

Programming Paradigms and Middleware for Sensor Networks

Kay Römer
Institute for Pervasive Computing
ETH Zurich
roemer@inf.ethz.ch

Abstract

Programming sensor networks is currently a cumbersome and error-prone task since it requires programming individual sensor nodes, using low-level programming languages, interfacing to the hardware and the network, only supported by primitive operating system abstractions. There is a strong need for programming abstractions that simplify tasking sensor networks, and for middleware that supports such programming abstractions. We outline challenges in the design of such abstractions and middleware. Also, we present and discuss currently examined approaches to sensor network programmability.

1 Motivation

Distributed programming abstractions like Remote Procedure Calls (RPC), the Distributed Object Model (DOM), or Distributed Shared Memory (DSM) have traditionally simplified and enabled the implementation of complex distributed systems. These programming abstractions served as foundations for successful middleware architectures such as ONC RPC [13], CORBA [15], or MUNIN [5]. Unfortunately, these abstractions and middleware architectures cannot be simply applied to sensor networks due to the new characteristics and peculiarities of the latter. Existing approaches have to be revisited or new approaches have to be developed to meet the requirement of sensor networks [11, 14, 16]. We will sketch these requirements below.

Programming Paradigm. Sensor networks are used to monitor a wide variety of environmental phenomena. Hence, mechanisms are needed for the specification of

high-level sensing tasks, for automated programming of individual nodes according to a sensing task, and for combining sensory data from individual sensor nodes into high-level sensing results. A programming paradigm should substantially support the development of applications according to this model. In particular, it should hide hardware and distribution issues from the programmer as far as possible. Ideally, a programming paradigm allows to program the sensor network as a single “virtual” entity, rather than focusing on individual nodes. While providing easy to use abstractions on the one hand, the paradigm should allow the efficient implementation of a wide variety of different applications. Typically, there are tradeoffs between the level of abstraction, expressiveness, and the efficiency of implementations.

Restricted Resources. Sensor nodes suffer from limited resources like energy, computing power, memory, and communication bandwidth. Hence, middleware components have to be lightweight to fit these constraints. Middleware should also provide support to dynamically adapt performance and resource consumption to the varying needs of the application, for example, by enabling dynamic tradeoffs between output quality and resource consumption. Since it is anticipated that a sensor network will execute multiple applications concurrently, middleware should also provide mechanisms to optimize resource allocation for overall system performance.

Network Dynamics. Ad hoc networks of sensor nodes may exhibit a highly dynamic network topology due to node mobility, environmental obstructions resulting in communication failures (e.g., a truck driving by), or hardware failures (e.g., depleted batteries, stepping on a device). Middleware should support the robust operation of

sensor networks despite these dynamics by adapting to the changing network environment. One prominent approach is data-centric communication, where network nodes are no longer addressed by unique identifiers, but by specifying the function or data they provide (e.g., “some device in my vicinity that can measure temperature”), allowing for automatic failover to a device with characteristics similar to the failed one.

Scale of Deployments. As sensor nodes become smaller and cheaper, sensor networks are anticipated to consist of thousands or millions of individual nodes. This precludes manual configuration, maintenance, fixing, or upgrading of individual devices due to their huge number. Hence, middleware should support mechanisms for self-configuration and self-maintenance of collections of sensor nodes. In an extreme case, starting from a totally symmetric situation (all devices are identical initially and do not possess unique identifiers), the collection of devices must self-configure in order to achieve an operational state (e.g., set up a network topology, assign tasks to devices, collaboratively merge and evaluate collected data). Although such mechanisms try to achieve a certain global structure or state of the network, it is often impossible to apply global approaches (e.g., flooding the network) to achieve these due to scalability reasons. Fortunately, it is often possible to approximate a global goal by using localized mechanisms, where devices interact only with their close neighborhood.

Real-world Integration. Sensor networks are used to monitor real-world phenomena. Hence, the categories time and space play a crucial role in sensor networks to identify events in the real world (i.e., time and place of event occurrence), to distinguish events in the real world (i.e., separation of events in time and space), and to correlate information from multiple sources. Hence, the establishment of a common time scale and location grid among sensor nodes is an important middleware service. Additionally, many applications are subject to real-time constraints, for example, to have the sensor network report a certain state of the real world within a certain amount of time. Hence, middleware may have to include real-time mechanisms.

Collection and Processing of Sensory Data. The collection and processing of sensory data is a core func-

tionality in sensor networks. Complex sensing tasks often require that data collected at many nodes be fused to obtain a high-level sensing result. However, limited resources preclude centralized data processing, since that requires transferring bulky raw sensor data from many nodes through the network. Instead, sensory data is preprocessed at the source to extract features relevant to the application. Such preprocessed data from multiple sources is fused and aggregated in the network as it flows towards the “user”, further reducing communication and energy overhead. These techniques to some degree blur the clear separation of communication and data processing typically found in traditional distributed systems. Middleware with support for the above data reduction techniques will require means to specify application knowledge and ways to inject this knowledge into the nodes of the network.

Integration with Background Infrastructures. Sensor networks are rarely stand-alone systems. Rather, they are connected to external devices and infrastructures for several reasons. Firstly, this may be necessary for tasking the sensor network, as well as for evaluation and storage of sensing results. Secondly, background infrastructures such as the Internet may provide resources (e.g., computing power, storage) which are not available in the sensor network. Hence, middleware should allow the integration with such background infrastructures, providing a homogeneous view on a wide variety of extraordinarily heterogeneous computing devices. Additionally, sensor nodes might not possess the resources to actually execute specific middleware components. In such cases, it might be a viable option to source out middleware components to the background infrastructure, only keeping minimal functionality on the devices. Note that this minimal set of on-device functionality might change over time in order to adapt to a changing environment, thus necessitating mechanisms for dynamically updating services executing on the devices.

Interfacing with the Operating System. Traditional middleware is typically designed to sit on top of established operating systems, which already provide rich abstractions such as task and memory management. In sensor networks, however, operating system abstractions and concrete operating systems for sensor nodes are currently an area of active research. Hence, the functional sep-

aration and the interface between operating system and middleware is not well understood. Due to resource constraints, it is likely that operating system functionality (e.g., task and memory management) will be rather primitive compared to traditional OS [6]. One possible option is to give up the separation between operating system and middleware and aim for a distributed operating system that unifies traditional OS and middleware functionality.

2 Middleware Approaches

Databases. A number of approaches [3, 9, 12] have been devised that treat the sensor network as a distributed database where users can issue SQL-like queries to have the network perform a certain sensing task. We will discuss TinyDB [9] as a representative of this class.

TinyDB supports a single “virtual” database table `sensors`, where each column corresponds to a specific type of sensor (e.g., temperature, light) or other source of input data (e.g., sensor node identifier, remaining battery power). Reading out the sensors at a node can be regarded as appending a new row to `sensors`. The query language is a subset of SQL with some extensions. Consider the following query example. Several rooms are equipped with multiple sensor nodes each. Each sensor node is equipped with sensors to measure the acoustic volume. The table `sensors` contains three columns `room` (i.e., the room number the sensor is in), `floor` (i.e., the floor on which the room is located), and `volume`. We can determine rooms on the 6th floor where the average volume exceeds the threshold 10 with the following query:

```
SELECT AVG(volume), room FROM sensors
WHERE floor = 6
GROUP BY room
HAVING AVG(volume) > 10
EPOCH DURATION 30s
```

The query first selects rows from `sensors` at the 6th floor (`WHERE floor = 6`). The selected rows are grouped by the room number (`GROUP BY room`). Then, the average volume of each of the resulting groups is calculated (`AVG(volume)`). Only groups with an average volume above 10 (`HAVING AVG(volume) > 10`) are kept. For each of the remaining groups, a pair of average volume and the respective room number (`SELECT`

`AVG(volume), room`) is returned. The query is re-executed every 30 seconds (`EPOCH DURATION 30s`), resulting in a stream of query results.

TinyDB uses a decentralized approach, where each sensor node has its own query processor that preprocesses and aggregates sensor data on its way from the sensor node to the user. Executing a query involves the following steps: Firstly, a spanning tree of the network rooted at the user device is constructed and maintained as the network topology changes, using a controlled flooding approach. The flood messages are also used to roughly synchronize time among the nodes of the network. Secondly, a query is broadcast to all the nodes in the network by sending it along the tree from the root towards the leafs. During this process, a time schedule is established, such that a parent and its children agree on a time interval when the parent will listen for data from its children. At the begin of every epoch, the leaf nodes obtain a new table row by reading out their local sensors. Then, they apply the select criteria to this row. If the criteria are fulfilled, a so-called partial state record is created that contains all the necessary data (i.e., room number, floor number, average volume in the example). The partial state record is then sent to the parent during the scheduled time interval. The parent listens for any partial state records from its children during the scheduled interval. Then, the parent proceeds as the children by reading out its sensors, applying select criteria, and generating a partial state record if need be. Then, the parent aggregates its partial state record and the records received from its children (i.e., calculates the average volume in the example), resulting in a new partial state record. The new partial state record is then sent to the parent’s parent during the scheduled interval. This process iterates up to the root of the tree. At the root, the final partial state record is evaluated to obtain the query result. The whole procedure repeats every epoch.

Mobile Agents. Another class of middleware approaches is inspired by mobile code and mobile agents [4, 7]. There, the sensor network is tasked by injecting a program into the sensor network. This program can collect local sensor data, can statefully migrate or copy itself to other nodes, and can communicate with such remote copies. We discuss SensorWare [4] as a representative of this class.

In SensorWare, programs are specified in Tcl [10], a dynamically typed, procedural programming language.

The functionality specific to SensorWare is implemented as a set of additional procedures in the Tcl interpreter. The most notable extensions are the `query`, `send`, `wait`, and `replicate` commands. `query` takes a sensor name (e.g., `volume`) and a command as parameters. One common command is `value` which is used to obtain a sensor reading. `send` takes a node address and a message as parameters and sends the message to the specified sensor node. Node addresses currently consist of a unique node ID, a script name, and additional identifiers to distinguish copies of the same script. The `replicate` command takes one or more sensor node addresses as parameters and spawns copies of the executing script on the specified remote sensor nodes. Node addresses are either unique node identifiers or “broadcast” (i.e., all nodes in transmission range). The `replicate` command first checks whether a remote sensor node is already executing the specified script. In this case, there are options to instruct the runtime system to do nothing, to let the existing remote script handle this additional “user”, or to create another copy of the script. In SensorWare, the occurrence of an asynchronous activity (e.g., reception of a message, expiry of a timer) is represented by a specific event each. The `wait` command expects a set of such event names as parameters and suspends the execution of the script until one of the specified events occurs.

The following script is a simplified version of the TinyDB query and calculates the maximum volume over all rooms (i.e., over all sensor nodes in the network):

```

set children [replicate]
set num_children [llength $children]
set num_replies 0
set maxvolume [query volume value]
while {1} {
    wait anyRadioPck
    if {$maxvolume < $msg_body} {
        set maxvolume $msg_body }
    incr num_replies
    if {$num_replies = $num_children} {
        send $parent $maxvolume
        exit }
}

```

The script first replicates itself to all nodes in communication range. No copies are created on nodes already running the script. The `replicate` command returns

a list of newly “infected” sensor nodes (`children`). Then, the number of new children (`num_children`) is calculated, the reply counter (`num_replies`) is initialized to zero, and the volume at this node is measured (`maxvolume`). In the loop, the `wait` blocks until a radio message is received. The message body is stored in the variable `msg_body`. Then, `maxvolume` is updated according to the received value and the reply counter is incremented by one. If we received a reply from every child, then `maxvolume` is sent to the parent script and the script exits. Due to the recursive replication of the script to all nodes in the network, the user will eventually end up with a message containing the maximum volume among all nodes of the network.

Events. Yet another approach to sensor network middleware is based on the notion of events. There, the application specifies interest in certain state changes of the real world (“basic events”). Upon detecting such an event, a sensor node sends a so-called event notification towards interested applications. The application can also specify certain patterns of events (“compound events”), such that the application is only notified if occurred events match this pattern. We discuss DSWare [8] as a representative of this class.

DSWare supports the specification and automated detection of compound events. A compound event specification contains, among others, an event identifier, a detection range specifying the geographical area of interest, a detection duration specifying the time frame of interest, a set of sensor nodes interested in this compound event, a time window W , a confidence function f , a minimum confidence c_{\min} , and a set of basic events E . The confidence function f maps E to a scalar value. The compound event is detected and delivered to the interested sensor nodes, if $f(E) \geq c_{\min}$ and all basic events occurred within time window W .

Consider the example of detecting an explosion event, which requires the occurrence of a light event (i.e., a light flash), a temperature event (i.e., high ambient temperature), and a sound event (i.e., a bang sound) within a sub-second time window W . The confidence function is defined as:

$$f = 0.6 \cdot B(\text{temp}) + 0.3 \cdot B(\text{light}) + 0.3 \cdot B(\text{sound})$$

The function B maps an event ID to 1 if the respective

event has been detected within the time window W , and to 0 otherwise. With $c_{\min} = 0.9$, the above confidence function would trigger the explosion event if the temperature event is detected along with one or both of the light and sound events. This confidence function expresses the fact that detection of the temperature event gives us higher confidence in an actual explosion happening than the detection of the light and sound events.

Additionally, the system includes various real-time aspects, such as deadlines for reporting events, and event validity intervals.

Other Approaches. Besides the ones discussed above, there are yet other approaches to programming sensor networks. MagnetOS [2], for example, is a distributed virtual machine, where an object oriented byte-code program is automatically partitioned and allocated to the nodes of a sensor network, such that communication overhead among the distributed components is minimized. EnviroTrack [1] is a middleware specifically tailored to observing mobile physical entities in the environment. As the observed entity moves through the sensor field, nearby sensor nodes form temporary groups to observe the target. Such a dynamically changing group is exposed as an addressable entity to the programmer. The programmer can attach so-called tracked objects to this entity, containing arbitrary data, computation, or actuation. The tracked object is executed by the (dynamically changing) sensor group.

3 Discussion

In the previous section, we presented concrete middleware approaches for sensor networks with different underlying programming paradigms (e.g., database approach, agent-based approach, event-based approach). These paradigms are not new, but required significant adaptation for use in sensor networks. The approaches differ with respect to ease of use, expressiveness, scalability, and overhead. We will discuss the above middleware approaches with respect to these criteria.

TinyDB provides the user with a declarative query system which is very easy to use. The database approach hides distribution issues from the user. Rather than programming individual sensor nodes, the network of nodes

is programmed as a single (virtual) entity. The user is relieved from interfacing with low-level APIs on the sensor node. On the other hand, the expressiveness of the database approach is limited in various ways. Firstly, adding new aggregation operations is a complex task and requires modifications of the query processor on all sensor nodes. But more importantly, it is questionable whether more complex sensing tasks can be appropriately supported by a database approach. For example, the system does not explicitly support the detection of spatio-temporal relationships among events in the real-world (e.g., expressing interest in a certain sequence of events in certain regions). The system also suffers from some scalability issues, since it establishes and maintains network-wide structures (e.g., spanning tree of the network, queries are sent to all nodes). In contrast, many sensing tasks exhibit very local behavior (e.g., tracking a mobile target), where only very few nodes are actively involved at any point in time. This suggests that TinyDB cannot provide optimal performance for such queries. Additionally, the tree topology used by TinyDB is independent of the actual sensing task. It might be more efficient to use application-specific topologies instead.

SensorWare uses an imperative programming language to task individual nodes. Even rather simple sensing tasks result in complex scripts that have to interface with operating system functionality (e.g., querying sensors) and the network (e.g., sending, receiving, and parsing messages). On the other hand, SensorWare's programming paradigm allows the implementation of almost arbitrary distributed algorithms. Typically, there is no need to change the runtime environment in order to implement particular sensing tasks. However, the low performance of interpreted scripting languages might necessitate the native implementation of complex signal processing functions (e.g., Fast Fourier Transforms, complex filters), thus requiring changes of the runtime environment in some cases. SensorWare allows the implementation of highly scalable applications, since the collaboration structures among sensor nodes are up to the application programmer. For example, it is possible to implement activity zones of locally cooperating groups of sensor nodes that "follow" a tracked target. The SensorWare runtime does not maintain any global network structures. One potential problem is the address-centric nature of SensorWare, where specific nodes are addressed by unique identifiers. This may

lead to robustness issues in highly dynamic environments as mentioned in Section 1.

DSWare provides compound events as a basic programming abstraction. However, a complete sensor network application will require a number of additional components besides compound event detection. For example, code is needed to generate basic events from sensor readings, or to act on a detected compound event. DSWare does not provide support for this glue code, requiring the user to write low-level code that runs directly on top of the sensor node operating system. This makes the development of any application a complex task, while at the same time providing a maximum of flexibility. DSWare supports only a very basic form of compound events: the logical “and” of event occurrences enhanced by a confidence function. It might be worthwhile to consider more complex compound events, such as explicit support for spatio-temporal relationships among events (e.g., sequences of events, non-occurrence of certain events). Note that more restrictive compound event specifications can avoid the transmission of event notifications and can hence contribute to better energy efficiency and scalability.

Overall, the examined middleware approaches exhibit a tradeoff between ease of use and expressiveness. While TinyDB is easy to use, it is restricted to a few predefined aggregation functions. More complex queries either require changes in the runtime environment, are inefficient, or cannot be expressed at all. While SensorWare and DSWare support the efficient implementation of almost arbitrary queries, even simple sensing tasks require significant programming efforts. Narrowing this gap between ease of use and expressiveness while concurrently enabling scalable and energy-efficient applications is one of the major challenges in the design of sensor network middleware. It is not yet clear, whether suitable programming abstractions will be inspired by known paradigms as in the three presented examples. Possibly, completely new approaches have to be developed to meet the above goals.

References

- [1] T. Abdelzaher, B. Blum, Q. Cao, Y. Chen, D. Evans, J. George, S. George, L. Gu, T. He, S. Krishnamurthy, L. Luo, S. Son, J. A. Stankovic, R. Stoleru, and A. Wood. EnviroTrack: Towards an Environmental Computing Paradigm for Distributed Sensor Networks. In *ICDCS 2004*, Tokyo, Japan, March 2004.
- [2] R. Barr, J. C. Bicket, D. S. Dantas, B. Du, T. W. D. Kim, B. Zhou, and E. G. Sirer. On the Need for System-Level Support for Ad Hoc and Sensor Networks. *Operating System Review*, 36(2):1–5, 2002.
- [3] P. Bonnet, J. E. Gehrke, and P. Seshadri. Querying the Physical World. *IEEE Personal Communications*, 7(5):10–15, 2000.
- [4] A. Boulis, C.C. Han, and M. B. Srivastava. Design and Implementation of a Framework for Programmable and Efficient Sensor Networks. In *MobiSys 2003*, San Francisco, USA, May 2003.
- [5] J. B. Carter, J. K. Bennet, and W. Zwaenepoel. Implementation and Performance of Munin. In *SOSP 1991*, Pacific Grove, USA, October 1991.
- [6] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System Architecture Directions for Networked Sensors. In *ASPLOS 2000*, Cambridge, USA, Nov. 2000.
- [7] P. Levis and D. Culler. Mate: A Tiny Virtual Machine for Sensor Networks. In *ASPLOS X*, San Jose, USA, October 2002.
- [8] S. Li, S. H. Son, and J. A. Stankovic. Event Detection Services Using Data Service Middleware in Distributed Sensor Networks. In *IPSN 2003*, Palo Alto, USA, April 2003.
- [9] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: a Tiny Aggregation Service for Ad-Hoc Sensor Networks. In *OSDI 2002*, Boston, USA, December 2002.
- [10] J. K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.
- [11] K. Römer, O. Kasten, and F. Mattern. Middleware Challenges for Wireless Sensor Networks. *ACM SIGMOBILE Mobile Computing and Communication Review (MC2R)*, 6(4):59–61, October 2002.
- [12] C. C. Shen, C. Srisathapornphat, and C. Jaikaeo. Sensor Information Networking Architecture and Applications. *IEEE Personal Communications*, 8(4):52–59, 2001.
- [13] R. Srinivasan. RPC: Remote Procedure Call Protocol Specification Version 2 (RFC 1831), 1995.
- [14] J. A. Stankovic, T. Abdelzaher, C. Lu, L. Sha, and J. Hou. Real-Time Communication and Coordination in Embedded Sensor Networks. *Proceedings of the IEEE*, 91(7), 2003.
- [15] S. Vinoski. CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments. *IEEE Communications Magazine*, February 1997.
- [16] Y. Yu, B. Krishnamachari, and V. K. Prasanna. Issues in Designing Middleware for Wireless Sensor Networks. *IEEE Network Magazine*, 2003. To appear.