PHILOSOPHICAL TRANSACTIONS OF THE ROYAL SOCIETY A

Phil. Trans. R. Soc. A (2012) **370**, 68–84
doi:10.1098/rsta.2011.0330

Review

Operating systems and network protocols for wireless sensor networks

By Prabal Dutta[1,*] and Adam Dunkels[2]

[1]*Computer Science and Engineering Division, University of Michigan, Ann Arbor, MI 48109, USA*
[2]*Swedish Institute of Computer Science, PO Box 1263, 164 29 Kista, Sweden*

Sensor network protocols exist to satisfy the communication needs of diverse applications, including data collection, event detection, target tracking and control. Network protocols to enable these services are constrained by the extreme resource scarcity of sensor nodes—including energy, computing, communications and storage—which must be carefully managed and multiplexed by the operating system. These challenges have led to new protocols and operating systems that are efficient in their energy consumption, careful in their computational needs and miserly in their memory footprints, all while discovering neighbours, forming networks, delivering data and correcting failures.

Keywords: operating systems; network protocols; sensor networks

1. Introduction

Sensor networks are inherently communication-centric systems. The sensor nodes form wireless networks through which sensor readings can be transported or where nodes can act directly on data communicated from their neighbours. For sensor nodes to build networks, they need network protocols. The network protocols define the sequence of steps for the nodes to take in order to communicate with each other as well as the message formats. Sensor network communication is made challenging both by the physical communication environment, which is characterized by frequent and unpredictable bit errors as well as frequently changing topologies, and by the resource constraints of the individual nodes.

For network protocols to operate, an operating system that implements the protocols runs on every node. The operating system manages the resources on each node, provides a layer of abstraction for the hardware, and gives the system developer a programming interface that allows applications to be efficiently implemented. The severe resource constraints, the diversity in hardware platforms and the novelty in applications make sensor network design a challenge.

*Author for correspondence (prabal@eecs.umich.edu).

One contribution of 11 to a Theme Issue ‘Sensor network algorithms and applications’.

2. Sensor network operating systems

Sensor networks have severe resource constraints in terms of processing power, memory size and energy, while operating in a communication-rich environment that interfaces both with the physical world and with other sensor network nodes. The operating system must efficiently manage the constrained resources while providing a programming interface, i.e. allow system developers to create resource-efficient software.

An operating system multiplexes hardware resources and provides an abstraction of the underlying hardware to make application programs simpler and more portable. Unlike general-purpose computers, which have settled for a number of semi-standardized hardware architectures, sensor network hardware is extremely diverse in terms of processor architectures, communication hardware and sensor devices. This makes operating system design for sensor networks a challenge.

In the sensor network research community, several operating systems have been developed, with each offering a different solution for the fundamental problems. TinyOS and Contiki are perhaps the two most well-known systems. TinyOS defines its own programming language called nesC [1], an extension to the C programming language, whereas Contiki uses standard C. Mantis [2], SOS [3] and LiteOS [4] are also widely cited sensor network operating systems.

Operating systems for sensor networks share some characteristics with real-time operating systems for embedded systems. Like sensor network nodes, embedded systems also often have severe resource constraints. But unlike embedded systems, sensor network nodes must interact both with the physical world and with each other: sensor networks are highly communication-intensive systems. This communication intensity adds additional challenges in terms of resource management and operating system structure.

(*a*) *Fundamental problems*

The fundamental problem that an operating system addresses is that of resource allocation. A sensor network node has a limited set of resources in terms of processor time, memory, storage, communication bandwidth and energy. The role of the operating system is to efficiently manage the available resources.

The operating system also provides a system programming interface to developers. This interface must be easy to use for system developers while providing efficiency. This results in additional constraints to the way the operating system can be designed.

(*b*) *Sensor network node hardware*

A sensor node (figure 1) consists of sensors and actuators, which interact with the physical world around the sensor node; a microcontroller, which interacts with the components and executes the software; a communication device, which typically is a radio; and a power source, which often is a battery but which also can be an energy-scavenging device such as a solar cell. Additionally, the sensor node may also contain secondary storage, such as on-board flash memory. Unlike general purpose computers, sensor network nodes do not have support for memory hierarchies, multiple protection domains or multi-level caches.

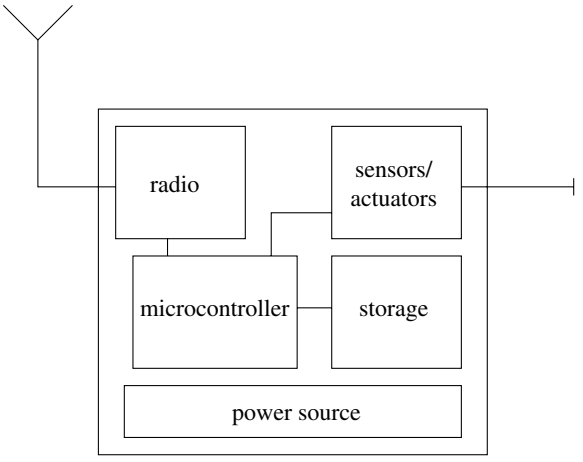

Figure 1. The hardware of a sensor node consists of a communication device, typically a radio, a microcontroller, a set of sensors and actuators and a storage device, typically a flash chip.

Typical hardware platforms for sensor network nodes have processing speeds of the order of a few megahertz, memory size of the order of hundreds of kilobytes and must run with less than 1 mW of power. Although Moore's Law has somewhat relaxed the resource limitations over the past 10 years, it has primarily driven the hardware development in the direction of smaller, less expensive hardware platforms and lower power draws. With many sensor network applications requiring extremely low-cost devices, hardware development is unlikely to yield any extensive improvements in resources for the foreseeable future.

All hardware devices draw power when active, but their activity patterns are different. The microcontroller draws power when it is executing instructions and the sensors draw power when sensing physical phenomena. The communication device draws power both when it is transmitting and receiving data and when it is in idle mode, listening for messages from neighbours. The storage device typically draws power only when it is actively read from or written to and not when it is idle. To make the discussion concrete, table 1 contains the empirical power draws of the components of the Telos sensor node platform [5].

(c) *Concurrency and execution models*

An operating system must manage processor time so that each application gets its fair share. In a sensor network node, multiple activities may happen concurrently with respect to each other: sensor readings are collected from the on-board sensors; they are processed by the microcontroller and possibly stored in secondary storage and are transmitted over the communication device; communication from neighbouring nodes is received and forwarded to other nodes, and timed events occur. The operating system must manage this concurrency in a way which is both resource efficient and easy to understand for the system developer.

Table 1. The power consumption of hardware components of the Telos sensor network node.

component	power draw (mW)
microcontroller, sleeping	0.163
microcontroller, active	5.40
flash read	12.3
flash write	45.3
radio transmit	58.5
radio listen	65.4

The execution model of an operating system determines the way concurrent applications execute. Sensor network nodes operate in a highly concurrent environment and with severe resource constraints. Many sensor network operating systems therefore follow an event-driven execution model. In the event-driven execution model, the primary unit of execution is the event handler. An event handler is invoked in response to an external or internal event. Examples of external events are an incoming message from the communication device and a sensed phenomenon from a sensor. An example of an internal event is a timer that expires.

An alternative to the event-driven model is the multi-threaded model, which is also the most commonly used concurrency model in general purpose operating systems. Under the multi-threaded model, applications are defined as multiple threads that run concurrently. The threads block when waiting for external events. Operating systems such as Mantis [2] make use of the multi-threaded model.

One of the primary benefits of the event-driven model over the multi-threaded model is memory efficiency. Since every event handler returns directly to the operating system, the system does not need to keep track of its state after the invocation is finished. By contrast, in the multi-threaded model, each thread must maintain memory for its stack. Much of this memory is unused, but must be kept free in case the thread needs to use it. Under the even-driven model, only one stack is needed, thereby reducing memory requirements.

When an event occurs in an event-driven system, the operating system finds the correct event handler to handle the event and invokes it. Event handlers have run-to-completion semantics: the event handler must quickly perform its action and return control back to the operating system. This approach works well for simple event handlers, which do their task quickly and return to the caller, but may be troublesome for more complex event handlers that need to wait for multiple events before continuing. Such event handlers must be split into multiple handlers since each event handler must run to completion. The developer must define a state machine that is driven by the event handlers. Research has shown that such state machines typically follow a set of simple patterns that correspond to how developers write programs under the multi-threaded paradigm [6,7].

The trade-off between memory efficiency and programmer complexity in the event-driven and the multi-threaded models has led to the development of several hybrid models. The protothread model [6] provides a sequential flow of control,

like the multi-threaded model, but without the overhead of multiple stacks. A protothread is a stackless type of thread that provides a conditional blocking wait primitive that allow programs to execute a blocking wait without a separate stack for each protothread. Another approach is to run multiple threads on top of an event-driven kernel, which allows the system developer to choose which execution model to use, depending on the needs of the application program. In the sensor network field, this hybrid threading model was first used in the Contiki operating system and was later improved upon in the TinyOS system [8].

(*d*) *Memory allocation*

On sensor nodes, the size of the memory is constrained on both physical and practical grounds. The size of the memory is determined by the number of transistors that hold the contents of the memory. This in turn affects both the power needed to maintain the memory contents and the manufacturing cost of the chip. Both limit the size of the memory used on sensor network nodes.

The memory is split into two parts: the static part, which contains the program code, and the dynamic part, which contains run-time variables, buffers, data and the stack. The static part is typically stored in read-only memory (ROM), whereas the dynamic part is held in random access memory (RAM). Because of the physical characteristics of existing microcontroller architectures, the RAM has a higher power draw than ROM and requires a larger physical chip area. For this reason, the RAM is typically smaller than the ROM. For example, the Telos platform has 48 kb of ROM and 10 kb of RAM [5]. Moreover, unlike general purpose computer systems, sensor node microcontrollers do not have memory indirection or memory protection mechanisms.

The fundamental problem that memory allocation mechanisms must handle is memory fragmentation. Memory fragmentation is when unused memory is scattered across multiple memory regions that are not contiguous. When memory is fragmented, allocations may fail despite the total amount of unused memory being larger than the allocation.

To avoid fragmentation, operating systems for sensor nodes typically have avoided dynamic memory allocation. Instead, all memory has been statically allocated. For dynamic allocation needs, the system developer must pre-allocate static buffers, which may be used at runtime. This allows the system developer to understand the total memory requirements of the system beforehand and reduces the risk of the system running into a fatal fragmentation situation at runtime.

(*e*) *Energy*

Sensor networks are typically battery operated. Since each battery has a fixed amount of energy, the power draw of each node effectively determines its lifetime. Energy is therefore a critical resource. As seen in table 1, the power draw of individual hardware components may differ by the order of magnitudes. Energy management is an essential service of the sensor node operating system.

To reduce the power draw, the operating system must switch off unused components as often as it is possible to do so. The microprocessor is switched to sleep mode when no application is running. When an event occurs, such as a sensor reading taking place or a timer firing, the microcontroller is woken up.

The operating system then invokes the appropriate application program. The communication component is difficult for the operating system to manage, as the component must be switched on for communication to occur. Communication energy management is therefore handled by a separate radio duty cycling mechanism.

Operating systems also may track energy consumption. For this, both hardware- and software-based approaches have been developed. Quanto uses a hardware-based energy meter coupled with a software-based power state and activity tracking system for TinyOS. The total time and energy measurements are dissected and attributed to hardware peripherals or logical activities [9]. The Contiki and Pixie operating systems use an entirely software-based approach based on power state tracking, in which the system tracks the states of all components of the system. Their state determines the power draw of the device. The system collects this information into energy capsules that are attributed to activities such as individual packet transmissions or receptions. Based on the cumulative energy information in the energy capsules, a power profile can be determined.

The Eon system [10] makes energy consumption a first-class abstraction and schedules application flows depending on the current energy profile. The Pixie operating system [11] takes a different approach, in which the programmer articulates the energy requirements each application has and the operating system schedules tasks accordingly. By contrast with systems that leave energy management to the application layer, the integrated concurrency control and energy management architecture [12] does automatic power management in the operating system, without the need for application involvement. This demonstrates that per-component energy management can be efficiently performed by the operating system without application-layer involvement.

(*f*) *Storage*

In sensor networks, secondary storage takes the form of on-board flash ROM or secure digital cards. Storage systems for flash-based storage must deal with the physical storage semantics of flash memory. In flash memory, unlike RAM memory and magnetic disks, bits cannot be freely written: individual bits can only be flipped from 1 to 0. To reset bits from 0 to 1, an entire sector of bits must be erased. A sector typically contains many kilobytes of data. The storage system must be able to efficiently map data onto the sectors to make writing and erasing efficient. To make matters worse, individual sectors have a fixed number of erase cycles before they wear out. The storage system therefore must perform wear-levelling to spread the erasure load evenly across the memory to avoid wearing the memory out.

The traditional approach secondary storage overlays a file system over the storage. With the file system, named files can be created, written to, read from and deleted. The file system approach is general enough to underpin many mechanisms running on top of a file system and the approach has therefore been widely used.

Recognizing the often simple storage needs of early applications, the early work on file systems for sensor networks, such as Matchbox and efficient log-structured flash file system (ELF) [13], used simplified file system models that only supported

append operations and did not allow files to be overwritten. Evolving needs have led more recent systems, such as Coffee [14], to provide a full file system interface that freely supports rewriting and deletion of files.

Other approaches to storage have also been proposed. Amnesiac storage [15] is a technique in which sensor data stored in secondary storage are compressed over time. Recent data are compressed with low loss and, as data get older, the compression ratio is increased at the cost of loss of detail. The intuition behind the system is that, as data get older, the importance of detail decreases. Another model is the sensor-as-database model, which turns the on-board storage into a database, from which records can be retreived with SQL-like queries [16].

(*g*) *Communication software architectures*

Application software running in sensor networks is often communication-bound. The sensor network operating system must make it easy for application programs to efficiently perform its communication tasks. Moreover, the operating system must make the underlying network protocols possible to implement efficiently. Each sensor network operating system provides a software framework in which network protocols can be implemented and efficiently executed. We call this the communication architecture of the operating system, and it performs memory allocation and management for message buffers, manages neighbour and address tables, and provides an interface for applications.

Traditional communication architectures follow a layered design in which different layers of the system solve an individual part of the communication problem. Early work in sensor networks challenged this traditional view, because of the novel resource constraints and application directions of sensor networks, and instead took the direction of rethinking layering and towards cross-layer optimization [17].

The TinyOS system uses a concept called Active Messages, where each message is tagged with an identifier that corresponds to an application at the receiver. When the operating system on the receiving node receives the message, it invokes the application that registered itself with the corresponding identifier. With an extension to the TinyOS system, Polastre *et al.* [18] argued that the narrow waist of the sensor network stack should be placed at the link layer. The authors showed that abstracting the link layer allowed for generality in both neighbour management and neighbour sleep cycles. A later modular network layer [19] added multi-hop functionality to this model, but more recent work has argued moving the narrow waist back to the network layer [20]. The Contiki Rime communication architecture [21] separates the protocol logic from construction and parsing of protocol headers, thereby making it possible to map the protocols across different underlying link layers and protocols, without sacrificing runtime performance.

Recently, the use of the Internet protocol (IP) architecture has become widespread in sensor networks. Contiki has long provided full IP-networking support through the uIP and uIPv6 stacks. Likewise, TinyOS provides IP-networking support through its Berkeley low-power Internet protocol (BLIP) stack. Because both systems are designed around the same underlying IP architecture, they share many of their design elements. This is a natural course of development as the community has progressed in addressing the fundamental problems in sensor network operating systems.

3. Network protocols

Sensor network protocols exist to satisfy the communication needs of diverse applications, including data collection, event detection, target tracking and control, as well as services such as localization and time synchronization. Unlike conventional networks with principally peer-to-peer or data access workloads, sensornets exhibit very different data networking needs. Workloads like collection (many-to-one), dissemination (one-to-many) and bulk transport (many-to-one) account for the majority of application traffic, while point-to-point communications have played a lesser role. Network protocols to enable these services are constrained by the extreme resource scarcity of sensor nodes—including energy, computing, communications and storage—and by the unpredictable dynamics of the wireless mesh networks that the nodes form to deliver their data. These myriad challenges have led to new protocols that are efficient in their energy consumption, careful in their computational needs and miserly in their memory footprints, all while discovering neighbours, forming networks, delivering data, and correcting failures quickly and reliably.

(*a*) *Network elements, organization and architecture*

Wireless sensornets consist of large numbers of resource-constrained nodes which are often embedded in their operating environments, distributed over wide geographical areas, or located in remote regions. The nodes in a sensornet are usually organized as a mesh in which most nodes both originate traffic locally and forward traffic on behalf of others. Figure 2 shows the network elements—*root*, *mesh* and *leaf* nodes—and their organization into a multi-hop wireless mesh network with default routes over which data flow.

(i) *Network elements: root, mesh and leaf nodes*

Root nodes, sometimes called base stations or border routers, connect the sensornet to external networks, like the Internet. Depending upon the network architecture, root nodes could be stateful application gateways that proxy the sensor network to the outside world or they could be traditional Internet gateways that forward packets but do not keep application state. Root nodes are resource rich: they are typically always-on devices that are wall powered, they often contain 32-bit processors with many megabytes of memory and code space, they may have gigabytes or flash storage and they have multiple media interfaces.

Mesh nodes both sense their local environment and forward data for other nodes in the network, but they are highly resource constrained. A typical node might have a 16-bit processor, 10 kb of RAM, 100 kb of ROM and 1 kb of flash memory storage (although some nodes have much greater resources). Because they are either mobile or embedded in their environment, mesh nodes typically operate from limited on-board energy reserves (e.g. AA batteries) or power harvested from ambient sources (e.g. solar). The power constraints of mesh nodes greatly limit the design space of sensornet protocols.

Leaf nodes sense their environment, just like mesh nodes, but they do not forward traffic on behalf of other nodes. Leaf nodes may be very energy constrained—operating from batteries or minuscule levels of energy harvested from the environment—and they may experience power disruptions to the

Figure 2. Sensor network elements and organization. Root nodes (triangles) connect the sensornet to external networks. Mesh nodes (squares) originate their own sensor and other data and forward traffic on behalf of other nodes. Leaf nodes (circles) are like mesh nodes, but they do not typically forward flow-through traffic.

intermittency of ambient sources. Owing to their limited energy and power volatility, these nodes usually interact with wall-powered, always-on root or mesh nodes. Leaf nodes are otherwise similar to mesh nodes in their computing, communications and storage capabilities.

(ii) *Network organization: a multi-hop wireless mesh topology*

Sensor nodes self-organize into a multi-hop mesh network that provides many benefits over a star topology, including reach, reliability and power. In challenging radio frequency (RF) environments (e.g. forests, bridges and buildings), obstructions and reflections cause shadowing and multi-path fading, which can cause communication failures. Mesh networks are able to route around the anomalies that cause shadowing, and reduce the odds of multi-path fading,

by offering multiple forwarding pathways through different peers. This improves the reach and reliability of communications. On the power front, data can be forwarded over multiple, short hops in a mesh network, so each transmission requires less power than a single, longer transmission. Since RF energy attenuates super-linearly with distance, several short transmissions consume less total energy than a single long one.

A mesh architecture sets sensornets apart from most other wireless networks in which nodes are either infrastructure devices or client devices, but usually not both. For example, in most IP networks, nodes are either routers or end hosts. The same is true for cellular networks. Topologically, sensornets are more like mobile ad hoc networks (MANETs) or wireless mesh networks, in which nodes forward data on behalf of other nodes. But the similarities end there. MANET traffic is principally point-to-point between endpoints within the MANET, while wireless mesh networks offer data transport for endpoints located outside the mesh network. Although workload and topological differences have played important roles in the evolution of sensornet protocols, it is the node-level resource constraints, and especially energy, that have had the greatest influence on network architecture.

(iii) *Network architecture: services, interfaces and protocols*

At the highest level, an architecture decomposes a problem domain into a set of *services*, which are functional components, their mechanisms and their responsibilities. Conceptually, network services are layered, with each layer building upon the layers below it and offering services to the layers above it. Typical layers include physical, link, network, transport and application. Services at the link layer include neighbour discovery, link quality estimation, channel contention, and unicast and broadcast transmission. Network layer services include neighbour discovery, configuration, routing, unicast and multicast forwarding, adaptation and header compression. At the transport layer, services include both reliable and unreliable data transfer, congestion avoidance and flow control. At the application layer, the key services include various forms of data representation—naming, locating and encoding—and transfer—items, streams and bulk data. Although layers are conceptually elegant, early work blurred the layer boundaries and often collapsed multiple layers together for a number of reasons but chiefly to economize on and optimize the allocation of scarce resources. More recent work has shown, however, that many of these optimizations can still be retained in a layered architecture.

An architecture can also define a set of *interfaces* to its services, which are the structures and functions with which services expose their mechanisms. The interface between layers is a contract which both exposes the functions and limits the scope of what a layer allows. Sometimes, interfaces expose important data structures that adjacent layers manipulate. A good example is the neighbour table which is shared between the link and network layers: the link layer discovers links (neighbours) and estimates the signal strength and quality of those links but the network layer chooses which set of neighbours to keep in the table and which ones to discard, based on their routing utility. Another example is the message buffer: as data traverse through the layers of the network stack, headers and footers may be prepended, appended or removed at each layer. The interfaces that define how

these operations occur are essential for ensuring that message processing is both computationally fast and memory efficient. One common approach uses buffers with space preallocated for the header and footer data based on how layers are statically composed. This approach avoids repeatedly copying and caching partial message data at each layer at the expense of being able to dynamically resize headers and footers.

Finally, at the lowest level, an architecture can specify its *protocols*, which include packet formats, communication exchanges, and state machines. Protocol specifications, like IEEE 802.15.4 and IP, are essential for interoperability across nodes and networks, respectively, and are especially important in multi-vendor environments. However, much of the early sensornet research focused on services and interfaces with little attention paid to the protocols beyond their implicit as-is specifications in embedded code. Meanwhile, industry proposed a series of monolithic protocol suites, like ZigBee, that were not flexible enough to accommodate or leverage emerging research results. In the past 2 or 3 years, these two approaches have been largely reconciled and today open protocols, like IPv6 routing protocol for low power and lossy networks (RPL), which incorporate key research results, are being standardized.

An architecture's services, interfaces and protocols are not created in a vacuum. Rather, successful architectures adopt foundational principles which reflect the requirements and constraints of the underlying problem space. In the case of sensornets, *the fundamental principle is resource conservation*: energy, computation, storage and all communications must be limited owing to their extreme scarcity. This principle permeates the design of nearly all practical systems and serves as the backdrop for the remainder of this paper.

(*b*) *Fundamental services and their resource-limited realizations*

(i) *Low-power communications*

The radios commonly used in sensor nodes can consume a significant proportion of the system power budget when operated continuously. Two important techniques have emerged to reduce this energy consumption: *sampled* and *scheduled* communications. Although we discuss them separately below, recent academic research and commercial offerings have combined the two approaches, viewing them as offering complementary benefits.

Sampled operation is the simpler technique and works by having the receiver sample the radio channel periodically for inbound traffic and powering down, or duty cycling, the radio between samples. If the receiver detects incoming traffic, it remains awake until it has received the entire packet. The transmitter, in this scheme, extends its transmission window so that it is at least as long as the interval between successive channel samples by the receiver—a technique called low-power listening. The extended transmission could be an extended (but content-free) preamble (B-MAC [22]), repeated back-to-back data packet retransmissions (X-MAC [23]), or the details of the intended recipient and a rendezvous time when the packet will be transmitted (Hui & Culler's MAC [20]). A variation on sampled communications replaces the channel sample (listening) with a channel probe (transmission) and the sender's extended transmission with an extended listen-before-send period—a technique called low-power probing [24]—that forms the basis of receiver-initiated communications [25]. The

power savings from sampled communications depend on the radio start-up time, the channel sample (or probe) time and the sampling period, but, in practice, it is possible to operate radios at 1–2% duty cycles, thereby reducing radio energy consumption by 98–99%.

Scheduled operation coordinates in advance when radios may (or may not) transmit and receive, which allows a radio to be powered down during periods of scheduled inactivity. This allows a node to keep track of its neighbours' on/off schedule and transmit only during those times when a neighbour's radio is turned on. This type of scheduled sleeping, which is really a form of duty cycling, is used in several link layers that synchronize the entire network to a global schedule with a fixed duty cycle (S-MAC [26]) while others adapt the duty cycle by extending (shortening) the radio on-time (off-time) to adapt to increases in data traffic while maintaining a fixed overall period (SCP [27]; T-MAC [28]). Variations on the theme may offset the radio on/off schedules of neighbouring nodes, or nodes along a path, to reduce contention under heavy data traffic, schedule concurrent communications across space, time and spectrum, or forgo global time synchronization and allow nodes to independently and locally select radio on/off schedules. However, with potentially dozens or hundreds of neighbours and limited memory, keeping track of all such independent neighbour schedules scales poorly with node density. Yet, broadcast communications—when a node transmits to all of its neighbours—is a common communications primitive required by many protocols.

Regardless of whether a sampled, scheduled or combined approach to low-power communications is employed, nodes learn of each others' presence through a process called *neighbour discovery*. In its simplest form, a node periodically broadcasts information about itself to make its neighbours aware of its presence and it listens for an extended period of time to receive its neighbours' broadcasts about their presence and particulars. This approach is used by S-MAC, a scheduled protocol, to discover neighbours. In a variation on the approach suitable for sampling protocols like X-MAC, a node instead transmits continuously for an extended time, once in a long while, to let its neighbours know of its presence. Both are instances of a more general, quorum-based, neighbour discovery process [29], although other approaches support discovery with dissimilar duty cycles [30].

(ii) *Data disseminating*

Data dissemination, or simply *dissemination*, is a fundamental problem in sensor networks and it is used in operations as simple as changing a sensor sampling rate to as complex as retasking the entire network with a new application. A variety of techniques have been proposed for data dissemination. The earliest of the proposals, called directed diffusion, advocated local communications in sensor networks [31,32]. In the directed diffusion model, some nodes publish data, some nodes subscribe to data, and intermediate nodes disseminate subscriber interests, establish an interest gradient, and (optionally) transform the data as they flow from sources to sinks.

Directed diffusion played an important role, but early applications did not require the in-network processing or dissemination among arbitrary endpoints that it offered. Rather, simple dissemination services were built using packet

floods. During *flooding*, the source broadcasts a packet with a unique identifier. Each node that receives the broadcast simply rebroadcasts the packet exactly once. Ideally, each set of rebroadcasts reaches a new set of nodes and the flood cascades in concentric rings out from the source. Flooding, in practice, faces several problems including unreliability due to packet collisions (called 'broadcast storms') and many unnecessary packet transmissions, resulting in a higher than necessary energy cost for dissemination.

Another successful approach to data dissemination recasts the problem as distributed consensus and uses 'polite gossip' to quickly and efficiently disseminate small and large data items across a network. The *Trickle* algorithm establishes a density-aware local broadcast with a consistency model that guides when a node communicates. If a node's data do not agree with its neighbours, the node communicates quickly to resolve the inconsistency. But, when neighbours agree, they slow their communication rate exponentially, such that in a stable state nodes send at most a few packets per hour. Instead of flooding a network with packets, the algorithm controls the sending rate so each node hears a small trickle of packets, just enough to stay consistent [33].

Although the Trickle algorithm was first developed to support network reprogramming, it has found many other uses that require consensus, including data collection and loop-free routing. The main reasons are its simplicity, reliability and efficiency: by relying only on local broadcasts, Trickle handles node additions, is robust to network transients, packet loss and node disconnection, and requires just a few bytes of state, making it ideal for resource-constrained sensor nodes.

(iii) *Data collection*

Data collection, or simply *collection*, is an important and well-studied class of protocols since the purpose of most sensornets is to deliver data from a network of sensors to one or more collection points. Most collection protocols build minimum cost routing trees that are rooted at the collection point(s). The goal of collection routing is to minimize the cost of delivering data from the nodes in the network to the collection point(s). The cost metrics for building collection trees have evolved from hop count to more sophisticated measures. The choice of the metric, and the efficiency with which it can be computed or measured, plays an important role in both the cost and performance of collection routing [34].

Much of the evolution in collection protocols has centred on how trees are built, how forwarding costs are computed, and how congestion is detected and avoided. Collection points initiate tree construction by advertising a zero cost using broadcast beacons. Neighbours that receive these beacons add their own forwarding cost to the one reported by the collection point and then they rebroadcast the beacons with the modified cost. Thus, the beacons advertise the cost of the path from a node to the root. Most nodes receive multiple beacons and choose one (or more) neighbours that offer the lowest cost path to the collection point. This process repeats to build out the collection tree until all nodes in the network have joined the collection tree and selected a minimum cost path to the root.

Early collection protocols used a hop count metric, with each node incrementing the hop count by one before retransmitting a beacon. Recognizing the widely varying loss characteristics of the wireless channel, subsequent

protocols estimated the expected number of transmissions per successful delivery and used this figure as the forwarding cost to each neighbour. Still newer protocols included physical layer information including signal strength and bit error rates in the cost computation. Recent protocols combine all of this information and supplement it with actual bi-directional link reliability computed using link layer packet acknowledgement rates. The newest, backpressure-based, protocol dispenses with tree building and instead uses queue gradients to dynamically route packets [35].

The key to scaling collection protocols lies in keeping a small amount of state at each node. A node keeps track of one or a few parents but not its children, which could grow unbounded. A node estimates the link quality and detects link asymmetries to each of its candidate parents without requiring the candidate to maintain any state about the node by using link layer acknowledgments to estimate the reverse path. Paying attention to these details ensures that collection protocols work across a wide range of densities without having nodes run out of memory.

(iv) *Reliable data transfer*

Collection routing's best-effort reliability semantics—owing to its lack of end-to-end acknowledgements that guarantee data delivery from source to sink—are insufficient for applications that require a complete dataset at the stream and block levels [36]. A number of reliable transport services have been developed to address best-effort collection's shortcomings, and offer reliable stream and block data transfer. An overview of the design space, including many theoretical and practical considerations, basic architectural choices, retransmission policies, end-to-end versus hop-by-hop recovery and choice of selective/cumulative or positive/negative acknowledgements, has been explored [37]. The rate-controlled reliable transport protocol improves on reliable data collection by recognizing that congestion plays an important role in data loss [38]. Reliable data transfer highlights the need to transmit end-to-end acknowledgements along a reverse path—from sink to source—which in turn incurs a memory cost and energy overhead to maintain.

(*c*) *Standardized protocols*

Most of the sensornet protocol research of the past decade has occurred outside the purview of standards or standards organizations. Although the most popular link layer for sensornets is based on IEEE 802.15.4, a link and physical layer standard, many of the IEEE-defined services remained unused in research environments. And, while some early sensornet efforts were standards based, it has only been recently that standards bodies like the Internet Engineering Task Forge (IETF) have embraced fully the challenges of running IP and routing IPv6 datagrams on extremely resource-limited sensor nodes.

One challenge with running IPv6 over the current version of IEEE 802.15.4 is that the link layer payload is just 127 bytes, so uncompressed IP headers would leave little room for application payloads. IETF's 6LoWPAN standard addresses this problem by defining encapsulation and header compression mechanisms that allow IPv6 datagrams to be sent and received using 802.15.4 frames, in most cases with just a few header bytes, and at most with 25 header bytes [39].

Recent research has shown that an IP-based architecture is quite compatible with the needs of sensor networks and that the IP's layered architecture allows sensornet protocol stacks to retain many of the optimizations required for efficient networking without compromising the interoperability of a standards-based approach [20]. Now that IPv6 datagrams are routinely carried on 802.15.4 frames to internetwork sensornets, the IETF has begun to standardize the routing [40] and consensus [41] algorithms that lie at the heart of sensornet collection, dissemination and point-to-point routing.

References

1 Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E. & Culler, D. 2003 The nesC language: a holistic approach to networked embedded systems. In *Proc. of the ACM SIGPLAN 2003 Conf. on Programming Language Design and Implementation, San Diego, CA, June 9–11 2003*, pp. 1–11, ACM.

2 Abrach, H., Bhatti, S., Carlson, J., Dai, H., Rose, J., Sheth, A., Shucker, B., Deng, J. & Han, R. 2003 Mantis: system support for multimodal networks of *in-situ* sensors. In *Proc. of the 2nd ACM Int. Conf. on Wireless Sensor Networks and Applications, 2003, San Diego, CA, September 19 2003*, pp. 50–59, ACM.

3 Han, C., Rengaswamy, R. K., Shea, R., Kohler, E. & Srivastava, M. 2005 SOS: a dynamic operating system for sensor networks. In *Proc. of the Int. Conf. on Mobile Systems, Applications, and Services* (*MobiSys*), *Seattle, WA, June 6–8 2005*, ACM.

4 Cao, Q., Abdelzaher, T., Stankovic, J. & He, T. 2008 The LiteOS operating system: towards Unix-like abstractions for wireless sensor networks. In *Proc. of the Int. Conf. on Information Processing in Sensor Networks* (*ACM/IEEE IPSN*), *April 22–24 2008, St. Louis, Missouri*, IEEE Computer Society.

5 Polastre, J., Szewczyk, R. & Culler, D. 2005 Telos: enabling ultra-low power wireless research. In *Proc. of the Int. Conf. on Information Processing in Sensor Networks* (*ACM/IEEE IPSN*), *Los Angeles, CA, April 25–27 2005*, IEEE Press.

6 Dunkels, A., Schmidt, O., Voigt, T. & Ali, M. 2006 Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In *Proc. of the Int. Conf. on Embedded Networked Sensor Systems* (*ACM SenSys*), *Boulder, CO, October 31–November 2 2006*, ACM.

7 Adya, A., Howell, J., Theimer, M., Bolosky, W. J. & Douceur, J. R. 2002 Cooperative task management without manual stack management. In *Proc. of the USENIX Annual Technical Conf., June 10–15 2002, Monterey, CA*, pp. 289–302, USENIX.

8 Klues, K., Liang, C. M., Paek, J., Musaloiu-E, R., Levis, P., Terzis, A. & Govindan, R. 2009 TOSThreads: thread-safe and non-invasive preemption in TinyOS. In *Proc. of the Int. Conf. on Embedded Networked Sensor Systems* (*ACM SenSys*), *November 4–6 2009, Berkeley, CA*, ACM.

9 Fonseca, R., Dutta, P., Levis, P. & Stoica, I. 2008 Quanto: tracking energy in networked embedded systems. In *Proc. of the Symp. on Operating Systems Design and Implementation* (*OSDI*), *December 8–10 2008, San Diego, CA*, pp. 323–338, USENIX.

10 Sorber, J., Kostadinov, A., Garber, M., Brennan, M., Corner, M. & Berger, E. 2007 Eon: a language and runtime system for perpetual systems. In *Proc. of the Int. Conf. on Embedded Networked Sensor Systems* (*ACM SenSys*), *Sydney, Australia, November 6–9 2007*, pp. 161–174, ACM.

11 Lorincz, K., Chen, B., Waterman, J., Werner-Allen, G. & Welsh, M. 2008 Resource aware programming in the pixie OS. In *Proc. of the Int. Conf. on Embedded Networked Sensor Systems* (*ACM SenSys*), *Raleigh, NC, November 5–7 2008*, pp. 211–224, ACM.

12 Klues, K., Handziski, V., Lu, C., Wolisz, A., Culler, D., Gay, D. & Levis, P. 2007 Integrating concurrency control and energy management in device drivers. In *Proc. of the ACM Symp. on Operating System Principles* (*SOSP*), *October 14–17 2007, Stevenson, WA*, pp. 251–264, ACM.

13 Dai, H., Michael, N. & Han, R. 2004 Elf: an efficient log-structured flash file system for micro sensor nodes. In *Proc. of the Int. Conf. on Embedded Networked Sensor Systems* (*ACM SenSys*), *Baltimore, MD, November 3–5 2004*, ACM.

14 Tsiftes, N., Dunkels, A., He, Z. & Voigt, T. 2009 Enabling large-scale storage in sensor networks with the coffee file system. In *Proc. of the Int. Conf. on Information Processing in Sensor Networks* (*ACM/IEEE IPSN*), *San Francisco, CA, April 13–15 2009*, IEEE Computer Society.

15 Nath, S. 2009 Energy efficient sensor data logging with amnesic flash storage. In *Proc. of the Int. Conf. on Information Processing in Sensor Networks* (*ACM/IEEE IPSN*), *April 13–16 2009, San Francisco, CA*, IEEE Computer Society.

16 Tsiftes, N. & Dunkels, A. 2011 A database in every sensor. In *Proc. of the Int. Conf. on Embedded Networked Sensor Systems* (*ACM SenSys*), *November 1–4 2011, Seattle, WA*, ACM.

17 Heidemann, J. S., Silva, F., Intanagonwiwat, C., Govindan, R., Estrin, D. & Ganesan, D. 2001 Building efficient wireless sensor networks with low-level naming. In *Proc. of the ACM Symp. on Operating System Principles* (*SOSP*), *October 21–November 14 2001, Banff, Canada*, pp. 146–159, ACM.

18 Polastre, J., Hui, J., Levis, P., Zhao, J., Culler, D., Shenker, S. & Stoica, I. 2005 A unifying link abstraction for wireless sensor networks. In *Proc. of the Int. Conf. on Embedded Networked Sensor Systems* (*ACM SenSys*), *November 2–4 2005, San Diego, CA*, ACM.

19 Cheng, T. E., Fonseca, R., Kim, S., Moon, D., Tavakoli, A., Culler, D., Shenker, S., & Stoica, I. 2006 A modular network layer for sensornets. In *Proc. of the Symp. on Operating Systems Design and Implementation* (*OSDI*), *November 6–8 2006, Seattle, WA*, USENIX.

20 Hui, J. & Culler, D. 2008 IP is dead, long live IP for wireless sensor networks. In *Proc. of the Int. Conf. on Embedded Networked Sensor Systems* (*ACM SenSys*), *Raleigh, NC, November 5–7 2008*, ACM.

21 Dunkels, A., Österlind, F. & He, Z. 2007 An adaptive communication architecture for wireless sensor networks. In *Proc. of the Int. Conf. on Embedded Networked Sensor Systems* (*ACM SenSys*), *Sydney, Australia, November 6–9 2007*, ACM.

22 Polastre, J., Hill, J. & Culler, D. 2004 Versatile low power media access for wireless sensor networks. In *Proc. of the Int. Conf. on Embedded Networked Sensor Systems* (*ACM SenSys*), *Baltimore, MD*, pp. 95–107. ACM Press.

23 Buettner, M., Yee, G. V., Anderson, E. & Han, R. 2006 X-MAC: a short preamble MAC protocol for duty-cycled wireless sensor networks. In *Proc. of the Int. Conf. on Embedded Networked Sensor Systems* (*ACM SenSys*), *Boulder, CO, October 31–November 3 2006*, pp. 307–320, ACM.

24 Musaloiu-E, R., Liang, C.-J. M. & Terzis, A. 2008 Koala: ultra-low power data retrieval in wireless sensor networks. In *Proc. of the Int. Conf. on Information Processing in Sensor Networks* (*ACM/IEEE IPSN*), *St Louis, MO, April 22–24 2008*, IEEE Computer Society.

25 Dutta, P., Dawson-Haggerty, S., Chen, Y., Liang, C. & Terzis, A. 2010 Design and evaluation of a versatile and efficient receiver-initiated link layer for low-power wireless. In *Proc. of the Int. Conf. on Embedded Networked Sensor Systems* (*ACM SenSys*), *Zurich, Switzerland, November 3–5 2011*, ACM.

26 Ye, W., Heidemann, J. & Estrin, D. 2002 An energy-efficient MAC protocol for wireless sensor networks. In *Proc. of the IEEE Conf. on Computer Communications* (*INFOCOM*), *New York, NY, June 23–27 2002*, IEEE.

27 Ye, W., Silva, F. & Heidemann, J. 2006 Ultra-low duty cycle MAC with scheduled channel polling. In *Proc. of the Int. Conf. on Embedded Networked Sensor Systems* (*ACM SenSys*), *Boulder, CO*, pp. 321–334, ACM Press.

28 van Dam, T. & Langendoen, K. 2003 An adaptive energy-efficient MAC protocol for wireless sensor networks. In *Proc. of the Int. Conf. on Embedded Networked Sensor Systems* (*ACM SenSys*), *Los Angeles, CA, November 5–7 2003*, ACM.

29 Tseng, Y.-C., Hsu, C.-S. & Hsieh, T.-Y. 2002 Power-saving protocols for IEEE 802.11-based multi-hop ad hoc networks. In *INFOCOM'02: Proc. of the 21st Annual Joint Conf. of the IEEE Computer and Communications Societies, June 23–27 2002, New York, NY*, IEEE.

30 Dutta, P. & Culler, D. 2008 Practical asynchronous neighbor discovery and rendezvous for mobile sensing applications. In *Proc. of the Int. Conf. on Embedded Networked Sensor Systems* (*ACM SenSys*), *Raleigh, NC*.

31 Heidemann, J., Estrin, D., Govindan, R. & Kumar, S. 1999 Next century challenges: scalable coordination in sensor networks. In *Proc. of the 5th Annu. ACM/IEEE Int. Conf. on Mobile Computing and Networking, Seattle, WA, August 15–20 1999*, pp. 263–270, ACM.

32 Intanagonwiwat, C., Govindan, R. & Estrin, D. 2000 Directed diffusion: a scalable and robust communication paradigm for sensor networks. In *Proc. of the Int. Conf. on Mobile Computing and Networking* (*ACM MobiCom*), *August 6–11 2000, Boston, MA*, pp. 56–67, ACM.

33 Levis, P., Brewer, E., Culler, D., Gay, D., Madden, S., Patel, N., Polastre, J., Shenker, S., Szewczyk, R. & Woo, A. 2008 The emergence of a networking primitive in wireless sensor networks. In *Communications of the ACM*, **51**, ACM.

34 Woo, A., Tong, T. & Culler, D. 2003 Taming the underlying challenges of reliable multihop routing in sensor networks. In *Proc. of the Int. Conf. on Embedded Networked Sensor Systems* (*ACM SenSys*), *Los Angeles, CA*, pp. 14–27, ACM Press.

35 Moeller, S., Sridharan, A., Krishnamachari, B. & Gnawali, O. 2010 Routing without routes: the backpressure collection protocol. In *Proc. of the Int. Conf. on Information Processing in Sensor Networks* (*ACM/IEEE IPSN*), *April 12–16 2010, Stockholm, Sweden*, ACM.

36 Welsh, M. 2010 Sensor networks for the sciences. *Commun. ACM* **53**, 36–39. (doi:10.1145/1839676.1839690)

37 Stann, F. & Heidemann, J. 2003 RMST: reliable data transport in sensor networks. In *Proc. of the 1st Int. Workshop on Sensor Net Protocols and Applications, Anchorage, AK*, pp. 102–112. IEEE.

38 Paek, J. & Govindan, R. 2010 RCRT: rate-controlled reliable transport protocol for wireless sensor networks. *ACM Trans. Sen. Netw.* **7**, 20:1–20:45. (doi:10.1145/1807048.1807049)

39 Montenegro, G., Kushalnagar, N., Hui, J. & Culler, D. 2007 Transmission of IPv6 Packets over IEEE 802.15.4 Networks. Internet proposed standard RFC 4944.

40 Winter, T. & Thubert, P. (eds) and RPL Author Team. 2011 RPL: IPv6 Routing Protocol for Low Power and Lossy networks. Internet Draft draft-ietf-roll-rpl-11.

41 Levis, P., Clausen, T., Hui, J., Gnawali, O. & Ko, J. 2011 The Trickle Algorithm. Internet proposed standard RFC6206.