Supporting Proactive Application Event Notification to Improve Sensor Network Performance*

Christophe J. Merlin and Wendi B. Heinzelman

Department of Electrical and Computer Engineering,
University of Rochester, Rochester NY
{merlin,wheinzel}@ece.rochester.edu

Abstract. As wireless sensor networks gain in popularity, many deployments are posing new challenges due to their diverse topologies and resource constraints. Previous work has shown the advantage of adapting protocols based on current network conditions (e.g., link status, neighbor status), in order to provide the best service in data transport. Protocols can similarly benefit from adaptation based on current application conditions. In particular, if proactively informed of the status of active queries in the network, protocols can adjust their behavior accordingly. In this paper, we propose a novel approach to provide such proactive application event notification to all interested protocols in the stack. Specifically, we use the existing interfaces and event signaling structure provided by the X-Lisa (Cross-layer Information Sharing Architecture) protocol architecture, augmenting this architecture with a Middleware Interpreter for managing application queries and performing event notification. Using this approach, we observe gains in Quality of Service of up to 40% in packet delivery ratios and a 75% decrease in packet delivery delay for the tested scenario.

Keywords: Wireless Sensor Networks, Middleware, Architectures.

1 Introduction

Wireless Sensor Networks (WSNs) have received much attention from the research community, and many advances have improved their behavior. However, WSNs provide many great challenges because of their resource constraints. Software solutions must operate on a variety of platforms and deployments while providing continuous Quality of Service (QoS) to the end user. In general, QoS should not exceed the level required by the application, as this usually results in depleting network resources faster. Fulfilling this requirement may be further complicated by the inherently dynamic topology of the network, whether sensor nodes are mobile, and whether their energy sources can be replenished.

* This work was supported in part by NSF #CNS-0448046.

J. Zheng et al. (Eds.): ADHOCNETS 2009, LNICST 28, pp. 3–18, 2010.

To support an application, protocols may take advantage of network information, such as knowledge of the resources or locations of the individual nodes. We argued in past work [1] that all protocols could benefit from such additional information, but that this wealth of new information requires proper channels for dissemination to a node's protocols and neighbors. For example, ad-hoc violations of an OSI model-based architecture could lead to a "spaghetti design" cautioned against by Kawadia et al. [2]. On the other hand, allowing information-sharing among protocols, as we proposed in X-Lisa (Cross-Layer Information Sharing Architecture) [1], safeguards against this risk while still providing the required information to all protocols in the stack. In particular, X-Lisa shares data repositories among all layers in the stack from the node activation layer to the Data Link / MAC layer through a common interface called *CLOI* (Cross-Layer Optimization Interface). However, X-Lisa does not provide any interface between the application and the protocols in the stack—this is typically the role of middleware.

In [3], Römer et al. define middleware through its purpose "to support the development, maintenance, deployment, and execution of sensing-based applications." To achieve this goal, middleware should necessarily abstract the network mechanisms and heterogeneity. In this paper, we introduce ideas for information-sharing architectures to support middleware. We also introduce the counterpart of middleware support: protocols in the stack can benefit from information that an application event has occurred. Thus, while the immediate focus of existing middleware techniques is resource, data, or code management horizontally among nodes (*e.g.,* interactions between middleware of two neighbors), we present the idea of vertical integration of middleware (inside a node itself).

We start with the idea that protocols in the stack may benefit from application-related information. For instance, consider a network in which an a fire is detected and will likely cause node failures. The routing protocol should likely circumvent the event in order to reach the data sink. With this in mind, we propose a *Middleware Interpreter* that logs complex queries from the application. Thus the Middleware Interpreter can determine that an event has occurred. If so, the Middleware Interpreter notifies all subscribing protocols and (if chosen so by the middleware) the node's neighbors, thus shifting the burden of event detection, notification and management away from middleware, which can then revert to its core tasks. These new features entail tight cooperation between the Middleware Interpreter and the neighbor information, as may be stored in a neighbor table, such as that provided in X-Lisa.

In the following, Section 2 outlines desirable goals for WSN architectures, and Section 3 introduces related architectures and middleware designs. Because existing solutions do not respond to the goals we set, we introduce our solution, X-Lisa, and we summarize the features provided by X-Lisa in Section 4. Section 5 describes the Middleware Interpreter, a new X-Lisa component for middleware support. Evaluation of the Middleware Interpreter is given in Section 6, and Section 7 concludes this paper.

2 Goals and Challenges of WSN Architectures and Middleware Support

In previous work [1], we identified some of the key goals for WSN architectures:

- Flexibility through modularity, universality accommodation to all platforms, event notifications and service support.
- Information freshness for an up-to-date vision of the local network.
- Low overhead for energy efficiency.
- Simplicity for quick adoption.

To help support these four goals further, we argue that WSN architectures should support middleware. Since the purpose of wireless sensor networks is to serve an application, information about the application is every bit as important to the functioning of the network as local or global network information. Support for middleware, which maps application requirements to the behavior of the network, appears as a key feature of WSN architectures. Application events such as the detection of a fire or the injection of a new query in the network often spur a change in direction for the protocols: respectively, the search of a newer and safer route, or the activation of more nodes.

However, few current middleware solutions share their information with the rest of the protocol stack. This forces middleware to contact protocols individually or to post information that has to be periodically read by various protocols (mobilizing compute resources frequently). Both solutions introduce more violations to layered models (a hindrance to modularity). On the other side of this issue, protocols that do not receive notifications from middleware are left in a reactive position, waiting for events to be brought to their attention (through a specific packet type to send or receive, periodic enquiry of data, explicit signaling from another protocol, etc.) before they can act. We argue that a proactive stance would increase the QoS, which we will show in Section 6.

Among resource management middleware [4], only MiLAN [5] suggests direct interactions with protocols in the stack, although in a very ad-hoc manner. This illustrates the need to facilitate the various middleware tasks, which Römer identifies [3] as formulating and dispatching complex sensing tasks and reporting related results. We propose to shift the burden of interacting with the protocol stack away from middleware. While middleware must still supply information understandable by the network, it is how this information is communicated to the stack that is the focus of this work. For instance, while middleware maps application requirements to a set of query conditions, automatic notification of application events by the architecture can simplify the interactions between middleware and the protocols. This allows middleware to concentrate on tasks such as aggregation and other network abstractions.

3 Related Work

This section provides a short overview of related work on cross-layer protocol architectures and middleware. We describe in more detail our past work, X-Lisa, in Section 4.

3.1 On Existing Architectures

In [6], Srivastava et al. provide a definition of cross-layer designs and a survey of existing cross-layer models. The authors define cross-layer interactions as back-and-forth information flows, merging of adjacent layers, design coupling without a common interface, and vertical calibration across layers. They also list implementations for cross-layer interactions: explicit interfaces between different layers, shared databases, and heap organization, which provides completely new abstractions (no protocol layers).

CLASS [7], MobileMan [8], CrossTalk [9], SNA [10] [11] and Chameleon / RIME [12] are some of the most prominent architectures for WSNs and Mobile Ad-hoc Networks (MANETs). They offer original solutions to integrate the protocol stack in a modular manner or to abstract parts of it.

3.2 On Existing Middleware

Wang et al. provide a survey of middleware for WSNs in [4]. They analyze middleware projects in terms of *programming abstractions* (how the programmer views the system), *system services* (support for application deployment, execution, and network management), *runtime support* (management and redistribution of resources when the node cannot provide them), and *QoS mechanism* (interaction between the application and the network infrastructure, usually in the form of cross-layer components).

Among existing middleware for QoS mechanism, Wang et al. cite MiLAN [5], whose role is to map application requirements to the nodes' sensing capabilities so as to provide the exact QoS required by the application. MiLAN allows the network protocols and the application to communicate via various graphs that specify what variables are needed under different states of the monitored environment and what abilities each sensor node has in providing QoS. As was shown in [13], this information can be used by the node activation and routing protocols to increase the time for which an application can be supported. However, interactions between MiLAN and lower protocols is done in an ad-hoc manner, which is not favorable to modular designs. The work presented here proposes an elegant solution to this problem.

Other middleware techniques also manage resources proactively: *e.g.,* DSware [14], MagnetOS [15], AutoSeC [16], etc. DSware does not require or allude to vertical interactions with the protocol stack. MagnetOS, although a Java-based operating system, carries middleware tasks to redistribute application components within the network. Even if direct interaction with the protocol stack is not supported, this redistribution has a direct impact on network protocols due to the shifting of communication endpoints. Conversely, AutoSeC specifically interacts with the protocol stack. It manages network resources by collecting data from various protocols and services. However, this relation is not bidirectional since middleware decisions are not explicitly fed back to the protocol stack.

Most closely related to this work is Impala [17]. Liu et al. proposed a middleware system, articulated around a new architecture that specifically focuses

on middleware. Their design goals mirror many of our own (modularity, ease of updates, energy efficiency, etc.). Much of Liu et al.'s work focuses on replacing pieces of code on the fly. Impala also lets the application dynamically adapt to improve QoS and energy efficiency through a dedicated interface. Packet-, data- and device-related events are provided to the application through this interface, which filters and dispatches these events to the relevant application. Our model differs in that a similar interface extends to all layers of the stack, and that the flow of information is bidirectional. Moreover, data filtering is done by checking whether an application query has fired before it is dispatched (a new sensor value will not necessarily create a new event if it does not match the conditions of a query).

4 X-Lisa, an Architecture for Cross-Layer Information Sharing

Following the goals outlined in Section 2, we argued that architectures relying on a non-abstracted shared database are both simple and flexible. Thus, we proposed X-Lisa [1], a new Cross-Layer Information-Sharing Architecture that combines simplicity with support for cross-layer interactions, services, information propagation and event notification. X-Lisa shares information between layers that are not necessarily adjacent, and provides atomic access to the information. It also facilitates modularity, a key aspect of network maintenance such as protocol upgrade or replacement.

The X-Lisa architecture retains a layered structure such that each layer is matched to a communication function in order to maintain a practical and simple design. While fused layer design is still possible with X-Lisa, it is not favored as it hinders modularity.

The Cross-Layer Optimization Interface (*CLOI*) provided by X-Lisa offers indirect access to a repository of information that may be needed by one or more protocols. *CLOI* maintains this information through three structures, a neighbor table, a sink table and a message pool, and it supports *services* that will fill these data structures either once or continuously, depending on the information. X-Lisa also supports event notifications to ease coordination between various layers in the stack.

CLOI is available to all layers and services in the stack and is in charge of automatically propagating information to other protocols and neighbors. It has no authority to make protocol decisions such as route selection, node activation or medium access.

4.1 Information Sharing Structures

Information about a node and its neighbors is kept in a neighbor table, a message pool and a sink table. The neighbor table is particularly dynamic thanks to a Key-Length-Value solution.

4.2 Event Signaling

X-Lisa already provides various types of event notifications through the same parameterized interface **CloiEvents**: it relays events generated in the protocol stack, and also adds network administration events such as a new packet in the pool, a new neighbor, full neighbor table, etc.

Protocols may subscribe to any type of notification, although they must do so at compile time because of the absence of dynamic wiring in TinyOS[1]. However, this requirement is not a limitation as protocols (for instance, MAC and routing) are designed to improve the network behavior given information of a known type (link quality, routing costs, etc.) prior to deployment.

4.3 Information Exchange

In order to keep the information contained in the neighbor table current, X-Lisa provides an automatic update service. The information exchange is carried by an *information vector* that updates the neighbors of a node. The information included in the vector can be curtailed to the needs of the protocols only, preventing unnecessary overhead.

4.4 Important Services

X-Lisa offers and coordinates services important to the correct behavior of protocols. Peripheral services, such as a location service or remaining energy measurement, supply some of the information needed to fill *CLOI*'s information repositories.

5 Middleware Support

5.1 General Ideas

Keeping in mind the goals set for a WSN architecture, we add middleware support. This section does not provide new middleware strategies, but sets to improve communication between existing middleware and other protocols, based on the idea that the protocols, too, could benefit from application information.

The guiding idea of this work is that protocols should be notified when an application event happens so that they may adjust their behavior to new application and network conditions in a proactive manner. Event signaling is the method of choice so that protocols need not constantly check whether an event has occurred.

In effect, the basic implementation principles are to keep an information-sharing structure with common data repositories storing neighbor, message, and query information. The queries must be stored in a way that is understandable by all the protocols. Since updates of monitored fields are not always meaningful to all the protocols in the stack, only a subset of the protocols can subscribe to

[1] Another operating systems, MeshC [18], allows dynamic wiring.

information changes. These updates, including those marking the occurrence of an application event, should be signaled through a simple and common interface (in the case of X-Lisa, *CLOI*). To the best of the architecture's capabilities, new information should benefit all protocols and services in a node so as to guarantee the most up-to-date view of the local network and the application. As a new query event fires, subscribing protocols are automatically notified so they may anticipate the new conditions in the network.

5.2 Integration into an Information-Sharing Architecture

Motivation and Modifications. In deployments for which the middleware layer is absent from the protocol stack, the application acts as the source of data packets in the network, reacting to a stimulus, or simply following a schedule, usually fixed before compile time. This imposes limitations on the complexity of the task that can be performed. For example, consider the difference between requesting the sensor reading at a specific node (as can be done when middleware is absent) and asking for locations where the sensor reading exceeds a certain value (which requires middleware to manage). The latter case is more of value to data-centric networks such as WSNs and can benefit from a middleware layer.

Middleware was introduced to map complex application requirements to an abstracted network. These requirements can be expressed by semantically rich queries, whether in SQL-style syntax [19] [14], or not [20]. We assume that the data sink is either able to map end user requests into a query that can be sent to and interpreted by the nodes in the network, or that the data sink receives this query already formatted for the network from another party. The query is propagated throughout the network, and it is interpreted and stored by the nodes' *Middleware Solution* (the logical layer in charge of query dissemination, interpretation, aggregation, etc.) and passed up to the application.

For example, the Middleware Solution can receive commands from the end user asking for data reports if a certain sensor reading is below or above a threshold. The Middleware Solution makes the decision to record the query depending

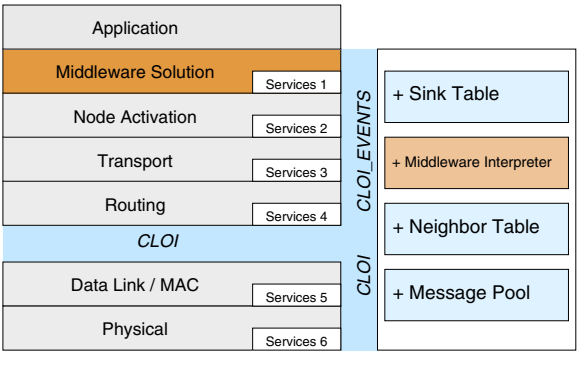

Fig. 1. New X-Lisa architecture with middleware support

on local conditions and parameters in the network. Madden et al. [19] give a good description of possible fields in such a query.

However, if the query information is stored within the Middleware Solution only, it may not benefit all layers in the stack. Therefore, we propose a *Middleware Interpreter*, or *MI*. This Middleware Interpreter's main function is to store queries in a form commonly understood by all the protocols. It also makes sense to store queries (that express an interest in a field such as sensor data) closely tied to the structure that manages these fields, the neighbor table. This can be easily accomplished using X-Lisa. A diagram of the updated X-Lisa architecture is shown in Figure 1.

Query Structure. A query stored in the *MI* is named by a number that helps identify queries uniquely in the protocol stack and throughout the network. A query ID identifies a particular semantic meaning, for instance "high stress level" (when a patient's heart rate and blood pressure are both high). A query is also associated with a field of interest, a date to live ("dtl", the time at which the query expires), and an epoch (the time separating two data reports about the monitored fields). These query fields have to be understood by all the protocols in the stack (this is a limitation of this work, although these fields are common in many middleware solutions).

We also added several additional fields to the stored queries in the *MI*:

- A `status` field to signal whether an event is happening ("FIRED") or not ("IDLE"),
- A `publish` field to request that X-Lisa automatically notify its direct neighbors that a query has fired, thus easing collaboration between nodes,
- A `composite` field, because queries may be aggregate (*e.g.,*"send data reports every t_{epoch} s from locations in the network where the temperature exceeds θ and the infrared measurement exceeds φ"),
- A `conditions` fields indicating values under and above which a query event has happened, although this could be replaced by a function wherever simple "higher" or "lower" conditions are not sufficient, and
- Additional memory space (the size of which must be indicated when the query is first added) that can be allocated for the needs of the Middleware Solution or other protocols (*e.g.,* for data caching, node scheduling, event confidence determinations). This is indicated as the field "extra" in Table 1.

Table 1 illustrates how these fields are stored within the *MI*.

Table 1. Some of the fields of the Middleware Interpreter with their TinyOS primitive type and an example

ID	publish	field	dtl	epoch	conditions	composite	status	extra
int8	int8	uint8	tos_time	uint16	uint32	int16	uint8	int8*
6	0	TEMP	0	5000	> 5	-1	IDLE	NULL

Entering a query into the Middleware Interpreter is accomplished by calls to two functions. The first, to

`query = call Cloi.getQueryBuffer(int queryID, int additionalSize)`,

which specifies the ID type of the query and the extra memory that must be allocated for the needs of the middleware or protocols, and that returns a pointer to the query. If the ID is already present in the *MI*, the returned buffer points to the address of the already registered query, and information may be overwritten. All fields with the exception of dtl and status, must be filled in the query before the query can be entered into the Middleware Interpreter. Invoking

`Cloi.addQuery(MiddlewareQuery *query)`

finishes the insertion of the query inside the *MI*. Access to a query is provided by read, add and remove functions, for instance

`command MiddlewareQuery* Cloi.readInterest(int8_t queryID)`

5.3 Composite Query Registration and Deregistration

The Middleware Solution must break complex queries into simple subqueries. Composite queries are entered with the composite field set to their query ID and by specifying the number of subqueries (this tells the Middleware Interpreter that it is part of a composite query). The Middleware Solution must enter subqueries immediately after with their `subQueries` field set to the ID of the composite query. The subqueries may be reused from already existing queries, and they can be part of other complex queries. A composite query is considered to have fired when all subqueries have fired (logical "AND" of the individual subqueries). The information is stored as shown in Table 2. In this example, the composite query with ID 6 denotes a fire (a high temperature, and a particular IR signature). It is composed of two subqueries of ID 1 (temperature) and 2 (IR light).

Table 2. Composite Query Stored within the Middleware Interpreter

	Field	...	ID	Composite	subQueries	Status
Comp.	−1	...	6	**6**	[**1 2**]	IDLE
Sub.	TEMP.	...	1	**0**	[**6 5**]	IDLE
Sub.	IR	...	2	**0**	**6**	IDLE
Comp.	−1	...	5	**5**	[**1 3**]	IDLE
Sub.	PRES.	...	3	**0**	**5**	IDLE

The inverse operation of query deregistration is invoked with the following call:

`call Cloi.removeQuery(int queryID)`.

For composite queries, the *MI* automatically searches for and removes all subqueries that are no longer used by any existing composite query.

5.4 Interest Registration and Deregistration

Protocols may register an interest in fields that are relevant to their behavior. An "interest" can be seen as a stripped down query that fires every time the field value is updated in the neighbor table.

The Middleware Interpreter can be tightly coupled to the neighbor table to follow what fields are of interest to the protocols in the stack. Interest in different types of queries is kept at the Middleware Interpreter, while other field (data or network related) interests are managed by the neighbor table. Upon updating a field in the neighbor table, *CLOI* checks whether it is of interest to any protocol. A positive answer results in signaling the change to subscribing protocols (if the value is different from the one previously stored). This procedure allows protocols to dynamically register for event notification, preventing unwanted events from interrupting the code when they are not needed.

5.5 Query Notification

A similar behavior is at work for queries. The burden of detecting that a query event has happened is now with the Middleware Interpreter, leaving the burden of data aggregation or query dissemination to the Middleware Solution. As a field is updated in the neighbor table (which includes information about the node itself, including sensed data values), the *MI* receives a notification from the neighbor table because it is a subscriber to all node field changes. The *MI* matches the new field value to conditions expressed by registered queries and determines whether a query has fired. If so, the *MI* signals the event to all subscribing protocols, including necessarily the Middleware Solution and possibly other protocols down the stack. The notification identifies the query with its ID number and returns a pointer to the query in the Middleware Interpreter query table. This process is illustrated by Figure 2.

For composite queries, the *MI* looks through its entries and searches for the status of all its subqueries (which are identified in the `subQueries` field). If

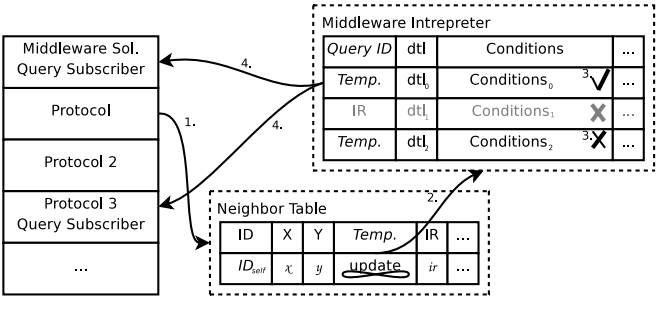

Fig. 2. Event filtering and notification process: 1. A service or protocol updates the neighbor table. 2. If the field is of interest, and the new value is different from the old one, it is submitted to the *MI*. 3. The *MI* checks conditions realizing a query. 4. The *MI* notifies subscribing protocols and services that the query has fired.

Table 3. Query Notifications and Status Based on the Preexisting Status of a Query and Whether Its Conditions Are Met Upon an Update in the Neighbor Table

Conditions / *Status*	X	√
IDLE	—	Notify and set to FIRED
FIRED	Notify and set to IDLE	—

all have a *FIRED* status, the `composite` field of any subquery points to the aggregate query that will serve as the basis for the event notification. In case the composite query must be published to direct neighbors, the notification contains the updated values of all the fields related to the composite query, as well as the ID number of the query.

The Middleware Solution does not need to establish ad-hoc interfaces with other protocols to signal that a query has fired. In fact, it does not even need to use the *protocol event* signaling described in Section 4.2 as this is carried on by the *MI* automatically through the process described directly above. The notified protocols do not preoccupy themselves with checking whether an event is occurring, but they simply wait for event notification and then perform the appropriate actions when they are notified that a query has fired. For example, notification that a query has happened may prompt a routing protocol to refresh a route, a node activation protocol to wake-up neighbors, etc. When the query conditions are no longer met, a notification that the event has gone *IDLE* is sent and the protocols can similarly take action.

Table 3 summarizes the behavior of the *MI* based on the preexisting status of a query and whether its conditions are met.

6 Evaluation of Middleware Support

Although we cannot easily evaluate the convenience of this new middleware component in X-Lisa, it follows our goals of modularity, data freshness, and simplicity—the Middleware Interpreter is accessible by simply wiring to the existing **Cloi** and **CloiEvents** interfaces in TinyOS. However, we set out to evaluate what gains in QoS can be obtained through proactive query status notification to the protocols.

Before considering the gains, we note here the additional memory cost for providing proactive query status notification to the protocols. Using the Tmote Sky motes, the overhead is 7.4 KB in the EEPROM and 100 B in the RAM.

6.1 Health Monitoring Test Scenario

Let us consider a health monitoring application served by a sensor network where nodes are attached with heart rate and blood pressure sensors. As samples about the state of the patient are gathered, the application will accept varying quality of service depending on the aggregate stress level detected by the sensors. The fixed tree network topology is represented by Figure 3, where node 5 is monitoring

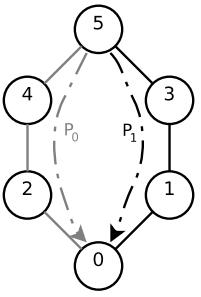

Fig. 3. Topology of the test network

the person of interest to the end user (all other nodes may act as routers). We assume that the node activation protocol keeps nodes 0 through 5 always active, and that the routing protocol has two routes from the source node 5 to the data sink 0: paths P_0, {5 - 4 - 2 - 0} and P_1 {5 - 3 - 1 - 0}. P_0 passes through nodes with higher residual energy, but suffers from higher packet error rates than P_1. Under high stress levels, packet delivery should be reliably sent to the data sink. The MAC protocol uses a Low-Power-Listening (*LPL*) scheme [21]: nodes sleep and periodically wake-up every t_i s to listen for incoming transmissions. Packets must therefore be sent with very long preambles (at least t_i s) during which the destination will wake-up at least once. Most *LPL* MAC protocols, although very energy efficient, cause significant delays of $^{t_i}/_2$ s on average. In order to reduce delivery latency for urgent packets, we implement a *LPL* MAC protocol that may reduce its t_i value by a factor of four when certain conditions in the network and application are met.

The application requirements are summarized in Table 4. In our simulation, the patient's blood pressure increases from normal to high rapidly, and the heart rate readings cycle from low, to medium, to high, and again. Consequently, the source node will send packets with types 0, 2 and 4 (see Table 4). Although this succession of sensor readings is not very likely in real deployments, it will help gauge the responsiveness of the network.

The simulation starts with the base station sending a query to spread its interest in detecting events related to the patient's health (source node 5). The routing and *LPL* MAC protocols subscribe to these queries (their type must be meaningful to all concerned layers). Every time a query state changes, notification to the routing protocol ensues, and the routing protocol selects either path P_0 or P_1 (for the case *without* the *MI*, the routing protocol is left randomly choosing between P_0 and P_1 because it does not benefit from advanced query notifications). When a query event fires, notification is sent to the *LPL* MAC protocol, which increases or decreases its duty cycle by a factor of four depending on the current situation.

We simulated ten runs of this scenario for $1,800$ s in TOSSIM (the TinyOS simulator) and compared them to the protocol stack *without* X-Lisa or the Middleware Interpreter. While surely the application could be equally well served with ad-hoc middleware / other protocols interactions, that architecture would

Table 4. Application requirements for the Middleware Interpreter only scenario

$BloodPressure \rightarrow$ / $HeartRate \downarrow$	Normal	High
Low	30 *s* epoch Delivery Failure OK Long Delivery Delay OK Packet type 0	
Medium	30 *s* epoch Delivery Failure OK Long Delay OK Packet type 1	15 *s* epoch No Delivery Failure Long Delay OK Packet type 2
High	15 *s* epoch No Delivery Failure Long Delay OK Packet type 3	5 *s* epoch No Delivery Failure Short Delay Packet type 4

be exceedingly complex and inappropriate for modular designs. Thus, for fairness purposes, the *MI* solution was only compared to a layered scheme where protocols share a neighbor table among themselves, although it does not support event and query notification.

For each packet type (or patient state), we measure the per-hop delivery delays, the "output delay" (as the difference between a change in the patient's aggregate stress level and the time a report is sent), and total delivery delay (the time at which the data sink receives a packet whose type corresponds to the new patient state). Packet delivery ratios are also of interest and give a good indication of the possible QoS improvement brought by the *MI*.

6.2 Simulation Results

Delays. Figure 4(a) shows the various delays experienced by different packet types. The per-hop delivery delay (top graph) illustrates the behavior of the MAC layer: for packet types 0 and 2, it is very similar for the case with and without the *MI*. However, type 4 packets enjoy a much reduced per-hop packet delay (at least 50% shorter). This is consistent with the requirements set by the application in Table 4.

With the *MI*, after a change in the state of the patient's health (middle graph), a packet is sent by the source after 15 *s* for type 0 (no particular emergency), and only a few milliseconds in other cases. Without the *MI*, corresponding delays vary: this is because in a certain state, the application checks for the sensor readings periodically depending on the reporting epoch for that state. In state 0, the application checks the sensor reading every 30 *s*, causing the highest packet output delay when going from state 0 to 2. In state 2, the application reads sensor outputs every 15 *s*, which explains the delay when transitioning from 2 to 4.

The total packet delivery delay after a change in the patient's state is shown in Figure 4(a), bottom graph. For the case when the patient is in a relaxed state

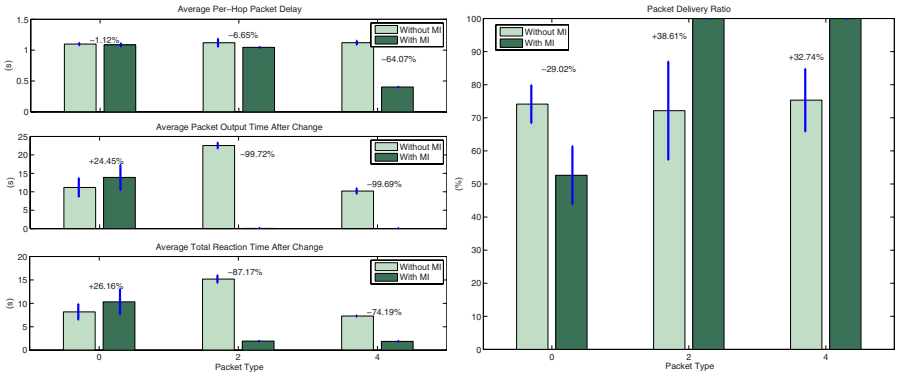

Fig. 4. Comparison of the (a) delays and (b) PDR for the patient monitoring scenario with and without the Middleware Interpreter

(type 0), the *MI* actually increases packet delay because the application does not require immediate notification. In other cases however, The *MI* helps reduce the first state change notice packet to the data sink by 50 to 75% depending on the severity of the patient's health. This delay combines per-hop delivery gains as well as faster reaction with the *MI*.

These results show that both the application and MAC protocols can adapt to proactive query notification and substantially increase the QoS to the application.

Packet Delivery Ratios. Figure 4(b) shows the benefits brought by the *MI* to the routing protocol and delivery reliability. Without the *MI*, packet delivery ratios stand at about 75% (the average of using a 100% reliable path P_1 and a $\approx 50\% = 0.8^3$ reliable path P_0). With the *MI*, delivery reliability of packets of type 0 is $\approx 50\%$ because the application can tolerate lower PDR on these packets, but 100% for packets denoting a higher patient stress level.

7 Conclusions and Future Work

WSN architectures should exhibit some important features in order to boost their popularity, most importantly support for cross-layer protocols and modularity. In this paper, we propose extending these principles to the middleware solution. A Middleware Intrepreter component extends the benefit of proactive event notification to all protocols, vertically (through layers of a same node) and horizontally (between nodes of the same neighborhood). The Middleware Interpreter provides a convenient information repository for application queries and their management. We added these ideas to X-Lisa, a cross-layer information-sharing architecture that retains a layered structure while supporting cross-layer improvements.

The originality of this work lies in the event filtering made possible by X-Lisa: in many cases, protocols need not be aware that a new sensor reading is available, only that an application query has occurred. For instance, the routing protocol may update its routes, and the node activation protocol may choose to wake up more neighbors.

We implemented these new ideas in TinyOS so as not to ignore limitations imposed by some lightweight operating systems. Simulation results showed up to 40% increase in packet delivery ratios for important packets, even as the number of data packets delivered to the data sink could fit exactly the needs of the application and the reality of the network. X-Lisa was also successful in reducing urgent packet delivery delay. These improvements (of up to 75% in packet delivery delays for the tested scenarios) were obtained with no increase in packet overhead.

Our future work will focus on facilitating other aspects of middleware systems, such as the ones cited by Wang et al. in [4]. We also plan to fully test the combined features of the Middleware Interpreter and X-Lisa.

References

1. Merlin, C.J., Heinzelman, W.B.: Information-sharing architectures for wireless sensor networks: the state of the art (submission, 2008), `http://www.ece.rochester.edu/~merlin/X-Lisa/X-Lisa_Survey_URTR.pdf`
2. Kawadia, V., Kumar, P.: A cautionary perspective on cross-layer design. In: Proceedings Wireless Communications (February 2005)
3. Römer, K., Kasten, O., Mattern, F.: Middleware challenges for wireless sensor networks. In: ACM SIGMOBILE Mobile Computing and Communications Review, vol. 6 (2002)
4. Wang, M.M., Cao, J.N., Li, J., Das, S.K.: Middleware for wireless sensor networks: A survey. Journal of Computer Science and Technology 23, 305–326 (2008)
5. Heinzelman, W., Murphy, A., Carvalho, H., Perillo, M.: Middleware to support sensor network applications. IEEE Network 18, 6–14 (2004)
6. Srivastava, V., Motani, M.: Cross-layer design: A survey and the road ahead. IEEE Communications Magazine 43, 112–119 (2005)
7. Wang, Q., Abu-Rgheff, M.A.: Cross-layer signalling for next-generation wireless systems. In: Proceedings of the IEEE Wireless Communications and Networking Conference (WCNC 2003), March 2003, vol. 2, pp. 1084–1089 (2003)
8. Conti, M., Maselli, G., Turi, G., Giordano, S.: Cross-layering in mobile ad hoc network design. IEEE Computer, 48–51 (2004)
9. Winter, R., Schiller, J.H., Nikaein, N., Bonnet, C.: Crosstalk: Cross-layer decision support based on global knowledge. IEEE Communications Magazine (2006)
10. Ee, C.T., Fonseca, R., Kim, S., Moon, D., Tavakoli, A., Culler, D., Shenker, S., Stoica, I.: A modular network layer for sensornets. In: Proceedings 7th Symposium on Operating Systems Design and Implementation (OSDI 2006) (November 2006)
11. Polastre, J., Hui, J., Levis, P., Zhao, J., Culler, D., Shenker, S., Stoica, I.: A unifying link abstraction for wireless sensor networks. In: Proceedings of the 3rd Embedded Networked Sensor Systems (SenSys 2005) (November 2005)
12. Dunkels, A.: An adaptive communication architecture for wireless sensor networks. In: Proceedings 1st ACM Conference on Embedded Networked Sensor Systems (SenSys 2007) (November 2007)

13. Merlin, C.J., Heinzelman, W.B.: Sensor network middleware for managing a cross-layer architecture. In: Proceedings DCOSS 2006 - EAWMS Workshop (2006)
14. Li, S., Lin, Y., Son, S.H., Stankovic, J.A., Wei, Y.: Event detection services using data service middleware in distributed sensor networks. In: Proceedings of the 2nd International Conference on Information Processing in Sensor Networks, April 2003, pp. 502–517 (2003)
15. Barr, R., Bicket, J.C., Dantas, D.S., Du, B., Kim, T.D., Zhou, B., Sirer, E.G.: On the need for system-level support for ad hoc and sensor networks. Operating Systems Review 36 (2002)
16. Han, Q., Venkatasubramanian, N.: Autosec: An integrated middleware framework for dynamic service brokering. IEEE Distributed Systems Online 2 (2001)
17. Liu, T., Martonosi, M.: Impala: A middleware system for managing autonomic, parallel sensor systems. In: Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2003), June 2003, pp. 107–118 (2003)
18. Belenki, Product Director, Luxoft Labs, A.: Overcoming challenges of TinyOS use in commercial zigbee applications. In: TinyOS Technology Exchange III (February 2006)
19. Madden, S., Franklin, M., Hellerstein, J., Hong, W.: TAG: a tiny aggregation service for ad-hoc sensor networks. In: Proceedings of the ACM Symposium on Operating System Design and Implementation (OSDI 2002) (2002)
20. Souto, E., Guimaräes, G., Vasconcelos, G., Vieira, M., Rosa, N., Ferraz, C.: A message-oriented middleware for sensor networks. In: Proceedings of the 2nd Workshop on Middleware for Pervasive and Ad-hoc Computing (MPAC 2004), October 2004, pp. 127–134 (2004)
21. Polastre, J., Hill, J., Culler, D.: Versatile low power media access for wireless sensor networks. In: Proceedings of the 2nd ACM Conference on Embedded Networked Sensor Systems (SenSys 2004), November 2004, pp. 95–107 (2004)