki lets users load binary application modules in the standard ELF (executable and linkable format) file format, or a variant called CELF (compact ELF).

References

Basic virtual-machine support

DAVIM is based on technology for sensor-network virtual machines that was developed to support low-cost updates of sensor-network applications. This technology exploits the observation that most sensor-network applications in a particular application domain use the same basic functionality but combine it in different ways. The virtual machine’s instruction set consists of high-level operations corresponding to this basic functionality. This lets application developers specify how to combine these building blocks rather than forcing them to reimplement the same functionality for every new application.

Dynamic deployment of operation libraries

DAVIM requires that the high-level operations in a virtual machine’s instruction set be grouped in operation libraries. DAVIM lets users add, update, or remove such libraries at runtime. The operating system provides support for loadable code. Once the code for an operation library is loaded, DAVIM plugs the library into its core so that it’s available for inclusion in the virtual machine’s instruction set.

Support for multiple virtual machines

DAVIM supports multiple concurrently executing virtual machines. Users can dynamically add, update, or remove these virtual machines from the system. This avoids the difficult task of predicting the number of virtual machines the sensor network will need during its lifetime. The DAVIM core completely isolates the virtual machines from one another.

Flexible composition of virtual-machine instruction sets

Because the instruction set of a virtual machine is limited in size, each virtual machine can include only a subset of the available services. The services needed vary across different virtual machines, depending on the application. Therefore, DAVIM lets users define the instruction set for each virtual machine by dynamically mapping blocks of bytecode to operation libraries. As a consequence, application bytecode is meaningful only if this mapping is also available.

Building blocks

Five components in DAVIM provide all the features just discussed: the application store, the VM controller, the operation store, the VM store, and the coordinator (see figure 2).1

Application store

As part of DAVIM’s basic virtual-machine support, the application store manages the application components. It cooperates with the coordinator to ensure that the application components stay up-to-date. When a new or updated application component arrives (through the coordinator), the application
store installs the component in the correct virtual machine. Thus, the application store is also aware of
the multiple-virtual-machines feature.

**VM controller**

This engine controls each virtual machine’s operation. It schedules the execution of the application
components installed in the different virtual machines and routes events to the correct virtual
machine. It thus cooperates in basic and multiple virtual-machine support.

**Operation store**

**DAVIM** supports the dynamic deployment of operation libraries mainly through the operation store. This
component keeps track of which operation libraries are loaded, and it cooperates with the operating
system to install, update, or remove an operation library. When a new operation library becomes
available or an existing one is updated or removed, the operation store notifies the VM store of this
change.

**VM store**

This component contains all information related to the virtual machines. It ensures the safe concurrent
execution of multiple application components in the same virtual machine. It also implements most of
the support for multiple virtual machines. In addition, the VM store dynamically maps the virtual-
machine instruction sets.

**Coordinator**

The coordinator supports the basic virtual-machine and the multiple-virtual-machine features by
providing the functionality to disseminate application components and virtual machines in the network.
In cooperation with the application store, the coordinator ensures that all network nodes have the
latest version of the application components installed. Similarly, it cooperates with the VM store to
ensure that all network nodes run the same versions of the virtual machines.

**Separation of services and applications**

The **DAVIM** architecture requires that services be implemented as operation libraries plugged into the
**DAVIM** operation store (see the right side of figure 2). The applications consist of bytecode scripts that
run in a virtual machine atop the **DAVIM** core (left side of figure 2). This separation is also possible with
state-of-the-art sensor-network virtual machines, but **DAVIM** additionally lowers the dependency
between the services and the applications by eliminating the static link between virtual-machine
instructions and operation libraries.

The operation libraries corresponding to the services that an application wants to use are included in
its virtual machine’s instruction set. The application then accesses a service by executing an
instruction, which the VM store maps to the corresponding operation library. When a new service
implementation becomes available, the VM store can remap the corresponding instructions to this new
instance. Thus, an application isn’t bound to a specific service implementation.

**Management of available services**

**DAVIM** supports easy management of services by letting users add, update, or remove them
dynamically. Whenever a new service, implemented as an operation library, must be deployed, the
network manager can load this operation library into the **DAVIM** operation store. The manager can
update an existing operation library to reflect a change to the service that this library implements.
When a service becomes superfluous, the administrator can request the removal of the corresponding
operation library from the operation store to free the resources held by the service.

**DAVIM** also provides support for selectively exposing a service to applications. The flexible composition
of a virtual machine’s instruction set, combined with the support for multiple virtual machines, lets the
network administrator expose a service to only those applications that need it and are allowed to use it.

**Isolation of multiple applications**

To isolate applications from one another, **DAVIM** runs them all in their own virtual machine. This minimizes interference between applications. To each application, it seems as if it’s the only application running on the sensor network.

The instruction sets’ flexible composition allows a customized set of services for each application. Therefore, updating a service that only one application uses has no impact on the other applications, because they’re unaware of this service. Furthermore, this flexibility broadens the range of possible applications and removes the implicit dependency between applications that would be present if they all used the same set of services.

**Prototype implementation**

We’ve implemented this architecture in a proof-of-concept prototype on micaZ sensor nodes. Because the architecture relies on an underlying operating system to provide dynamic memory and loadable modules, we chose SOS for our implementation. SOS provides these features and has good support for our micaZ hardware. We implemented the core architecture and the basic operation libraries as SOS modules. Users can also implement new operation libraries as SOS modules, so it’s possible to deploy these operation libraries dynamically.

The limited amount of dynamic memory that micaZ hardware can currently handle prevents realistic field tests. MicaZ sensor nodes have 4 Kbytes of RAM, and SOS reserves 2 Kbytes of this memory for dynamic allocation. Experiments on micaZ nodes have shown that more dynamic memory is needed to activate multiple virtual machines. Yet, simulations show that **DAVIM** needs less than twice the amount of dynamic memory available on micaZ hardware. This promising result motivates us to implement the prototype on more recent hardware—for example, TelosB (10 Kbytes of RAM), XYZ (32 Kbytes of RAM), or iMote2 (32 Mbytes of RAM). Our expectation is that more recent hardware will solve current memory limitations.

**Evaluation**

We evaluated the architecture and the prototype implementation to gain insight concerning three questions:

- Does the architecture’s dynamic nature imply an unreasonable overhead compared to existing virtual-machine architectures?
- What is the benefit of dynamic deployment of operation libraries?
- What are the benefits and drawbacks of supporting multiple virtual machines?

**Overhead**

To answer the first question, we measured **DAVIM**’s execution overhead with respect to SOS and compared it to the overhead of Bombilla (the standard virtual machine of Matê) with respect to TinyOS. We simulated a surge application, periodically sending a sensor reading to a base station, on these four platforms using the Avrora AVR simulator [http://compilers.cs.ucla.edu/avrora). To foster comparable results, we modified the surge application included in the TinyOS release to use the MintRoute component for multihop routing that Bombilla also uses.

All the surge implementations in our tests sent one light sensor reading every 10 seconds. The simulations ran for 18,000 seconds on a 4 × 4 grid with a spacing of 15 meters. We performed the simulations with the following software:
• Avrora development version—CVS (code versioning system) checkout of 5 April 2007, 10:25 CEST,
• SOS 2.0.0,
• TinyOS 1.1.15’ and
• Maté 2.2.2.

We measured the percentage of time the CPU was active for the four surge implementations. Table 1 shows that the overhead of DAVIM compared to SOS is in the same range of the overhead of Bombilla compared to TinyOS (10 to 12 percent). This DAVIM implementation was an unoptimized prototype. We also implemented a quick optimization specialized for the surge application, which reduced DAVIM’s overhead to 8.9 percent.

Table 1. Average CPU active time of the surge application on different platforms.

<table>
<thead>
<tr>
<th>System</th>
<th>Percent active</th>
<th>Overhead (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOS</td>
<td>7.89</td>
<td>NA</td>
</tr>
<tr>
<td>DAVIM</td>
<td>8.80</td>
<td>11.5</td>
</tr>
<tr>
<td>TinyOS</td>
<td>5.31</td>
<td>NA</td>
</tr>
<tr>
<td>Bombilla</td>
<td>5.86</td>
<td>10.3</td>
</tr>
<tr>
<td>DAVIM (surge optimization)</td>
<td>8.59</td>
<td>8.9</td>
</tr>
<tr>
<td>DAVIM (periodic updates)</td>
<td>8.81</td>
<td>11.6</td>
</tr>
<tr>
<td>Bombilla (periodic updates)</td>
<td>5.86</td>
<td>10.3</td>
</tr>
<tr>
<td>DAVIM (static mapping)</td>
<td>8.80</td>
<td>11.5</td>
</tr>
</tbody>
</table>

To measure DAVIM’s overhead during dissemination and installation of new bytecode scripts, we repeated the simulation of the surge application, but we injected a bytecode script that restarted the application every 305 seconds. We did an equivalent experiment with Bombilla. Table 1 again shows that the overheads are comparable.

Dynamic deployment of operation libraries

To answer the second question in our evaluation, we added a new instruction to both DAVIM and Bombilla and compared the size of the update that the sensor nodes must disseminate through the network. We implemented an instruction to calculate an exponentially weighted moving average. To deploy this new instruction for Bombilla, we had to replace the entire image using Deluge, the over-the-air reprogramming system for TinyOS. This image's size for micaZ was 50,046 bytes. To deploy this new instruction for DAVIM, we only had to load a new operation library. To do this, the module-loading system of SOS sent only 612 bytes (two orders of magnitude less than for Deluge). Because communication over the radio is by far the largest source of energy consumption for most sensor-node devices, dynamically updating a virtual machine’s instruction set is far more energy efficient with DAVIM than with Bombilla.

Another benefit of dynamically deploying operation libraries is that adding, updating, or removing virtual-machine functionality doesn’t require rebooting the sensor node. The importance of this benefit is twofold. First, it provides a better separation of services and applications. Adding a new service or
removing an unused one doesn’t interfere with the applications. Second, it supports better isolation of
applications. An update to an operation library in use by one application doesn’t interfere with the
other applications on the sensor node.

Multiple virtual machines

The answer to the third question depends largely on the application domain under consideration. For
an application domain in which only one user runs just one application at a time on the sensor
network, there’s obviously no benefit to having multiple virtual machines installed on a sensor node.
However, this capability is a major requirement for realistic business scenarios in which different users
run several applications in parallel.

State-of-the-art virtual machines, such as Maté and DVM (Dynamic Virtual Machine), have
shortcomings in such complex scenarios. Their support for running multiple bytecode scripts
concurrently is suitable for running multiple applications in parallel, but they don’t provide support for
isolation between those applications. Because the applications all run in the same virtual machine,
updating one of them causes the virtual machine to reboot and affects all the applications. These
applications all must use the same set of operations; hence, the range of functionality they can
implement is limited.

DAVIM provides more flexible support for scenarios in which multiple concurrent applications of
different users are needed. DAVIM’s support for multiple virtual machines lets the network
administrator isolate different users’ applications. DAVIM also allows varying the virtual machines’
instruction sets. This makes it possible for the same sensor-network infrastructure to support a
broader range of applications. In addition, it lets the network administrator limit the available
functionality for untrusted users.

To better understand the drawbacks of using multiple virtual machines compared to running multiple
applications on the same virtual machine, we evaluated the overhead of DAVIM’s support for multiple
virtual machines. Table 2 shows the extra size of the virtual machine and the context (execution
environment for an application component) data structures due to the support for multiple virtual
machines.

Table 2. Overhead required to support multiple virtual machines, including the extra size of
data structures, the size of extra network packets, and the memory overhead required to
support multiple virtual machines compared with the total memory for the applications.

<table>
<thead>
<tr>
<th>Data structure</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>State per virtual machine</td>
<td>25 bytes (60%)</td>
</tr>
<tr>
<td>State per context</td>
<td>5 bytes (11%)</td>
</tr>
<tr>
<td>Trickle state</td>
<td>27 bytes (36%)</td>
</tr>
<tr>
<td><strong>Network packet</strong></td>
<td><strong>Size</strong></td>
</tr>
<tr>
<td>Trickle virtual-machine summary</td>
<td>32 bytes/packet</td>
</tr>
<tr>
<td>Trickle virtual-machine description</td>
<td>18 bytes/packet</td>
</tr>
<tr>
<td><strong>Applications</strong></td>
<td><strong>Memory overhead</strong></td>
</tr>
<tr>
<td>Surge application</td>
<td>87 bytes (31%)</td>
</tr>
<tr>
<td>Three complex applications</td>
<td>242 bytes (5%)</td>
</tr>
</tbody>
</table>
DAVIM uses an extra instance of the Trickle algorithm (originally developed for Maté) to maintain version coherence of the virtual machines. Table 2 shows the memory overhead to store the state of this extra Trickle instance. The table also lists the size of the packets sent by the algorithm. However, the network overhead of Trickle in a stable state is very low.

For the surge application, 31 percent of the total memory consumption relates to the support for multiple virtual machines. However, this application’s bytecode scripts are, on average, very small (6.7 bytes). Moreover, these scripts use only one global variable and no local variables. For a more realistic usage scenario in which three applications run in their own virtual machine with approximately eight application components of 128 bytes per application, only 5 percent of the total memory consumption can be attributed to the support for multiple virtual machines.

The dynamic mapping of instructions to operations introduces an extra level of indirection, with an increase in interpretation overhead as a consequence. As table 1 shows, for the surge application, this increase has no significant effect on the CPU active time compared to DAVIM with a static mapping.

The round-robin scheduler in our current implementation doesn’t distinguish between scheduling application components from one virtual machine or from multiple virtual machines. Thus, the presence of multiple virtual machines doesn’t increase the scheduling overhead in our current implementation. Although the scheduler doesn’t use virtual-machine information in its decision process, it could exploit the use of multiple virtual machines to ensure that every application gets a fair amount of the available CPU time. With multiple applications in the same virtual machine, this wouldn’t be possible.

Therefore, we conclude that whether the overhead for supporting multiple virtual machines is reasonable depends on the expected usage scenario. For complex scenarios, the outcome of this trade-off is in favor of supporting multiple virtual machines.

Our future research will gradually extend its focus toward end-to-end support for complex business applications that employ sensor-network data. Our goal is to develop a complete end-to-end middleware platform that supports the development, deployment, and management of end-user business applications. We will validate our work in the context of industrial case studies, such as multimodal container logistics.

Acknowledgments

We thank Christophe Huygens, Nelson Matthys, and Ann Heylighen for their valuable comments on this article and for proofreading the text.

References

**Wouter Horré** is a PhD research student in the Department of Computer Science at Katholieke Universiteit Leuven. His research interests include software architecture and middleware for sensor networks, and management of large-scale distributed systems. He received his MSc in computer science from Katholieke Universiteit Leuven. He is a PhD fellow of the Research Foundation - Flanders (FWO). Contact him at IBBT-DistriNet Research Group, Dept. of Computer Science, Katholieke Universiteit Leuven, Celestijnenlaan 200A – bus 2402, B-3001 Leuven, Belgium; wouter.horre@cs.kuleuven.be.

**Sam Michiels** is a postdoctoral researcher in the Department of Computer Science at Katholieke Universiteit Leuven. His research interests include reconfigurable middleware for managing large-scale distributed systems. He received his PhD in computer science from Katholieke Universiteit Leuven. He is a postdoctoral fellow of the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT-Flanders). Contact him at IBBT-DistriNet Research Group, Dept. of Computer Science, Katholieke Universiteit Leuven, Celestijnenlaan 200A – bus 2402, B-3001 Leuven, Belgium; sam.michiels@cs.kuleuven.be.

**Wouter Joosen** is a professor in the Department of Computer Science at Katholieke Universiteit Leuven. His research interests include software architecture for distributed systems, aspect-oriented and component-based software development, adaptive middleware, security solutions, and secure software. He received his PhD in computer science from Katholieke Universiteit Leuven. Contact him at IBBT-DistriNet Research Group, Dept. of Computer Science, Katholieke Universiteit Leuven, Celestijnenlaan 200A – bus 2402, B-3001 Leuven, Belgium; wouter.joosen@cs.kuleuven.be.

**Pierre Verbaeten** is a full professor in the Department of Computer Science at Katholieke Universiteit Leuven. His research interests include open system software, dynamic configuration and integration, and ad hoc networks. He received his PhD in computer science from Katholieke Universiteit Leuven. He's a member of the IEEE and ACM. Contact him at IBBT-DistriNet Research Group, Dept. of Computer Science, Katholieke Universiteit Leuven, Celestijnenlaan 200A – bus 2402, B-3001 Leuven, Belgium; pierre.verbaeten@cs.kuleuven.be.

---

Cite this article: