A Middleware-centric Design Flow for Networked Embedded Systems *

F. Fummi, G. Perbellini, R. Pietrangeli, D. Quaglia

University of Verona - Department of Computer Science - Strada le Grazie, 37134, Verona, Italy
{fummi,perbellini,pietrangeli,quaglia}@sci.univr.it

Abstract

The paper focuses on the design of networked embedded systems which cooperate to provide complex distributed applications. A milestone in the effort of simplifying the implementation of such applications has been the introduction of a service layer, named middleware, which abstracts from the peculiarities of the operating system and HW components. However, the presence of the middleware has not been yet introduced in the design flow as an explicit dimension. This work presents an abstract model of middleware supporting different programming paradigms; it can be used as component in the design flow and allows to simulate and develop the application without doing premature assumptions on the actual HW/SW platform. At the end of the design flow the abstract middleware can be mapped to an actual middleware. The methodology has been analyzed both theoretically and practically with the actual application on a wireless sensor network.

1 Introduction

In recent times, interest in Ambient Intelligence has increased considerably [1]. In this context, typical applications consists of Networked Embedded Systems (NES) which are integrated into human environments and exchange data through communication networks.

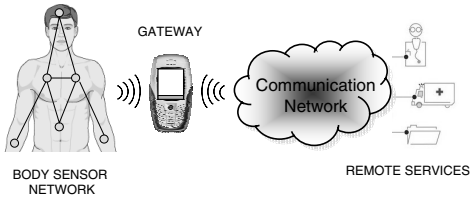

Figure 1. Health-care application of NES.

Fig. 1 illustrates an example of health-care application in which a body sensor network monitors patient's parameters (e.g., temperature, blood pressure, and motion) and transmits them trough the Gateway to a Remote Service to control/monitor the user health (e.g., hospital). Traditionally, many NES applications have been developed without support from system software [1] excepts for device drivers and operating systems. State-of-the-art techniques [2] for NES focus on simple data-gathering applications, and in most cases, the design of the application and the system software are usually closely-coupled, or even combined as a monolithic procedure. Such applications are neither flexible nor scalable and they should be re-written if the platform changes. The use of a middleware layer is a novel approach to fully meet the design and implementation challenges of NES applications. Middleware has often been useful in traditional systems to bridge the gap between the operating system and the application and to make inter-operable distributed processes. All middleware services should respect the constraints of NES, i.e., limited amount of memory, reduced processing power, scalability, and heterogeneity. Furthermore, there are at least four different programming paradigms. Nowadays, the choice of the middleware to design NES application is based on the following criteria: programming skills of the system architect [3] and platform constraints [4]. The down-side of this approach is a more complex design flow. In fact, the same application cannot support different platforms, limiting application re-use, interoperability and scalability.

Various proposals have been made to solve this issue. In the context of software engineering the Model-Driven Architecture [14] shifts the focus of software development from writing code to building platform-independent models. Some works [5] support true interoperability between different applications, as well as between different implementation platforms and ensure scalability of the NES technology by proposing a universal application interface, which allows programmers to develop applications without having to know unnecessary details of the underlying platform. Such works define a standard set of services and interface primitives called SNSP (Sensor Network Services Platform), to facilitate the development of Wireless Sen-

*Research activity partially supported by the FP6-2005-IST-5-033506 ANGEL European Project

978-3-9810801-2-4/DATE07 © 2007 EDAA

sors/Actuators Network applications.

The goal of this paper is to present a middleware-centric design flow for NES, where the middleware plays a decisive role in the design process. This proposal has three main advantages:

(1) It provides a set of abstract services supporting the programming paradigms of different actual middleware implementations: the *Abstract Middleware Services* (AMS). AMS facilitates the design flow by providing an interface which abstracts the specific HW/SW platform and meets the skills of the designer as far as programming paradigm is concerned.

(2) The application can be simulated at early stage of the design flow.

(3) The AMS can be mapped to *Actual Middleware Services* (ACMS) to execute the application on the actual platform; this guarantees the correct trade-off between level of abstraction and efficiency of implementation.

The paper is organized as follows. Section 2 classifies the actual middleware approaches according to their programming paradigms. Section 3 and Section 4 introduce the proposed solution describing the AMS services and the AMS-based design flow. Section 5 discusses the simulation environment cooperating with AMS to simulate the whole NES model and in Section 6 we show how to map the abstract middleware services on an actual middleware. Finally, in Section 7 we experimentally evaluate the effectiveness of the Middleware-centric design flow.

2 Middleware classification

Middleware services can follow different programming paradigms [6].

Database. With this approach, the NES network is viewed as a distributed database where users extract data through SQL-like queries. TinyDB [11] is a query-processing middleware for WSN based on TinyOS.

Tuplespace. This kind of middleware derives from Linda [9], a coordination language for concurrent programming in which processes communicate by reading and writing *tuples* on a shared repository called *tuple space*. Tuples can be seen as vectors of typed values; they are anonymous and thus are selected through pattern matching on their type/value contents. Some examples belonging of this class are: T-Spaces [8] and TinyLime.

Object-oriented (OOM). Applications based on object-oriented middleware consist of distributed objects interacting via location-transparent method invocation. Typically, this middleware offers *1)* an interface definition language (IDL) which is used to abstract over the fact that objects can be implemented in any suitable programming language, *2)* an object request broker which is responsible for transparently directing method invocations to the appropriate target object, and 3) a set of services (e.g. naming, time, transactions, replication etc.) which further enhance the distributed programming environment. Ice [10] is a new object-oriented middleware platform that allows developers to build distributed client-server applications with minimal effort.

Message-oriented (MOM). Message-oriented middleware increases the flexibility of an architecture by enabling processes to exchange asynchronous messages. This approach is quite suitable in pervasive environments such as NES, where most applications are based on events. In this programming model, sources "publish" data to the entire network and interested sinks "subscribe" to specific topics. The network then only forwards them downstream if there is at least one subscriber on that path. Mires [12] proposes an asynchronous communication model that is suitable for event-driven NES applications.

3 Abstract Middleware Services

Any middleware type provides different services (or API) to the application designer. Apart from core features, almost all middleware approaches presented in Section 2 offer several services that simplify the application development by providing access to particular operating systems and HW functionality.

Database Services Database middleware provides the user with a query system which hides distribution issues. We can summarize the services provided by the database middleware in a unique service:

`query` : insertion, update or request of data contained in the distributed architecture.

The following pseudo-code represents a simplified version of an application based on database middleware.

```
string request="SELECT temp
                FROM table WHERE temp>8"
response = middleware.query(request);
```

Tuplespace Services In Tuplespace middleware data is represented by elementary structures called *tuples* stored in the so-called *tuple space*. Each *tuple* is a sequence of typed fields (e.g., <"temperature", 25, XYZ>) and the communication between processes is implemented by writing and reading *tuples* in the *tuple space*. The main services provided by the programming paradigm are:

`read_b(template)` : to read a tuple from the *tuple space* without removing it; the calling process is blocked until a matching tuple appears.

`read_nb(template)` : like the previous service, but in this case the operation returns `null` if no matching *tuple* exists in the *tuple space*.

`take_b(template)` : to read and remove a tuple from the *tuple space*; the calling process is blocked until a matching tuple appears.

`take_nb(template)` : like the previous service, but in this case the operation returns `null` if no matching *tuple* exists in the *tuple space*.

`write(tuple)` : to insert a *tuple* in the *tuple space*.

In *read* and *take* operations, a `template` should be specified to describe the type of the requested tuple. The template itself is a tuple whose field contain either values (actuals) or "wild cards" (formals). Typically, if multiple tuples match a template, the one returned by the read or take operations is selected non-deterministically.

The following pseudo-code represents a simplified version of an application based on tuplespace middleware.

```
while(1) {
     response = mw.take_nb(<"temp",?>);
     if (response!=NULL AND
           response[1] > 800)
           execute_operation;
     endif;}
```

Object-oriented Services A typical object-oriented middleware provides *1)* a mechanism to describe an object interface and to map it to an actual object implemented in common programming languages, *2)* a mechanism to provide the client with a local reference of the remote object, and *3)* a public repository in which instances of the actual object have been registered. Let `Srv` and `SrvImpl` be the name of the object interface and of its actual implementation respectively, then the middleware should provide the following services:

`register(obj,name)` : to register an instance of `SrvImpl` into the public repository and to assign it a public name.

`lookup(name)` : to obtain a local object of type `Srv`.

The following fragment of application uses the service provided by a remote object.

```
Middleware mw=new Middleware(host,port);
Srv mySrv=(Srv)mw.lookup("Service");
mySrv.do();
```

The following pseudo-code represents a fragment of an application which creates an instance of `SrvImpl` and registers it in the public repository.

```
Middleware mw=new Middleware(host,port);
SrvImpl mySrvImpl=new SrvImpl();
mw.register(mySrvImpl, "Service");
```

Message-oriented Services Publish/Subscribe framework is an asynchronous messaging paradigm in which publishers post messages to an intermediary broker and subscribers register subscriptions with that broker. In a topic-based system, messages are published to "topics" or named logical channels which are hosted by a broker. Subscribers in a topic-based system will receive all messages published to the topics to which they subscribe and all subscribers to a topic will receive the same messages. The current programming model uses the following services:

`publish(Message,Topic)` : to publish a message to the topic.

`subscribe(Topic)` : to subscribe to a particular topic.

`on_receive(Message)` : invocated when the subscriber receives a message.

The following pseudo-code represents a fragment of application based on message-oriented middleware.

```
main() {
    Topic topic = "temperature";
    middleware.subscribe(topic);}
on_receive(Message msg) {
    if (msg[1] > 800)
         execute_operation;
    endif;}
```

3.1 AMS implementation

Based on the previous analysis we introduce a set of services and interface primitives, called Abstract Middleware Services (AMS), which should be made available to an application programmer independently on the implementation on any actual middleware. Each programming service, previously described, should be seen as a component of AMS. AMS library has been implemented using SystemC. This allows to simulate each component of the whole distributed application at different level of abstraction by using the Transaction Level Modeling (TLM) library, thus providing an early platform for software development. Following the TLM fashion we have defined three AMS implementations (AMS_1, AMS_2 and AMS_3), usable in different abstraction levels of the application design flow, as shown in Section 4.

4 AMS-based design flow

Fig 2 shows the design flow based on the AMS. A typical NES application is a distributed application composed by a set of interactive modules running on a heterogeneous HW/SW embedded architecture (network nodes interact

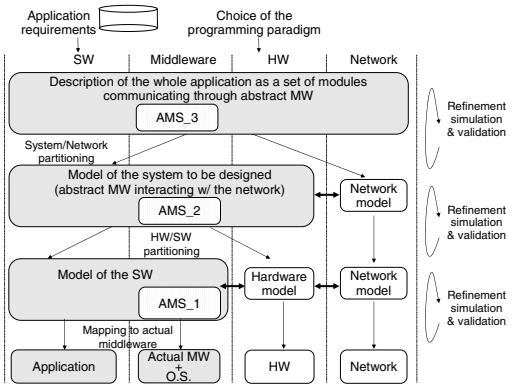

Figure 2. Design flow.

through communication network) to carry out the functionality. During the first step of the design flow proposed in this paper, the designer has to specify the application requirements (performances, functionalities, power consuming, etc.) and choose the programming paradigm; a programming model should substantially support the development of the application hiding hardware and communication issues from the programmer as far as possible. Ideally a programming paradigm allows to program the networked platform as a single "virtual" entity, rather than focusing on individual nodes. The choice of the programming paradigm depends on the application designer skills and on the type/nature of the application.

Therefore, in this phase the NES application will be built using the abstracted services provided by the AMS_3, based on the programming paradigm chosen, to verify and simulate the functional property of the application. Furthermore, because of the separation of the application model from the middleware, the application development can be done in parallel with the design of the HW/SW platform or even without knowledge about the final platform. Then *System/Network partitioning* is applied to this model. Some modules are mapped to network nodes and simulated with a network simulator; an integer number of modules can be assigned to each node. We refer to the model of the node under design as *System model*. Communications between nodes are described as network communications in the *Network model* which reproduces the behavior of network protocols such as TCP/IP or ZigBee/802.15.4. The *System model*, modeled using the services of AMS_2, interacts with the *Network model* (horizontal arrow in Figure 2). In this phase, an performed of the communication protocols, simulated by the *Network model*, can be evaluated. The AMS_2 API services used to design the NES application as the same of the previous design step; however, in this case, the implementation of the AMS_2 services is different to allow the interaction with the network simulator.

Then a traditional design flow is applied to the *System model*, while the different parameters of the *Network model* can be tuned to improve performance. In particular, *HW/SW partitioning* is performed on the *System model* to map functionalities to HW and SW components according to several constraints (e.g., performance, cost, and component availability). SW components interact with HW components (named *Hardware model*) and HW components interact with the *Network model* (horizontal arrows in the Figure 2). Also in this case, the AMS_3 API services as the same of the previous design steps, but the implementation is different to communicate with the network simulator, simulating the *Network model*, and the hardware simulator, simulating *Hardware model*.

Finally, SW functionality is then mapped to the application running on an actual middleware (e.g., TinyDB, Tuplespace, etc.) and its operating system. Synthesis is applied to HW modules to build actual components. An actual network is deployed according to its model. At this phase the NES application can be run on real HW or simulated using s simulating platform, like TOSSIM for WSNs.

5 Simulation environment

One of key advantages of the design based on abstract middleware is that simulation can be done at the early stage of the design flow. With reference to Figure 2 there are different simulation mechanisms and capabilities at the different levels of the design flow.

At the first stage, the whole application is described as a set of SystemC functional modules interacting together through the interface provided by abstract middleware (i.e., AMS_3). Simulation mechanisms are provided by the SystemC simulation environment in which each function and the middleware are considered as concurrent processes.

At the second stage, system/network partitioning has been performed and the *system model* of each node interacts with other nodes through communication links described in the *network model*. At this level a different version of the abstract middleware library (i.e., AMS_2) is used; it provides the same interface to the application code but communications are implemented through packet exchanges. System models are simulated by the SystemC simulator while packet delivery is simulated by a network simulator. A cooperation between the two simulation environments is needed; in particular, the network simulator must provide SystemC with an API to transfer packets from system models to the network and viceversa; for this purpose some approaches are available [13].

At the third stage, HW/SW partitioning has been performed. HW components are simulated by SystemC and interact with the network model as in the previous stage. Also the software is simulated by SystemC; a third version of the abstract middleware library (i.e., AMS_1) is used; it provides the same interface to the application code but communications are implemented through calls to HW func-

tions. These calls will be replaced by an actual operating system in the final deployment stage.

6 Mapping to actual MW

Starting from the NES application designed using the AMS library, the final design step involves a mapping between the Abstract Middleware Services and the Actual Middleware Services (ACMS). The mapping allows to run the application on the actual middleware used by the Networked Embedded System or its simulator (e.g., TOSSIM).

This section describes how to implement the mapping between AMS and ACMS. The actual middleware services are already described in Section 3 classified according to the programming paradigm. It is important to underline that the actual middleware can fall either into the same class of the abstract services or into another class; in the former case, the mapping is very simple while in the latter, the mapping implies a paradigm interpretation and conversion.
This section describes the mapping from tuplespace services to database services.

A tuple T is an ordered set of elements
$T =< e_1, e_2, ..., e_n >$, where n represents the number of tuple elements. Before presenting the mapping rule between tuplespace middleware and database middleware, let us partition the *tuple space* into subsets of homogeneous tuples; then a database table is created for each subset P_i as shown in Fig. 3.

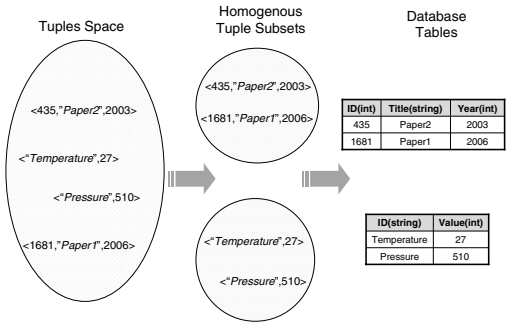

Figure 3. Example of transformation of a tuple space into database tables.

The relationship between the *read* service of the tuplespace middleware and the query of the database middleware can be described as follows.

```
read(<e1,e2,...,en>)
        ⇩
SELECT   *
FROM     Pi
WHERE    column1 = e1 AND
         ...
         columnn = en
```

The WHERE condition has to be composed by the AND operation of the template actual fields, skipping the "wild cards" elements. Tuples can be also extracted from the tuple space using the destructive *take* operation; the corresponding implementation in a database middleware can be described as follows.

```
take(<e1,e2,...,en>)
        ⇩
SELECT   *
FROM     Pi
WHERE    column1 = e1 AND
         ...
         columnn = en
DELETE
FROM     Pi
WHERE    column1 = e1 AND
         ...
         columnn = en
```

Finally, a *write* operation of the tuplespace middleware can be translated into an INSERT or UPDATE command, based on the existence of the record in the table.

```
write(<e1,e2,...,en>)
        ⇩
If ( SELECT *
     FROM   Pi
     WHERE  column1 = e1 AND
            ...
            columnn = en
     UPDATE …
     else
     INSERT …)
```

7 Experimental analysis

The proposed middleware-centric design flow has been applied to an application for the scenario depicted in Fig. 1, in which a mobile terminal (GW) works as a gateway between a body sensor network (BSN) and a remote service (RS). Several BSN monitor the body temperature of a group of people; the GW checks the received data and informs the RS if some sample exceeds a given threshold. This application has been modeled by using the AMS_3 library and then mapped to two different middleware paradigms, i.e., Tuplespace and Database.

The abstract application uses the Tuplespace services of the AMS_3 library. Fig. 4 shows SystemC code for the three actors of the application (BSN, GW, RS). Fig. 4.1 represents the application code running on each sensor node which samples the body temperature and then makes its value available by using the tuplespace `write` service. Fig. 4.2 represents the application code running on the GW which extracts (`take` service) available temperatures and checks for values above 40 degree; in this case, the GW generates an alarm through the `write` service. Fig. 4.3 represents the application code running on the RS to verify whether an alarm has been raised (`take` service). Finally, Fig. 4.4 describes the instantiation of the all actors (50 nodes have been simulated).

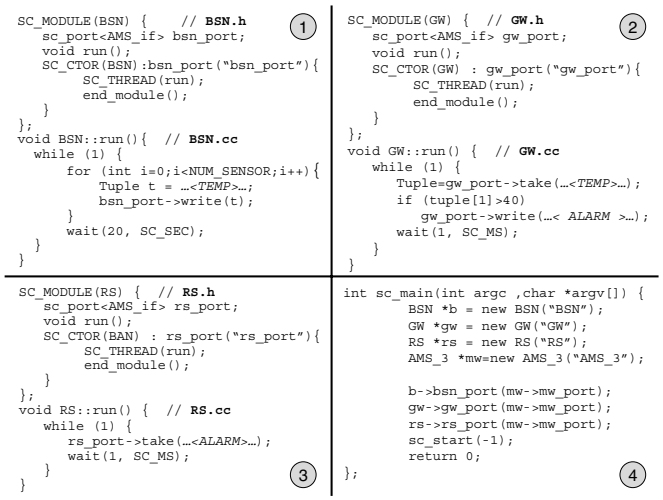

Figure 4. Application described by using the AMS_3 library.

	Code lines	Simulation time [sec.]	MW calls
AMS_3	60	0.37	51300
Tuplespace	151	1.06	51300
Database	170	6.60	66974

Table 1. Simulation performance.

The abstract model has been simulated in the SystemC environment to verify the correct implementation of functionalities. As reported by Table 1 the simulation is faster than real-time (0.37 s for a 100 s simulation). Then, the abstract code has been mapped to two actual implementations based on different middleware paradigms. The Tuplespace application can be used with actual TinyOS motes; the GW and the RS are Java applications interacting with NesC code on sensor nodes through a tuplespace middleware implementation. In this case, a ono-to-one mapping of tuplespace service calls has been done. The application has been simulated in the TOSSIM environment with a network of 50 sensor nodes. The Database application is a distributed application based on Database services. The different actors are Java applications running on Linux operating system and the middleware is based on a MySQL server.

These experiments have shown that *1)* an abstract model of the application can be easily developed and simulated through the middleware services provided by AMS_3, *2)* the mapping process supports the rapid generation of actual implementations for very different platforms (i.e., TinyOS motes and Linux machines), *3)* the simulation of the abstract application is faster than real-time and than TOSSIM simulation allowing the rapid exploration of design alternatives before generating the actual implementation.

8 Conclusions

We have presented a middleware-centric approach for the design of complex distributed applications based on network of heterogeneous embedded systems. Based on the assumption that the presence of a middleware software simplifies the design of distributed applications, we have developed an abstract middleware library providing services belonging to different programming paradigms. The designer can thus rapidly develop a model of the application according to the preferred paradigm; this model can be simulated then mapped to different platforms for actual deployment. Experimental results show that the mapping process supports the rapid generation of actual implementations for very different platforms (i.e., TinyOS motes and Linux machines). Furthermore, the simulation of the abstract application is faster than real-time and than TOSSIM simulation allowing the rapid exploration of design alternatives before generating the code for the actual implementation.

References

[1] Twan Basten, Marc Geilen, Harmke de Groot, "Ambient Intelligence: Impact on Embedded System Design", *Kluwer Academic Publishers*, 2003.

[2] P. Volgyesi, A. Ledeczi, "Component-Based Development of Networked Embedded Applications", *In Proc. of Euromicro conference*, 2002, pp. 68-73.

[3] Kay Rmer, "Programming Paradigms and Middleware for Sensor Networks", *http://citeseer.ist.psu.edu/666689.html*, 2004.

[4] V. Subramonian et al., "Middleware Specialization for Memory-Constrained Networked Embedded Systems", *In Proc. of IEEE Real-Time and Embedded Technology and Applications Symposium*, 2004, pp. 306-313.

[5] M. Sgroi, Adam Wolisz, Alberto Sangiovanni-Vincentelli and Jan M. Rabaey, "A Service-Based Universal Application Interface for Ad-hoc Wireless Sensor Networks", whitepaper, U.C.Berkeley, 2004.

[6] Salem Hadim and Nader Mohamed, "Middleware: Middleware Challenges and Approaches for Wireless Sensor Networks", *IEEE Distributed Systems Online*, 7(3), Mar. 2006.

[7] D. C. Schmidt et al., "TAO: A Pattern-Oriented Object Request Broker for Distributed Real-time and Embedded Systems." *IEEE Distributed Systems Online*, 3(2), Feb. 2002.

[8] T. J. Lehman, Stephen W. McLaughry, and Peter Wyckoff, "T Spaces: The Next Wave." *In Proc. of the Int. Conference on System Sciences*, 1999.

[9] David Gelernter, "Generative communication in Linda", *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 1, pp. 80-112, 1985.

[10] Michi Henning, "A New Approach to Object-Oriented Middleware", *IEEE Internet Computing*, Vol. 8, No. 1, pp. 66-75, 2004.

[11] S.R. Madden et al., "TinyDB: An Acquisitional Query Processing System for Sensor Networks", *ACM Trans. Database Systems*, Vol. 20, No. 1, pp. 122-173, 2005.

[12] E. Souto et al., "A message-oriented middleware for sensor networks", *in Proc. of the Workshop on Middleware for Pervasive and Ad-Hoc Computing*, 2004, Vol. 77, pp. 127-134.

[13] F.Fummi et al., "A Timing-Accurate Modeling and Simulation Environment for Networked Embedded Systems", *in Proc. of the IEEE Design Automation Conference (DAC)*, 2003, pp. 42-47.

[14] J. Siegel, "Why Use the Model Driven Architecture to Design and Build Distributed Applications ?", *in Proc. of the Int. Conf. on Software Engineering*, 2005, pp. 37-37.