J Internet Serv Appl
DOI 10.1007/s13174-011-0052-9

SI: FOME - THE FUTURE OF MIDDLEWARE

Mission-oriented middleware for sensor-driven scientific systems

Alan Dearle · Simon Dobson

Received: 25 October 2011 / Accepted: 16 November 2011

Abstract Coordinating the networks needed for modern scientific data collection and control presents a significant challenge: they are typically power- and resource-constrained, operating in noisy and hostile environments, and need to adapt their behaviour to match their operation to their sensed environment while maintaining the scientific integrity of their observations. In this paper, we explore several aspects of managing and coordinating sensor-driven systems. We draw some conclusions about how to design, structure, and implement programming and middleware abstractions for long-lived and adaptive sensor networks.

Keywords Sensor networks · Middleware · Programming models

1 Introduction

Wireless sensor network systems (WSNs) are—and will remain—severely limited in terms of the computational and communications capabilities. While platforms may become more capable, this almost invariably comes at the cost of increased power utilisation, and hence shorter lifetimes. Given that nodes are often deployed in conditions that preclude their recovery, and that each individual node is vulnerable to damage from the environment independently of power degradation, it seems inevitable that, within a fixed budget, designers will continue to prefer to use more, less-powerful nodes in preference to fewer, more powerful nodes.

It is time to re-imagine programming and middleware for resource-constrained, sensor-driven systems, whose requirements are specified en masse for the entire network and where many of the techniques commonly applied in other domains will prove unacceptable. We first examine the issues raised from the current state of WSN programming, and suggest an alternative model that may serve better as WSN systems become more complex and demanding of advanced software.

A. Dearle · S. Dobson (✉)
School of Computer Science, University of St Andrews,
St Andrews, UK
e-mail: simon.dobson@st-andrews.ac.uk

A. Dearle
e-mail: alan.dearle@st-andrews.ac.uk

Published online: 02 December 2011

2 What makes sensor networks different?

Modern computers are surprisingly homogeneous in terms of the core concepts, programming, operating system, and middleware abstractions they present. On the one hand, this represents the successful evolution of concepts with universal appeal and application; on the other hand, familiarity can blind one to important differences.

2.1 Network-centred functionality

The key feature of WSN systems that differentiates them from other application domains is that their functionality lies in the *network* rather than in the *nodes*: in no other domain do individual nodes make less of an *individual* contribution to the overall system behaviour. All the interesting features that one wants to identify in a WSN—robustness, longevity, coverage, and so on—are properties of the *network*, not of the individual nodes. While the same is somewhat true of classical dependable systems, these typically dealt with small numbers of highly capable nodes, which allowed the compositional effects of the network connecting them to be de-emphasised.

Let us take this argument a step further. In programming, most distributed applications we can, to first order, focus on the programming of individual nodes whose functions will then be bound together using a communications model that, while it may be sophisticated in detail, adds little to the individual nodes' functions. In a WSN, the reverse is true: programming must perforce deal with the network, since no matter how sophisticated the individual nodes' programming becomes its impact is insignificant compared to that of the network on the system's overall behaviour and evolution [9, 12].

The vast majority of work in middleware over the past decades has focused on *simplifying interactions between nodes*: typically nothing more complex than pairwise interactions, sometimes augmented with multicast. Middleware has generally adopted a *flat network abstraction* that hides the details, costs, and risks of failures of routing between nodes. Moreover, at least in the case of object, and service-oriented middleware, it has considered the *network to be non-computational*, responsible only for transferring uninterpreted byte streams and simple records without in-flight processing or provenance capture. (While message-oriented middleware offers the promise of such processing, its implementations often do not deliver it and applications seldom make critical use of it.) Finally, it has adopted a *simple model of failure* in which failures are considered exceptional, transient, largely independent, and remedied out-of-band by some mechanism external to both the middleware and the process it is supporting.

2.2 Abstracting the network

The most basic requirement of a WSN is to deliver sensed data to external recipients, in particular to sink nodes. To do this in a traditional programming environment might look something like this:

```
INetSocketAddress sink
    = new INetSocketAddress(...);
if( sink != null ) {
    Socket s = new Socket( sink );
    if( s != null ) { int sent = write( sink,
                                        data );}
}
```

We refer to this model as the *flat network model* in which the network is treated as fully-connected over which routes may be abstracted. In many systems, the node-level code is structured as a tower of layers and abstractions. However, this model contains a number of implicit assumptions about the networked environment model: it assumes

1. That it is possible to abstract over a route to some other node (such as the sink);
2. That data is delivered reliably and the number of bytes delivered to the endpoint can be reported; and
3. That data is being delivered as an un-interpreted message to sink, with no in-network computation over it.

Abstracting over routes has served us well. However in a wireless environment, node neighbours may come and go due to failure, mobility and environmental conditions and may offer widely differing qualities of service between different links and even over a single link over time. Furthermore, the *write* operation above reports the number of bytes delivered to the endpoint, implying bi-directional data transmission. This may not be possible over a single link, far less a multi-hop route in a WSN.

This model does not express anything about the budget of the operation or what to do in the case of failure. Traditionally network code is governed by time-outs that determine how long to try some operation for before giving up. In a multi-hop environment, it is not clear how to set the values of the time-outs. Clearly, to establish a 5-hop connection is going to take longer than a 2-hop one, but since the number of hops has been abstracted over using the flat network model it is unclear where in the system has the necessary control information. Furthermore, since network operations are expensive, it may be desirable to set a resource budget on each operation: a *global* budget since routing will accrue costs to other nodes. It is worth noting again that it may be commonplace for a node to be able to communicate with another node, but for that node not to be able to reply.

Even ignoring the global budget problem, we do not have mechanisms for determining the budget of individual (network) operations. Route formation and maintenance are complex tasks involving the maintenance of data structures and the transmission of many messages. It is difficult to assign these costs to different individual operations.

The example above also assumes that data is being delivered as an un-interpreted message to sink, with no in-network computation. This assumption results in nodes that are close to sinks necessarily having to process more data than those that are further away—and, therefore, be more likely to expire early resulting in a network with severely diminished capacity. In-network computation can (but does not necessarily) reduce this issue. This approach has been demonstrated in Flask [8], which provides a "staged" functional model in which target code is structured using higher-order operations that are compiled-out of the final target binary. It is therefore possible to write something like:

```
deliver( aggregate( produce() ) )
```

in which the *produce* code runs on the performing sensing, the *aggregate* code runs in network and the *deliver* code runs at the sink.

2.3 The single and unchanging application

The model of computation advocated by Flask is essentially that the whole network is a single large functional

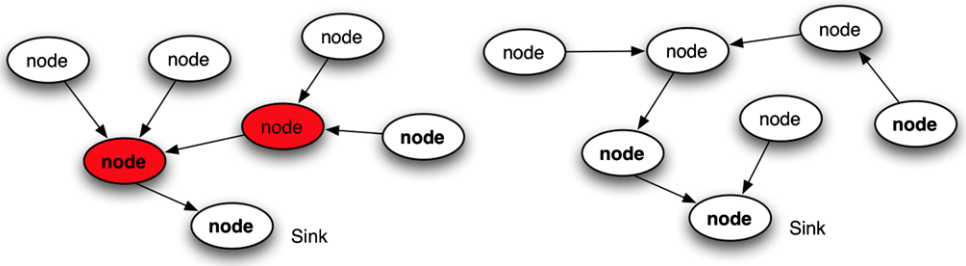

Fig. 1 Routing graphs need not be bidirectional or stable over time

program that never changes over the system's lifetime. We call this model the *single unchanging application abstraction*. This model is also followed by systems such as TinyDB [7], which describe the system as a single query. Both approaches are declarative, although this need not be the case—there is no reason why computation stages should not include computation over state, and both approaches generate rather less declarative nesC programs.

The single unchanging applicationabstraction has a number of intrinsic features:

1. It is possible to describe the application objectives in a single specification.
2. The topology of the network is known a priori.
3. The application objectives are compiled away in the code on the nodes.
4. Mechanisms exist for deploying the components onto nodes.
5. Appropriate communications infrastructure is available to achieve the required data transmission, synchronisation and choreography of the application.
6. No evolution of purpose takes place.

The necessity of describing the application objectives in a single specification is not always a bad thing: indeed, this has been the centre of traditional software engineering for many decades. However, it somewhat implies that each WSN has a single, fixed task. It seems likely that in the future we will have more than one application running concurrently on collections of nodes, especially for systems where the deployment of nodes is complex and their retrieval infeasible, as for many environmental sensing applications.

The ability to locate computation in appropriate geographical locations requires a priori knowledge of the network topology. This is also a requirement of SNEEql [1], which performs static *wherescheduling* to decide where to place the computation required to implement query fragments. We contend that this is highly undesirable in a WSN environment.

Consider the network topologies shown in Fig. 1 containing the same number and geographical positioning of nodes. It would not be uncommon in deployments for communications to be bi-stable between the two topologies due to environmental conditions. However, in the left network computation such as database joins or averaging could be usefully performed at the red nodes where as in the right diagram it could not.

Single-unchanging-application systems compile-away the application objectives as part of the compilation process leaving dumb code to be executed on the nodes. This leaves no opportunity for autonomous adaptation. Knowledge about how nodes are expected to behave is compiled statically into the implementation of *all* the nodes in the system. This shared knowledge tightly defines the operating parameters of every node leaving little scope for adaptive behaviour.

WSN systems have had poor support for deploying code onto nodes, although some sophisticated techniques have been developed to side-step this [6]. Many systems are intended to have code deployed prior to the nodes being placed into the environment to be sensed. In many WSN systems the entire application stack including application code, operating system, and networking layers are compiled into a single image, which precludes individual components from being replaced. Some recent systems [4, 11] have supported the run-time replacement of components. However, not all permit code to be shipped across the network. It is intrinsically expensive to ship code since, as discussed above, WSN radios permit the transmission of relatively small packets with an associated high transmission cost. A second problem is that of bi-directionality. For example, in the networks shown in Fig. 1 above, it may be possible for the nodes to transmit through the routing graphs to the sink but for the sink to be able to communicate back to the nodes. In such cases, dynamic deployment is not possible.

The last major problem with the single-unchanging-program abstraction is the assumption that no evolution of purpose takes place. In this model such stability is intrinsic—a high-level specification leads to a program, which is compiled-down using various technologies into fixed code and a topology that runs on a network whose properties are known in advance. We believe that this model is fundamentally flawed since the network topology, the set of operation nodes, and the software placed upon them must be able to evolve to respond to environmental changes, node, and routing failures.

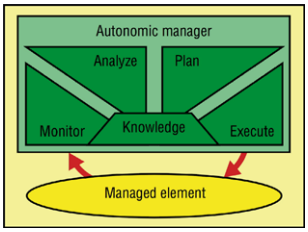

Fig. 2 The autonomic management cycle of a single node, from [5]

2.4 Locating autonomic behaviour

The autonomous management cycle of Kephart and Chess [5] (Fig. 2) supports a number of aspects of "self-*" properties including management, configuration, optimisation, healing, and protection [3].

A fundamental issue with respect to wireless sensor nets is where the autonomic cycle is executed and how the monitoring is performed. Many would advocate performing this at every level in the system: within the entire WSN system, within nodes, and within software components. We examine each of these in turn.

Autonomic management of the entire system is perhaps the most difficult to manage. In order to monitor, information about the status of all the managed elements must be collected. The first question is where this should be collected. The obvious place is at the sink(s) (if there are any) since they are often less resource constrained than the other nodes. It is also more likely that monitoring data will be deliverable to the sink node(s). However, the choreography of change may be difficult (or impossible) from the sink node due to the inability to address nodes (through lack of bi-directional links). Assuming that system change can choreograph from somewhere inside or outside the network, the synchronisation of change presents a further challenge. Mechanisms are required that permit the system to gracefully move from one global state to another. This is difficult in the face of inexact information, unreliable communications, and clock skew.

The analysis phase of autonomic management requires some model of the desired system state. This model must describe (at least) the desired state in terms of resources including software components and physical hosts, the desired state of the network, relationships between hosts and components, and constraints over these. Similarly the collected monitoring data must contain similar levels of detail so that the autonomic planning can be carried out. Such monitoring is a relatively energy-intensive sensing task in itself, and there may not be available computational or network resources to carry it out. Thus, deciding the frequency at which such monitoring occurs is itself a complex task worthy of its *own* autonomic management.

By contrast with system-level autonomic management, node- and component-level management is relatively easy, resource constraints aside. The major problem with such changes is that node level understanding is required of how local changes will affect the entire system. For example, changes to the MAC level protocols on a node cannot be made without complementary changes on the nodes in which the node is in communication. This is similarly true for synchronisation, buffering, and message formats.

There is a need for the individual nodes and components to have a *system model* of how they relate to the entire system. Without this, they cannot autonomously change their behaviour and remain compliant with the requirements of the rest of the system. Similarly, component level changes must have a model of node level behaviour if any consistent changes are to be made.

3 Toward an alternative model

We can step back and ask two questions: What functions do we want to be provided by the operating systems, middleware, and languages on WSNs, and what system and language structures best provide these functions? We believe that answering these questions allows us to re-imagine how we should provide system and development support for the next generation of WSNs.

3.1 No a priori abstractions

For most systems, programming abstractions are composed using static layering where each layer presents new, extended, or simplified concepts in terms of those provided in the layers below. This can lead to redundancy and conceptual mis-match that, while acceptable on larger systems, become problematic when the abstractions are fundamentally inappropriate—for example by abstracting over failure, which prevents applications from dealing with such failures intelligently. It is possible to adopt a more modern, component-oriented approach to systems in which individual features are defined independently along with statements of the dependencies and provided services. This principle can be applied to applications and device drivers—but also to protocol handlers, and even programming languages and type systems.

Taking a more modern approach makes it easier to evolve software over time. This has been achieved in the Lorien system [11], which permits arbitrary components to be loaded and unloaded dynamically (including over the radio). Composition operators are made available for inspecting the manifest of available components, discovering and resolving dependencies between components and for dynamically instantiating, stopping, and deleting them. Components can only be instantiated if their dependencies are satisfied. This system does not provide high-level type guarantees, but such mechanisms could be added to it.

Another approach to component reconfiguration is provided by the Insense language [2], which models the world as active components that only communicate via typed channels. This gives components complete independence of each other. In the Insense model, message send is the only form of IPC. A consequence of this is that threads always reside within the component in which they were created. This has the advantage that components can be replaced independently of each other without concerns about inter-component thread state such as that stored on stacks. Each Insense component has a top-level behavioural loop, which controls the component's behaviour. This model fits well with having the scheduling of components controlled by a resource-aware scheduler.

The replacement of arbitrary system components such as MAC level drivers makes it difficult to make guarantees about the behaviour of sub-systems or even their interface. One answer to this potential problem is through the mission specification. If a mission specification specifies that some piece of functionality is required, the evolutionary process on nodes can be constrained to prevent changes that violate that specification. Enforcement of this is in itself problematical since, in the limit, it is intractable to determine the behaviour of an individual component. One solution to this problem might be code that carries certificates that specify their behaviour and only permit components to be replaced with others that carry appropriate certificates.

3.2 Introspection

The next requirement of a middleware system is to provide the ability to introspect. Without the ability to introspect, in-system evolution is not possible, the only evolution possible is externally imposed heteronomy based on reported information or observations about the system's behaviour. There are a number of system attributes that require introspection:

1. The node state, including the state of the devices, battery, the hardware, and software;
2. The mission or missions on which the node is employed; and
3. The local network environment, including the node's neighbours.

Node level software needs to both introspect and react to the node's network environment. It should be able to discover the nodes neighbours and be alerted to changes in the neighbourhood set. However, such discovery is potentially expensive in terms of energy usage with a trade-off existing between highly accurate information with high-energy usage and less timely information and lower energy usage; if the sampling rate is too slow it may be that events such as node arrival and departure are missed entirely. The parameters governing the introspection of the network environment therefore need to be controllable by software.

3.3 Capturing and understanding mission

Adaptation and evolution are crippled if they must occur only at the behest of (and using information known reliably to) a central autonomic controller. Breaking this restriction requires that the required system-wide behaviours must be defined using explicit *mission specifications*made available to the network at run-time. Such a specifications permit top-level constraints, trade-offs, and adaptive strategies to be captured explicitly and used to inform software deployment and evolution in a well-founded manner. A mission specification defines the *envelope* within which deployed software may autonomously evolve, and constrains heteronomous evolution to respect timing, distribution, and other constraints. The mission is expressed at a high level in terms of abstract "mission components" that represent common design patterns for WSNs (such as signal processing or collection trees). Mission components can be composed resulting in composition of both their functionality and their behavioural envelopes.

The mission specification must be available throughout the system's lifetime to all the processes that operate within it: for example, to processes performing both intra- and inter-node evolution; to processes making decisions about the locations of computational elements; and to those making routing decisions and configuring MAC-level hardware components.

3.4 Building belief

We have argued above that wireless sensor nets are inexact environments, in the sense both of their noisy control inputs and their limited knowledge of their own (and their environment's) state over time. This suggests that nodes must base many decisions on their *beliefs* about these factors, rather than about their *knowledge* of them—and that they must function in or recover in situations where this belief turns out to be incorrect. The example given above serves to illustrate: the more frequently the network environment is scanned, the higher the confidence in the accuracy of the reported set of nodes in the network environment, but at the cost of additional power and communications budget. Another example is in the transmission of data to a sink: at one extreme, the synchronous confirmation of packet delivered to a sink may result in high levels of belief about message delivery; periodic confirmation, perhaps via management reports, would result in lower degrees of certainty; and at the other limit would create a fully asynchronous best-effort-delivery event system.

In both these examples, it can be seem that the degrees of belief are related to the power budget expended on the maintenance of those beliefs. In practice, a compromise must be made between the costs of maintenance of the degree of belief required.

Belief techniques can be applied at the semantic level too, by making use of models of expected observations. If two adjacent nodes report radically different ambient temperatures, for example, then one might reduce the level of belief one places in their accuracy; on the other hand, this may be *expected* behaviour for nodes on opposite sides of a wall or subject to some other environmental factor. Conversely, temperature measurements that do not vary add little information to that already collected and could perhaps be discarded. The use of models alongside missions can thus be used to improve the performance of a WSN both at the semantic and the operational levels.

The encoding of mission may be used to inform requirements for communications. For example, a system comprising nodes attached to animals that is required to detect encounters with other animals requires fast node detection and bi-directional communications between adjacent nodes, with no other dissemination mechanisms required. By contrast a system monitoring an agricultural environment may require a relatively static uni-directional diffusion tree with no acknowledgements of message delivery. Each of these scenarios has quite different requirements on the per-node communications stack.

The initial configuration of communications stacks should therefore be based on the mission specification. The specification would set out the operational envelope of the communications stack such as initial MAC protocols, sampling frequencies and routing protocols. The initial implementation could be changed if the operational environment were found to be different from that expected or if it violates the constraints specified in the mission specification.

Some changes need to choreographed across the entire system. For example, changing the routing protocol from an AODV-based [10] approach to a clustered system would require much inter-node synchronisation. Thus, certain changes may be precluded based on the system information available by introspection. For example, if a node had never received any messages from its (expected) neighbours, it may alter the network protocols it were using or alter parameters that govern them; if, however, the node regularly receives messages from its neighbours some agreement protocol may need to be followed in order to effect a change. Note that in either case information from the lower levels of software needs to be made available to the decision-making processes.

3.5 Building provenance

Closely related to the idea of belief models is that of provenance. When a sensor network reports that the average temperature in some area is 21°C, the provenance of this data needs to be understood. Of interest are the accuracy of the sensing devices, the number, and spread of the devices that have contributed to the result, the node instances that have contributed to the result, the standard deviation, mean, modal values, and other derived signals, how recently the data was collected, the computation that has been applied to the raw readings before they are reported. To be able to report such information requires provenance considerations to be built in from the ground up in the same manner as belief systems.

Again, the encoding of missions and models helps with these tasks, since they help to inform decisions on data handling and to record the fact that these decisions have been made. A strong model of expected temperature variation may allow (or encourage) the middleware to discard outlying values on the grounds that they are likely to be noise (and so reduce communications overheads), but may also record that this optimisation has been made for future analysis. It is important to remember that the design of middleware is not neutral with respect to the scientific data collected, and may experiments are exquisitely sensitive to the processing that occurs on data streams. It may not be a problem that outliers are discarded, for example, but it may compromise later statistical analysis if the *fact* of this discarding is not available.

3.6 Taking and relinquishing control

Finally, we believe that the use of components can be extended throughout the system. Components can be used to provide *all* aspects, including scheduling, type matching, language features, memory management, and other system functions normally regarded as fixed.

Why would one do this? Any fixed choice embodies a collection of beliefs and decisions about the costs and benefits of different design choices. In modern systems, such choices can be made once and fixed, since the systems are so powerful that discrepancies between what is provided and what is optimal will not have a critical impact. We simply do not believe this to be true for WSNs: a single-application system need not pay the penalty for a coercive scheduler that could be replaced by a co-operative scheduler, while conversely a system with time-sensitive interrupts might require the opposite. An individual component is unlikely to be impacted by either choice, whilst the system's overall performance, longevity, and stability might well be.

4 Conclusion

We have explored some of the factors influencing the practice of developing software for sensor networks for extended scientific use in the field. While current systems have allowed many exciting applications, we firmly believe that

the future of WSNs will involve complex design and management choices being made in the face of uncertain information, overlapping, and evolving missions. These cannot be addressed without significant improvement in the middleware, programming languages, and software engineering techniques being deployed.

Our experiences to date suggest that component-based techniques can be a significant asset in defining flexible WSNs. These need to be augmented with new developments in autonomic systems architecture. Specifically, it would seem to be advantageous to move descriptions of the system's architecture and mission down into the systems level to inform and control adaptations over time, and to ensure that the systems evolve within an acceptable mission envelope. Additionally, we need to make improvements in the ways we handle uncertainty and failure, and in the making of decisions in the face of compromised inputs.

These factors are by no means unique to WSNs, and it would be an interesting exercise to see to what extent *all* complex distributed systems would benefit from some of the techniques we have advocated. It is certainly the case that the next generations of many systems will feature uncertainty, evolution, distribution, and ubiquitous failures to be managed. It would be fitting if approaches envisioned for low-resource devices—copying in many ways the constraints of far earlier generations of computing platforms—were to have implications for a wide range of future systems.

References

1. Brenninkmeijer C, Galpin I, Fernandes A, Paton N (2008) A semantics for a query language over sensors, streams and relations. In: Proceedings of BNCOD, pp 87–99
2. Dearle A, Balasubramanian D, Lewis J, Morrison R (2008) A component-based model and language for wireless sensor network applications. In: Proc 32nd annual IEEE international computer software and applications conference (COMPSAC 2008), pp 1303–1308
3. Dobson S, Denazis S, Fernández A, Gaïti D, Gelenbe E, Massacci F, Nixon P, Saffre F, Schmidt N, Zambonelli F (2006) A survey of autonomic communications. ACM Trans Auton Adapt Syst 1(2):223–259
4. Dunkels A, Finne N, Eriksson J, Voigt T (2006) Run-time dynamic linking for reprogramming wireless sensor networks. In: Proc ACM SenSys, pp 15–28
5. Kephart J, Chess D (2003) The vision of autonomic computing. IEEE Computer 36(1):41–52
6. Levis P, Culler D (2002) Maté: a virtual machine for tiny networked sensors. In: Proc 8th ACM international conference on architectural support for programming languages and operating systems, October 2002
7. Madden S, Franklin MJ, Hellerstein JM, Hong W (2005) TinyDB: an acqusitional query processing system for sensor networks. ACM Trans Database Syst 30(1)
8. Mainland G, Morrisett G, Welsh M (2008) Flask: staged functional programming for sensor networks. In: Proceedings of ICFP
9. Mottola L, Picco GP (2011) Programming wireless sensor networks: Fundamental concepts and state of the art. ACM Comput Surv 43(3)
10. Perkins C, Belding-Royer E, Das S (2003) Ad-hoc on-demand distance vector (AODV) routing, RFC3561, IETF, July 2003
11. Porter B, Coulson G (2009) Lorien: a pure dynamic component-based operating system for wireless sensor networks. In: Proc MidSens, pp 7–12, December 2009
12. Sugihara R, Gupta R (2008) Programming models for sensor networks: a survey. ACM Trans Sens Netw 4(2)