Designing new Architectures and Protocols for Wireless Sensor Networks: A Perspective

Mukundan Venkataraman, Kartik Muralidharan and Puneet Gupta,

Software Engineering and Technology Labs
Infosys Technologies Ltd.
Electronics City, Hosur Road
Bangalore 560100. India.

Email: {mukundan_v, kartik_muralidharan, puneet_gupta}@infosys.com

Abstract—Inspiring work in the areas of Wireless Sensor Networks (WSNs) is being achieved, with Mobile Ad hoc Networks (MANETs) clearly a key ingredient to this. Primary constraints to system design in WSNs are bandwidth, energy, memory and processing. One pattern very evident from the very incipient stages of research in this area is the dominance of Internet design being transported to these environments. Though the Internet architecture is extremely appropriate for its environment, we argue that transporting it to WSN could only prove disastrous. What is to be learnt from the Internet is more a philosophy of building successful stacks. To that end we investigate “Stack Aware Architectures” and advocate its inclusion in stack designs. We also validate this concept by analyzing the Internet architecture for this property. Another pattern very evident in WSN research is that individual modules are validated over a set of workloads at least thought to be representative of the entire application domain. The class of applications and their requirements in WSNs, however, are varied. We go on further to analyze classes of applications that use WSNs, their individual requirements from the layers beneath, and a basic example to a dynamic and adaptive stack for WSN, suited for all application types (based on a per-packet mapping of functions required for it), which of course is also stack aware.

I. INTRODUCTION

Advancements in mobile communications has enabled us to interconnect a wide range of devices. With diminishing hardware costs and the ability to interconnect disparate devices, the promise of truly pervasive computing is becoming more and more a fact to be coped with in time.

Pervasive computing environments are envisioned to create a system that is ubiquitously embedded in the environment, completely connected, effortlessly portable, and constantly available. This will be an environment where there will be a collaboration of humans, wireline and wireless technologies. These environments extend mobility and connectivity into the realms of autonomous, self-collaborative and distributed set of devices. While researchers have portrayed different visions to their version of pervasive computing environments, there are a few basic requirements rooted deeply in the concept, like (i) Invisibility: At a minimal level, an end user is only aware of the devices that he interacts with. The software overlay is totally submerged; (ii) Proactive: Newer perspectives impose a degree of intelligence on the software that the devices carry, that can both anticipate the user needs and guide him towards it; (iii) Autonomous: the set of participating devices should be autonomous/self-governing and self-collaborative in nature. They should require little intervention, and more importantly, should not rely too much on supporting infrastructure; (iv) Ubiquitous: The system should be easily deployable and available anywhere; (v) Robust: The system should be resistant to failures, and should further be resilient; (vi) Local Resources: This is an environment where devices become ”smart”, and are aware of the context in which they operate. In other words, they are aware of their ”neighboring” devices and milieu at large. The stress is hence on local resources, probably one or two hops away from the user.

A wealth of sample scenarios and examples of such environments have been discussed and analyzed in literature (see, for example, [1], [2] and [3]). The examples taken up in literature are inspiring, and authors have also gone on to list challenges that one is confronted with when bridging this gap between vision and reality. While a pervasive computing environment in general refers to a specialized set-up, an active area of research is the wireless sensor network (WSN) with an ad hoc network of devices actively capturing and disseminating data.

Challenges in system design have been analyzed on different axes, like human-computer interactions, system software, suitable hardware for these scenarios, protocols, energy efficiency, wireless interference, networking support and so on. In this paper, we deal with Wireless Sensor Networks (WSNs) and ad hoc networks, the important role they play, the challenges they pose from a pervasive computing perspective and the constraints they possess.

A. What are WSNs and MANETs?

Sensor networks are a class of wireless networks whose primary motive is environment sensing. They could consist of nodes with sensing and communication devices embedded in them. On the hardware side, these nodes could be small devices (size of ”motes” [15] designed by University of California, Berkeley are comparable to a quarter). They can form the basis for many ”smart” environments, like hospitals, conference rooms, smart homes or battlefields. Primary operation in WSNs could be sensing the environment and reporting
B. Motivations for this work

Motivations for this paper are twofold:

1) The Internet Influence: The Internet architecture and its component protocols are admirably suited to their purpose, and have witnessed immense popularity and growth in usage population for the past three decades. But its very success and stability tends to inspire almost all of networking research, where most of research is an attempt to migrate the same successful Internet design while patching up parts to fit the new scenario. This mode of operation only proves disastrous, if not worse, especially in scenarios like WSNs where requirements are completely different.

2) Module Validation: Individual modules designed for WSNs are often validated over a set of workloads that are at least thought to be representative of the entire application gamut of WSNs. This is not a very valid assumption, since the application types and their requirements are varied. Applications range from point-to-point (P2P) to broadcast to anycast. Some applications demand reliable communication while others do not, and some have real time constraints (like low latency) while others do not. Hence, it is not very appropriate to assume that a given module performing over a particular workload or tuned to do well for one application can be generally applied everywhere. On the other hand, it is almost impossible to come up with a module that handles all of the application types successfully.

Our contributions in this paper are as follows: (i) We identify aspects that a successful stack must posses, and advocate their inclusion in future designs rather than protocol migration from the Internet. We call this concept a "stack aware" architecture; (ii) We validate the Internet design for "stack aware" properties; (iii) We develop a taxonomy of application types for WSNs and list functions that each application type demands; (iv) We develop a basic blueprint of a dynamic stack that maps requirements for an application to available modules on a per packet basis based on preamble processing. This work could serve as a basis for future stack design for WSNs.

The rest of the paper is organized as follows: Section II deals with stacks in general, introduces the concept of Stack Aware architectures and concludes with a case study of the Internet Architecture. Section III discusses problems with WSNs in depth, Section IV presents an application taxonomy and a basic blueprint of a dynamic stack. We conclude the paper in Section V.

II. COMMUNICATION STACKS

Our understanding of computer networks has greatly evolved ever since the first archived experiment of trying to send data from one computer to another. We have successfully designed a system as complicated and as magnificent as the Internet. Much of this success is attributed to one key achievement: formulation of a communication stack.

A. What is a Stack?

The basic communication process consists of sending data from source to destination. Primarily, it is the case of two applications wanting to communicate with each other. Hence, application at source generates information, which is encoded and transmitted to destination, and the destination application decodes the same. This entire process is logically partitioned into a definite sequence of events or actions, and individual entities then form "layers" of a communication "stack".

Traditionally, the process of communication was formally organized as the ISO-OSI reference model stack. This stack consists of seven layers, Application to Physical.

B. "Stack Aware" Architectures

Developing network software required clear cut division of labor and an unambiguous function specification at each layer. The OSI reference model served some of these requirements, but it lacked an implementation experience and was not completely grounded to reality. Many protocols sprang up using this as a "reference" model (a name which is retained till today) making modifications to the specifications to suit the case at hand.

It is very valid to have a modular and layered architecture for stacks in general, for all the goodness that a replaceable...
architecture brings. There is, however, one small deciding factor that really goes on to make a stack efficient, practicable, and popular. Since every layer either invokes or provides functionalities to another layer, it is not merely enough to map a sequential array of functions to be performed onto layers to make a successful stack. Care has to be taken to ensure that a function in a layer actually leverages benefits of using another layer, or in other words, idiosyncrasies of one layer should not prove harmful for another layer preceding or following it. As an example, assume a sample stack with layers A, B, C, and D, in that order, from application to physical with A interacting with the application and D having the physical interface ingrained in it. The typical set of functionalities required for data exchange are somehow distributed over these four layers. Assume that this stack is to be used in a context where bandwidth is scarce. This imposes a constraint: to try making control messages and all other non-data packets (pure overheads) a minimum. To that effect, we introduce a module in layer "C" to mitigate these packets generated otherwise (assume an arbitrary mechanism that can do this). Now if layer "D" is such that it generates excessive non-data packets, the stack is almost useless (see Fig. 1. which depicts this).

We propose the following four guidelines for calling a stack architecture "stack aware":

1) Application Demands: Application driven research has been the foundation of excellent science contributions from the Computer Science community. Computer communication is really about two applications talking to each other. Hence, protocols are merely there to facilitate this, i.e., protocols and modules are primarily designed with the application requirements as the central theme.

2) Adaptable and room for modification: Change is inevitable, most certainly so in computer networks. If a protocol works in a particular environment, research will push it further to try performing it in newer environments. As our requirements change, we will evolve continuously. Room for modification is hence a positive thing. While optimizing design to suit the environment is a good thing, it should not be hopelessly rooted to that environment alone. A basic property of any stack is its modular (and replaceable) design. At any point of time, we should have the space to reconsider our design, and/or add/replace modules as per our needs.

3) Layers leverage benefits of using one another: The crux of the concept: layers should be so designed that they appear to be "designed for each other". Lower layers should be aware of the predominant nature of applications using them. Functions in layers should ideally not overlap, and more importantly not interfere. Layers should be aware of what other layers expect from them, and the environment in which they operate (like wireless networks, optical fibre etc.).

4) Implementation Experience: Implementation is perhaps the biggest eye opener. Once implemented, the exercise gives a very precise picture of our standing and reveals caveats in the design. In other words, good stack architectures are not purely a theoretical exercise. The stack is to be designed with the application demands and the medium constraints in mind. In other words, the basis for evaluation is a stack with some implementation rather than abstract theory alone.

C. Internet Architecture: A Case Study

It is a well known fact that the transmission control protocol (TCP) and Internet Protocol (IP) largely account for the success and stability of the Internet. It is also interesting to note that TCP is the most dominant protocol in operation today, despite the OSI reference model preceding it. We take the Internet architecture as an example to analyze a stack. We will not delve into intricate details of the stack, references [11] and [10] are an excellent source of information.

At a basic level, the Internet stack (shown in Fig. 2) consists of the following layers: Application, Socket, Transport, Network, Ethernet and Physical. The following aspects of Internet architecture make it "stack aware":

- Application demands: When work on the Internet started, the stress was to transfer data reliably and in-order. This trend was a valid foresight, since bulk transfers, FTP, e-mail and HTTP accounted for (and still do) almost all of the traffic on the Internet. These require reliable, in-order and error free delivery. Latency is not such a driving point, reliability is. These applications are "elastic" in nature. The protocols underneath do remarkably well to carry out this requirement: TCP provides reliability (acknowledgement), flow control, congestion control and error free transmissions. IP provides addressing, fragmentation and reassembly. The Internet stack is hence very well designed for the applications it caters to.

- Constant adaption and High Optimizations: One very striking feature of the stack is that it is highly aware of the physical transmission media and its intricacies.
For example, initially there were modules for extensive error checking and correction. But with the introduction of optical media and with copper wires becoming less and less prone to errors, such functionalities were largely relaxed. For this same reason, packets drop are blindly assumed to be because of network congestion. In effect, it is highly optimized for the environment that it works in, making it that much more efficient, while allowing room for modifications in the future. While it is optimized for its context, it is also highly flexible. This is due to TCP’s philosophy of end system intelligence, which pushes intelligence away from the center to the end nodes. This allows newer applications and functions to be seamlessly deployed.

- Layers compliment each other: There is a very high degree of harmony among the layers in the stack. As stated, TCP accounts for reliable in-order data flow. TCP is connection oriented while its layer beneath (IP) is connectionless. This aspect of IP gives the stack the goodness of connection oriented and connectionless approaches. Packets using IP are routed independently, and this takes care of load balancing and growing congestion in the network. The IP Datagram is fundamental to the Internet Protocol. A datagram has enough information in it to carry it from source to destination; there is no advance set-up required within the network to do this. The network makes a best effort to send the packet across, best effort meaning that if something goes wrong, the network does nothing, it tried its best. This form of unreliable service is the most basic building block of the Internet. A note may be made on the highly cited ability of IP to "run over anything". Most of the technologies over which IP runs today did not exist when IP was invented, and no protocol has proved to be too bizarre for IP; it has even been claimed that IP can run over a network that consists of two tin cans and a string! [20]. IP also provisions packet fragmentation when gateway routers have packet size restrictions. TCP is end to end, and hence takes care of most of end system processing. IP is hop-by-hop, and hence takes care of hop to hop packet processing. Functions provided in either protocol do not interfere or overlap with each other. In others words, the stack is so designed that every layer is as if designed for the other layers present.

- Implementation Experience: The stack as it stands today has evolved with time and experience. As the size of the Internet grew, newer problems (like congestion, WAN routing, minor problems with previous modules etc.) started emerging. In short, its present state is an answer to all the problems that we have encountered and solved thus far. The stack is still an active area of research.

These aspects particularly highlight Internet’s stack aware property. Other factors also contributed largely to its success, most notably it being open source and freely available to the developing community.

### III. Wireless Sensor Networks

The MANETs and their environments apart, WSNs bring in their own flavors of challenges. We seek to understand what they require before approaching the topic of stack development for them. Notice that with ad hoc networks as the assumed component, we have to connect to our requirements of WSNs while acknowledging challenges of both MANETs and pervasive computing.

A typical environment will consist of a dense collection of wireless and heterogeneous nodes collecting and disseminating data. This consists of intense device-device interactions and human-device interactions. From an ad hoc networking perspective, the set-up could be short range (of the order of two meters maximum), and the devices would be small in volume, less in mass and very economic to procure (ideally just about $1).

We discuss problems and issues in individual layers of WSNs, as we also bring out the ineffectiveness of migrating the Internet architecture as we do so.

#### A. Applications

Application types are varied, and are not as consistent as that of the Internet. Though varying application classes should not be a real problem (for example, even the Internet transports applications varying from highly point-to-point (P2P) to broadcast, multicast or anycast), it is the scarcity of resources that acutely effects performance in WSNs. For example, we might have a WSN setup that provisions for link layer acknowledgement. This might be beneficial for P2P applications where the intent is to, say, pass control information to a base station. When a node in this set up tries to perform a broadcast or anycast, however, acknowledgements prove useless and end up consuming bandwidth, energy, processing and memory; all of which are scarce.

#### B. Routing and Addressing: Issues of Transport and Network Protocols

This area has received a lot of attention among researchers, both for ad hoc networks and now for WSNs. Ad hoc research presented two classes of protocols: proactive, which compute routes in priori and try keeping tables consistent (eg Destination Sequenced Distance Vector routing, or DSDV [14]); and reactive, which initiate a request-reply dialogue to discover routes in an on demand basis (eg Dynamic Source Routing, or DSR [8]). Both these approaches, and in general routing solutions developed thus far for ad hoc networks are simply inappropriate for WSNs.

The very first packet routing protocol for ad hoc networks was based on the DARPA packet radio protocols [19], which allowed for a network of 50 packet radio nodes with some degree of mobility to communicate.

Attempt to migrate the Internet design and patch it up for ad hoc networks has been particularly strong in case of routing in the more recent past: for example, DSDV was an extension of the Internet Distance Vector (DV) routing, and patched up sufficiently to fit the ad hoc bill. Realizing...
ineffectiveness of proactive approaches, Ad hoc On Demand Distance Vector (AODV) [7] was introduced, which computed routes on demand rather than proactively. It still was based on Distance Vector routing, though the divergence from Internet was a little greater. Dynamic Source Routing (DSR) [8] was then introduced, it used source routing (the entire route in the preamble) and in a way was a combination of DSDV and AODV.

A bit of exciting work has also been underway with protocols specifically meant for WSNs: SPEED [16], Geographic Forwarding (GF) [17], Implicit Geographic Forwarding (IGF) [18] etc. to name a few. SPEED is a stateless protocol tuned for real time communication, IGF is more on robustness and is also stateless, while GF uses a greedy forwarding mechanism. These protocols perform much better than conventional ad hoc protocols largely because they address issues of WSNs better and are not migrated and patched up.

We discuss issues with WSNs as under:

- **IP Based Addressing:** Internet architecture is based on IP addressing, one needs an IP address to start a session. Subsequently, most of routing has worked on trying to create networks with IP addresses work when nodes are mobile. The tendency is to replicate P2P connections based on IP addressing, much like the Internet. WSNs require location information more than actual ID or address of a node. For example, "report temperature in region X" or "get images from cameras of region Y" or tracking applications are not dependent upon a nodes ID at all. Location awareness is more a central theme.

Multicast, Broadcast and Anycast: Much of present work tries to achieve multicast, broadcast and anycast by extending the P2P concept with a group of IP addresses. The problem is exacerbated if reliability is additionally provided. These particular class of applications are again more concerned with region and location rather than specific ID’s as such.

- **Broadcast nature:** Transmission antennae can be highly point to point or broadcast in nature. Prevalent communication is however broadcast, such that every node in the transmitting nodes radius can potentially receive that packet. Since the medium is so naturally conducive to broadcast mode, a key advantage would be to utilize this property rather than work around it by replicating point-to-point connections.

- **Stateless Protocols Score:** A "state" is loosely the knowledge of other nodes or the network at large. Routing table computation, whether proactive or reactive, requires resources, and is advisable for address based specific P2P routing. Maintaining a state is expensive, valuable resource goes into it: node movement causing exceptions and subsequent updates, control messages, and active memory to store them on every node, apart from energy drain.

Node movement in WSNs is a norm rather than an exception (unlike the Internet), so expecting it is wiser than working around it. Stateless protocols who at a bare minimum maintain only neighbor lists (based on beacons) naturally score over.

- **Real Time Constraints:** Monitoring is a prime application type for WSNs, and being real time in nature, it is sensitive to latency. Solutions like "on-demand" routing are just not appropriate: (i) they take time to compute and set-up routes and (ii) there is a problem of premature drops (there is a Link Layer (LL) queue that connects application to protocol. If route resolution is underway, and the application pumps packets and the queue is full, packets are prematurely dropped [32]).

- **Congestion and Load Balancing:** This arena has only recently started receiving attention. Internet routing can safely assume TCP to take care of this issue. WSN stack would be rather micro in nature due to memory constraints, and we do not have space for something as complex as TCP to work over a routing layer, and neither is all of TCP very relevant. This aspect must hence be integrated at about this layer. Load balancing is important since it is wiser to be able to have multiple paths from sources to destinations, and evenly spreading traffic would be nice, especially when talking about carrying real time streams [16]. Though routing protocols may find a path that is less congested at route acquisition (like reactive routing), doing so on a per packet basis is wiser since the ultimate goal is to dynamically balance load.

- **Client Server models:** The Internet architecture is based on a client-server model which has one central server with a pre-configured IP address that passively listens for connection requests. Some examples of collaborative environments [21] like a collaborative many-to-many chat session for example would send messages to the server, which in turn broadcasts it to participants. Such designs cripple efficiency is resource usage. Once again, replicating Internet design and extending its architecture to forcibly fit the bill is clearly not wise. An interesting solution of a many-to-many protocol (M2MP) and many-to-many invocation (M2MI) is presented in [21].

- **Gray Areas:** Woo [22] and Zhao [23] independently studied volatility in link qualities. Studies indicate the presence of "gray areas", where some nodes receive more that 90% packets while neighboring nodes report only 50% reception. Studies also show that these "gray areas" are rather large: about one third of the total communication range. Link quality estimation is hence crucial, since it allows us to make better estimate about a path to the next possible node. This can either be done by beacons or by snooping into network traffic [22].

### C. Medium Access and Energy Efficiency

The dominant Medium Access Control (MAC) standard in WSNs and ad hoc networks in general has been the IEEE 802.11 (particularly the Distributed Co-ordination Function, or DCF) [31]. The standard itself is an active area of research. Energy efficiency as a module is necessary, because energy
is scarce. Further, the objective is to put as many nodes as possible in the "sleep" state, where energy conservation is at its minimum. Studies show that a node in the "idle" state when it is not sending or receiving data consumes energy comparable to a node which is doing so (data indicates ratios of idle:receiving:transmitting as $1 : 1.05 : 1.4$ [24] and $1 : 2 : 2.5$ [25]). The only significant means of conserving energy is to put as many nodes as possible in the sleep state, as well as try to dissipate energy uniformly from all participating nodes. This is done in a wake to increase network lifetime.

There have been attempts to combine MAC and routing (eg IGF [18]) or to combine MAC and energy efficiency. These three components are highly inter-related: energy efficiency used determines number of active nodes, which in turn affects routing protocols, as well as MAC since channel contention is made by active nodes. There is further the issue of deciding who remains awake and who goes to sleep.

Issues in these layers are as under:

- **Reliability:** Some applications, like habitat monitoring, require reliability. It is wise to allow link-layer acknowledgement to provision for reliable communication.
- **Gray Areas:** Gray areas have an effect on MAC as well. Using link strength estimation is hence a key to success here as well.
- **Adaptable functions:** Most of MAC research is highly tuned for one particular application, and the module is often validated with that one class. Studies have shown the diversity in application types: traffic patterns are erratic with low and high workloads, and generic MAC protocols are sensitive to this. An interesting case is that of B-MAC [26] that uses a small core channel access function and allows applications to a certain degree a level of freedom to use specific functions.
- **Preamble sampling:** When nodes send out data (typically broadcast), nodes in the receiving range would have to sample through the preamble to determine the course of their action with it. This has been an active area of pursuit, with researchers arguing for long or short preambles. References [15], [27], [28] provide more on this direction for MAC solutions.
- **Switching States:** Switching between sleep to awake is a crucial phase, and care must be taken to not produce excessive control messages. Most protocols broadcast a variety of control messages to notify neighbors of their transitions.
- **Uniform Energy Dissipation:** Network lifetime is the primary target for energy conservation techniques. This means no single node must end up dissipating too much energy in seclusion, and when possible, neighboring nodes should cyclically take up the load to ensure a uniform (at least an "almost" uniform) energy dissipation model. As is obvious, this part is easier stated than achieved.

**D. Other Issues**

Apart from the aforesaid issues in transport, network, medium access, application and energy efficiency, there are issues that are generally applicable to WSNs in general, like:

- **Wireless Medium:** Wireless mode of data transport brings in its own flavors of problems, like lossy links, high bit-error rates, issues of security (medium is broadcast).
- **Creating robust and reliable systems:** We envision WSNs in future to be made up of small devices which can be conveniently embedded in environments, or even thrown about here and there. These nodes would form a network, and sustain themselves. With diminishing hardware costs, one can even imagine them to cost lesser than a standard battery. In the sense, one can throw away a mote rather than recharge its battery. Hence, these motes would hardly be attended to. This brings reliable and sustained operations into focus.
- **Security, Confidentiality and Authentication:** These issues are persistent in all of networking. The issue here is constrained processing and memory, which might affect complex implementations.
- **Micro nature of stack:** Environment monitoring is a prime application. Nodes in WSNs are constrained for processing and memory, and the internetworking problem is nowhere close to the complexity of the Internet. At a bare minimum, it is sufficient to be able to form connected graphs with strong links with neighboring nodes. Further, nodes receiving packets for forwarding must do this "on-the-fly". The concept of "store-process-forward" is not recommended because: memory and processor are in demand, and some applications are sensitive to this additionally induced latency.
- **Scalability:** Needless to say, the network should scale to a large number.

**IV. DYNAMIC STACK**

To cope with the above problems, and also acknowledging memory, processor and energy limitations, we propose a dynamic stack. Our architecture would consists of a series of micro functions pre-compiled. Using guidelines of "stack aware" architectures, we pay attention to application demands first.

We add a byte of preamble to the application payload. The bits in this byte are purported to precisely describe application demands. The stack is "dynamic" because based on preamble processing, it invokes specific functions from the available pool. Hence, an application wanting reliability gets it, and the ones not wanting it will not. This section provides a detailed description of such an architecture.

Much of related work concentrates on individual modules, which are in turn validated by a set of workloads for which that work is tuned. To support a larger or almost all of the application type is a primary motivation of this work. Integrating available modules to a larger harmonious whole has received little attention as well. We further advocate fresh
designs that are earmarked to fit the WSN arena, rather than extensions of the Internet design.

In our quest for a dynamic stack, we note the following:

- Resources are scarce. Conserving them and using them efficiently is a prime motive. This includes all of memory, processing, energy and bandwidth. On the same lines, a bulky stack with too many optional functions is also not very wise, since it eats up too much memory.
- Applications are varied.
- Challenge is to fit application demands to precisely to the required functions. This means doing away with general purpose solutions.
- Make decisions at the source and intermediate nodes on a "per-packet" basis. This makes the architecture adaptable to changing conditions.

A. Preamble

The one byte preamble added to the application payload is as shown in Fig. 3. The preamble consists of the following fields:

1) Acknowledgment Flag: Enabling this facilitates use of Link Layer acknowledgements to be used. Note that this does not entail a retransmission. This would only notify the upper layer of the ACK packet, depending upon the choice of solicitRetx() used. Default value for this is decided with the application type and data packet in question.
2) Link Estimate Flag: This would be used by application where loss is not tolerated. A good example might be a control packet. Enabling this ensures a strong link as a metric for the next candidate node, possibly even a trade off between a short path for it.
3) Data Type Flag: Packets are broadly classified as control information (which takes a higher priority) and data packets. The sensitivity of a data packet is further estimated on the application type.
4) Retransmission Flag: This flag indicates want of a retransmission upon failure of packet delivery. This requires Ack-Flag to be set.
5) Application Type: This is a two bit mandatory field, along with data-type flag, can help define default parameter values for other fields. Applications types are classified as point-to-point, multicast, broadcast and anycast.
6) Real Time Flag: This bit is also mandatory. Real time data is sent on a greater priority (and at par with control messages), compared to normal data. Real Time Data is also liable to undergo load balancing procedures, unlike control messages.
7) Load Balance Flag: When this flag is set, a forwarding node takes into consideration load balancing as a criteria in the next candidate hop selection. Packets may be routed to a node that is less loaded with a possible trade-off for a shortest path.

To be setting the fields, we allow a set of API’s to the application programmer. This offers flexibility in designing a wide variety of application types with specific tailor made requirements from the networks. An entire host of decisions can be taken based on the fields set.

B. Stateless Protocol

The protocol should be stateless. It does not maintain the states of nodes outside its transmission range. This boils down to a list of neighbors only, with an added metric of link estimation. The beacons used could facilitate this. Beacons additionally can provide estimations of congestion at that node also.

When a packet is to be sent, nodes use geography information rather than IP addresses. Note that SPEED [16] achieves a lot of these (it is stateless, uses back-pressure for congestion control and load balance, and uses geography information for routing), and these make it an apt fit as a good choice for a routing protocol. The drawback with SPEED is that it is highly optimized for real time streaming applications, and would be unable to optimize performance for other metrics, like a packet requiring reliability for example. The protocol, however, can be revised to handle these.

C. Energy Efficiency and MAC

There are many modules available for energy efficiency. SPAN [5] and GAF [13] notably for ad hoc networks, and MAC has the popular 802.11 DCF in power saving mode (PSM) [31] and more recently SMAC [29], B-MAC [26] and IGF [18] specifically for WSNs.

SPAN and GAF use a lot of control messages, especially when making transitions from sleep to awake and vice versa (IGF scores over these on these grounds). Grid sizes in GAF are static and nodes require geography information. SPAN uses a variety of control messages. IGF uses both geography information and has lesser number of control messages. With a metric of conserving bandwidth and energy, IGF would make a better choice.

SMAC implementations are a lot more bulkier than B-MAC, and SMAC deals with lot of synchronization which makes it a little complex [26]. BMAC is an interesting architecture, since it allows nodes freedom to select use of acknowledgement and in its core uses a small and easy to implement channel access method.
// Packet Type
#define BEACON 1
#define ACK 2
#define CONTROL 3
#define DATA 4

// Application
#define P2P 1
#define BCAST 2
#define MCAST 3
#define ACAST 4

Fig. 4. Macro Usage

interface ApplicationInterfaceToStack {

// Acknowledgement
command result_t EnableAck();
command result_t DisableAck();

// Link Estimation Decision
command result_t UseLinkEstimate();
command result_t DisableLinkEstimate();

// Packet Type
command result_t DefinePacketType(int);

// Retx
command result_t SolicitRetx(int flag);

// Application type
command result_t DefineApplicationType(int);

// Real Time Nature
command result_t IsRealTime(int flag);

// Load Balance
command result_t BalanceLoad();
}

Fig. 5. Interface used by application to Stack

Also note that acknowledgements, if used, should be provided at the Link Layer. This has a specific advantage: it allows for link-layer re-transmissions (when necessary) which does not bother higher layers.

D. API Descriptions

The set of API’s required by applications is shown in Fig. 5 and the required macros to be used are as shown in Fig. 4. We use a notation similar to that of TinyOS [30] implementations to describe our concept.

Going by the bits in the preamble, we have a corresponding set of API’s exported to the application:

1) EnableAck(), DisableAck(): This would set or unset link layer acknowledgement for packets delivered. Using this only informs the application layer of packet delivery status. The application is free to take action upon its discretion, like a retransmission.

2) UseEstimateLink(int flag): This would be used by application where loss is not tolerated. A good example might be a control packet. Enabling this ensures a strong link as a metric for the next candidate node, possibly even a trade off between a short path for it.

3) DefinePacketType(int flag): Used to define the type of packet generated, broadly a “data” packet or a “control” packet. We have loosely classified all non-data packets to be of control type by making the field in the preamble a flag type. This is done with a rationale that control packet is more sensitive to loss than a data packet.

4) DefineApplicationType(int): Used to define application as point-to-point, multicast, broadcast or anycast.

5) IsRealTime(int flag): This is further used to describe the nature of the application, in being real-time (latency sensitive but loss tolerated) or not real time (latency tolerated, but not loss). Real time data is sent on a greater priority (and at par with control messages), compared to normal data. Real Time Data is also not liable to undergo load balancing procedures, unlike control messages.

6) BalanceLoad(int flag): This bit lets dynamic load balancing to take place when a packet is at a node. When a packet is to be forwarded, depending upon the selections it has made, a suitable next candidate node is decided. Upon invocation of this function, a trade-off to a shorter path may be made to balance network traffic and curb growing congestion.

7) SolicitRetx(int flag): Most of the application types, broadly, do not require retransmissions. It is wiser to make provisions to send sensitive data on a strong channel than to send it on a pre-defined path and tolerate loss by retransmitting. Use of this API enables link-layer retransmissions.

E. General Guidelines for API Usage

The general set of API’s described is a loose definition of a possible implementation. Providing API’s allows application programmers to accurately describe their application demands and leave details of implementation to the lower layers with an assurance that resources are used in the most efficient fashion.

We note a few rules that follow as an inherent inference to our classification of application types and demands in general:

1) A SolicitRetx() call is undefined when DisableAck() is called. Logically, one cannot make a provision for a retransmission if there are no acknowledgements.

2) Control messages, real time data and anycast messages are by default not provided an acknowledgement. Using the first clause, this would mean SolicitRetx() is also not provided. Control messages are of a higher priority and it makes sense to send them on the strongest link possible to ensure they reach intended destinations(s). In case of real time data, loss of a frame/packet or two is usually tolerable, but latency is not. As an example, for a real time streaming application sending frames captured,
a delayed or retransmitted frame is as good as a lost one. Likewise, an anycast packet has no specific destination in mind, and it makes sense to transmit it one time on a strong link.

3) Real Time and Broadcast Data packets are not given link estimates.

Note that it is possible for an application programmer to write programs that make inefficient use of resources. For example, a real time application using acknowledgements, no load balancing and allowing retransmission of frames is possible to write with the given API’s. The design hence depends upon the judiciousness of the programmer also. When unsure, one can proceed with only the required definitions (like type of data, application type and IsRealTime(flag)), and let the stack decide the best combination of activities to be performed.

F. Taxonomy of Applications

We develop a taxonomy of all allowed application types and probe deeper into getting a basic understanding of what they want from the layers beneath.

Applications are classified on the parameters of the type of application they are (p2p, broadcast, anycast or multicast), the type of packets they carry (data or control) and whether they are real time or not.

Such a taxonomy is as shown in Fig 6. The "Defaults" column lists the default action the stack takes when specific parameters are not asked for by the application code. Note that an explicit API call should naturally override the defaults, since the motive is to give flexibility and freedom to the application programmer. The compilation is only a possible "default" setting.

We rate control packets as sensitive to loss. They are not provided acknowledgements (something like a beacon or an ACK packet itself would be a control packet). We instead make sure they are sent on the best possible link out of the given links to candidate nodes. Load balancing is disabled for p2p, but enabled for others.

P2P data packets are provided with reliability (ACK’s and retransmissions), are sent on strong links and are prone to be distributed for load balancing as well. Providing reliability here is more sensible compared to other applications like broadcast or a multicast since traffic is highly directional. The same is not true, however, for real time p2p data, where the sheer number of packets that may be transported might prove disastrous for a reliable communication.

We follow a general trend for real time streams, wherein we disable reliable communication, disable link estimations and disable load balancing. Load balancing is disabled because the data is sensitive to latency. We cannot have out of order packets, with varying delay times (a delayed packet is as good as a lost one).

Broadcast, multicast and acncast data further are not provided reliability. We enable link estimate for anycast data, never so for broadcast data.

<table>
<thead>
<tr>
<th>DataType</th>
<th>Application</th>
<th>RealTime</th>
<th>Defaults</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control</td>
<td>P2P</td>
<td>0</td>
<td>DisableAck(), EstimateLink(true), BalanceLoad(true)</td>
</tr>
<tr>
<td>Control</td>
<td>All Others</td>
<td>0</td>
<td>DisableAck(), EstimateLink(true), BalanceLoad(true)</td>
</tr>
<tr>
<td>Data</td>
<td>P2P</td>
<td>0</td>
<td>EnableAck(), SolicitRetx(), EstimateLink(true), BalanceLoad(true)</td>
</tr>
<tr>
<td>Data</td>
<td>P2P</td>
<td>1</td>
<td>DisableAck(), EstimateLink(true), BalanceLoad(true)</td>
</tr>
<tr>
<td>Data</td>
<td>Multicast</td>
<td>0</td>
<td>DisableAck(), EstimateLink(true), BalanceLoad(true)</td>
</tr>
<tr>
<td>Data</td>
<td>Broadcast</td>
<td>0</td>
<td>DisableAck(), EstimateLink(false), BalanceLoad(false)</td>
</tr>
<tr>
<td>Data</td>
<td>Anycast</td>
<td>0</td>
<td>DisableAck(), EstimateLink(true), BalanceLoad(true)</td>
</tr>
<tr>
<td>Data</td>
<td>Multicast</td>
<td>1</td>
<td>DisableAck(), EstimateLink(false), BalanceLoad(false)</td>
</tr>
<tr>
<td>Data</td>
<td>Broadcast</td>
<td>1</td>
<td>DisableAck(), EstimateLink(false), BalanceLoad(false)</td>
</tr>
<tr>
<td>Data</td>
<td>Anycast</td>
<td>1</td>
<td>DisableAck(), EstimateLink(true), BalanceLoad(true)</td>
</tr>
</tbody>
</table>

G. Analysis of Dynamic Stack

This work has not concentrated more on individual layers, the theme has been on making a harmonious stack rather than deal with advancements in every layer. From the preceding sections, it should be clear that such an architecture makes an efficient use of resources.

We would imagine an integration to be of the following type: a stateless routing protocol; a minimal neighbor information estimating link strength, congestion, constructed using beacons or acknowledgements; link layer acknowledgement and a retransmission mechanism; a medium access mechanism and finally interface to the physical medium.

The set of API’s effect the routing protocol as well as the link layer. The choice of next candidate routing node would be based on suitability (in terms of a node being a “candidate” node if it can take the packet to destination). A candidate node would then be put to a test of link estimate and load balancing tests (if they are enabled) in that order. At the end of the entire process, the appropriate node is selected and the packet forwarded.
A note may be added on finding the "candidate node". This may be found using "direction", using addresses or any suitable mechanism. The end result would be a set of nodes who can "potentially" take the packet forward towards the destination.

An example stack based on our basic set of API's is as shown in Fig 7. The diagram only shows the path of a packet from application to physical. Actions upon receiving a packet at a destination or at an intermediate hop would be similar in nature with suitable changes to the calling functions. The packet stops at the routing and the link-layer. At the routing layer, once a pool of candidate nodes are identified, one can perform link estimate (of a higher priority) and then a load balance test may be run, based on settings in the preamble. The set of processes feeds into the Packet* SendTo(node*) function which would prepare the packet to be sent the selected node.

The packet at Link Layer takes another decision. If ACK is demanded, the bits are appropriately set. Alternately, if a bi-directional session is already running between two nodes, one can use this packet to piggyback pending ACK’s for the destination. If retransmission is solicited, the packet is additionally buffered.

We then have modules for energy efficiency and medium access. Protocols may be used from the choice of available ones present or ones to come.

The question remains as to how "stack aware" such an architecture would really be. We run our guidelines to investigate:

- Application demands: We have almost earmarked this design with applications in mind. The entire architecture is to ensure applications get the best and most suitable service they require from the stack.
- Adaptability: Modular nature ensures adaptability since functions (or layers) can be conveniently added or replaced.
- Layers leverage benefits of using one another: This is an important issue, since we have to make sure that protocols selected ensure this property. At one level, when applications make appropriate function calls, they already ensure much of this property to a good degree. Care must, however, be taken to ensure that the set of protocols selected must also have this property.
- Implementation Experience: Tested protocols already in the community make good choices. Implementations of them working together may be an issue. Work is also underway on our part to design this stack over TinyOS implementations.

H. Related Research

There exist ample research on individual modules to perform tasks for WSNs. Little work, however, focuses on coming up with integrated architectures and issues dealing with them. We discuss two such projects in this section: The TinyOS [30] and the Hermes project [33]. Both are radical new designs: TinyOS deals with a basic framework as a basis for future designs and Hermes integrates modules to a harmonious whole. We are not aware of any other significant work focusing on these issues.

TinyOS is an open source (source at [34]) event driven operating system for wireless embedded sensor networks. The architecture is based on a component model, which can be improved/replaced while keeping the implementation light following severe memory constrains for mote like architectures. Its event-driven execution model enables fine-grained power management yet allows the scheduling flexibility made necessary by the unpredictable nature of wireless communication and physical world interfaces [35]. This work was mainly designed with Berkeley motes in mind, and forms an excellent starting point as a basis of much of WSN research.

The Hermes project at the University of Virginia, Charlottesville, is a bottom up approach to building a stack. The work aims to build a "scalable, robust and time-energy aware network architecture for sensor networks" [33]. Hermes creates a set of relevant modules for every layer of the stack, earmarking functions specifically for WSNs, and finally integrates modules to form a stack. Contributions, apart from integration, include: energy efficiency, realiable and robust MAC, localization service, application independent data aggregation and more. The interested reader is encouraged to read more about modules and the Hermes project at large [33]

V. Conclusion

This paper presents a new perspective to stack development with the philosophy of stack aware architectures, validates this property for the Internet, advocates fresh designs for WSN
Acknowledgment

The authors would like to thank Dr. Anirban Chakrabarti (of SETLabs, Infosys) for his comments and suggestions, which improved the overall presentation of this paper.

References