Score: A Sensor Core Framework for Cross-Layer Design

Safwan Al-Omari
somari@wayne.edu

Junzhao Du
dujunzhao@hotmail.com

Weisong Shi
weisong@wayne.edu

ABSTRACT

We present *Score*, a sensor core framework for cross-layer design in wireless sensor networks. Network components running in the context of *Score* have the ability to collaborate without the need for pair-wise interfaces. This collaboration promotes protocol optimization in the resource constrained wireless sensor networks, a technique widely known as cross-layer design. We also demonstrate the advantage of *Score* through three example network components.

Categories and Subject Descriptors

C.2.1 [**Network Architecture and Design**]: Wireless communication

1. MOTIVATION

The Internet protocol stack is widely known for its modular layered design, in which cross-layer interface is confined to adjacent layers, where a higher layer uses services provided by the layer immediately beneath it in the stack. Researchers think that this will not be the case in the future wireless sensor network (WSN) communication stack due to several reasons including limited energy supply, limited computational power, and unreliable wireless communication [2, 4]. These limitations make the need for more optimal solutions another primary requirement besides modularity. Protocol optimizations are usually possible by allowing layers to collaborate more closely when performing their functions, this technique is widely known as *cross-layer design.*

Typically, in cross-layer design, some pieces of information at one layer are used to improve the performance of another layer in the communication stack. For example, a routing protocol can consider link quality provided by a link quality service when selecting a path from a source to a destination to improve end-to-end delivery rate. Likewise, a topology management protocol can take advantage of the node's duty schedule maintained by the application to put the node into a full sleep mode –when idle– and save energy. The former example represents a traditional top-down interface, while the latter represents an unusual bottom-up interface. An effective cross-layer design framework should allow for arbitrary interface between any two protocol components, yet maintains enough degree of modularity among the different communication stack components. In this paper, we present *Score*, a framework to facilitate cross-layer design and maintain network components modularity.

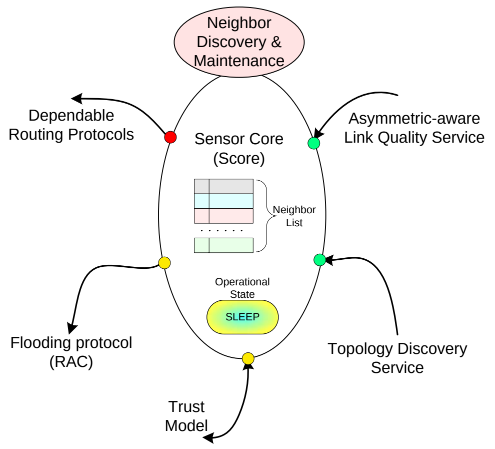
Neighbor Discovery & Maintenance
Dependable Routing Protocols
Sensor Core (Score)
Asymmetric-aware Link Quality Service
Neighbor List
Operational State
SLEEP
Flooding protocol (RAC)
Topology Discovery Service
Trust Model

Figure 1: *Score* vision as a baby frog. *Score* is depicted as the body, neighbor discovery as the head, terminals as different network components.

2. SCORE FRAMEWORK VISION

We envision *Score* as a framework that facilitates other network components to collaborate in an arbitrary fashion while maintaining a modular communication stack. As a core module, *Score* provides other components with the means to maintain and access the *neighbor set* and the *operational state*, which are the fundamental pieces of information that all the network components base their actions and optimization. Fig. 1 depicts this vision as a baby frog. The body represents *Score* module including the *neighbor list* and the *operational state.* The head represents a neighbor discovery component that maintains the neighbor list using interfaces provided by *Score.* Each one of the bay frog terminals represents a network component in the communication stack that have access to the neighbor list and monitors the operational state. Note that any interface between any of the network components (i.e., terminals) has to go through *Score* (i.e., the body). Arrows differentiate provider from consumer services. For example, the topology discovery service inserts parameters into *Score*, whereas, the dependable routing read them out to perform dependable routing.

```
interface SCore{
    // Sequential Access Iterator commands
    command result_t first();
    command result_t next();
    event result_t nextDone(uint16_t neighborID);

    // Ramdom Access Iterator command
    command result_t seek(uint16_t n_id);
    event result_t seekDone(result_t success);

    // SCore Reader
    command result_t read(uint8_t *neighbor);
    event result_t readDone(uint8_t *neighbor);

    // SCore Writer
    command result_t write(uint8_t *neighbor);
    event result_t writeDone(result_t result);
}
```

(a)

```
interface State{

   // To change the node's current state
   command result_t change(uint8_t newState)

   // Fired whenever the node's state changed
   event result_t changed(uint8_t newState);

}
```

(b)

Figure 2: *Score* APIs, (a) neighbor set abstraction API and (b) state interface.

3. SCORE FRAMEWORK FEATURES

Score provides three basic mechanisms and interfaces to facilitate network components collaboration. First, *a unified neighbor set abstraction*. Second, *a modular cross-layer interface*. Third, *a cross-layer coordination* mechanism.

3.1 Neighbor Set Abstraction API

Using `Score` access interface (Fig. 2(a)) a network component can read or write any neighbor record simply by pointing at the required record and performing a read or a write. Moving the pointer can be done in two ways, sequentially using the *first* and *next* commands, or randomly using the *seek* command (Fig. 2(a)). Following nesC/TinyOS philosophy, *Score* provides split-phase operations to keep the sensor node responsive to external events [3]. *Score* does not impose any limitations and is not involved in deciding which nodes are included in the neighbor set. In other words, *Score* only provides the mechanism and not the policy. Refer to [1] for an example neighbor discovery service implementation.

3.2 Cross-layer Interface

Score plays a significant role in decoupling network components, by providing a mechanism for them to interface and communicate without the need for pair-wise interfaces. *Score* defines a global neighbor record structure, in this structure, each network component is allocated a number of bytes. The network components can use these bytes to annotate the neighbors with useful information that other components wish to access. For example, A trust network component can rank the neighbors based on some trust criteria, and annotate the neighbors with this value. Another component, the routing protocol for example, can access these trust values through *Score* and exclude untrussed neighbors while building a routing tree.

To keep the `Score` access interface simple and general, *Score* does not provide individual read and write commands to read and write specific fields in the neighbor record, it only supports reading and writing entire records. By doing so, *Score* is not severely involved and dependent on a particular neighbor record structure, which we think can change in different WSN applications. Reading and writing entire records raises the need for *Score* to prevent network components unintentionally or intentionally (malicious component implementations) from overwriting each other's information in the neighbor record. Therefore, each network component is assigned a *writing mask*, This mask (for short) is statically defined in *Score* according to the current neighbor record structure. Each time a network service writes a neighbor record, *Score* will first apply the mask, on the new record, which sets all the unrelated bits to zeros, and then perform a bit-wise *OR* operation with the old neighbor record. The masking process does not only provide inter-protocol overwrite protection, it also allows for multiple writers at the same time with no need for inter-component synchronization (each component writes its own bytes only in the shared neighbor record).

3.3 Cross-layer Coordination

Score supports cross-layer coordination by maintaining a sensor node *operational state*, this *state* (e.g., `DISCOVERY`, `BOOTED`, `SLEEP`, and `ACTIVE`) describes the current sensor node operational status. Each network protocol can react in its own way when a new *state* is announced by *Score*. For example, a *neighbor discovery* service will send neighbor probing messages if the node state change to `DISCOVERY` (a `DISCOVERY` state means there are no enough neighbors in *Score*), while a routing service will hold its protocol messages as there are no enough neighbors to maintain a routing tree, and so save the precious node's energy from being wasted for nothing. *Score* provides a *state* interface (Shown in Fig. 2(b)), which provides a command to change the node's current state and uses an event to announce state changes. Any network component wishes to react to state changes must provide implementation of the *changed* event, in which the component can take the appropriate action.

4. CASE STUDIES

In this section we show some examples on network components running in the context of *Score* and how these components can interface through *Score* without the need for pair-wise interfaces, to achieve better performance.

4.1 Topology Discovery Service

On its own, the topology discovery service does not map into any of the OSI reference model layers and it represents a typical network service that solely supports cross-layer design. Other traditional network layers, such as MAC and routing layers, use the topology parameters maintained by the topology discovery service in order to improve their performance. In order to build and maintain these parameters, the topology discovery service actively sends and receives protocol messages, for example, neighboring nodes exchange their neighbor lists to find *communication redundancy* and *freshness*. At node x with neighbor list (NS_x), *communication redundancy* and *freshness* are defined for each node $(y \in NS_x)$ as the cardinality of $(NS_x \cap NS_y)$, and $(NS_y \setminus NS_x)$ respectively. Fig. 3 shows two nodes x and y with their re-

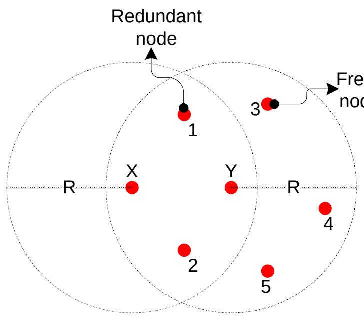

Figure 3: Several nodes with their respective communication ranges (R) are shown. X has a communication redundancy and freshness values of 2 and 3 with node Y respectively. Nodes 1 and 2 are redundant, while nodes 3 , 4, and 5 are fresh.

spective communication ranges and neighbor lists. Node x has a *communication redundancy* and *freshness* of 2 and 3 respectively with node y. Fig 4 shows the skeleton of the topology discovery service in *Score*.

4.2 Redundancy-Aware Controlled Flooding

The topology discovery network component acts as a service provider in *Score*. To complete the picture, we present a controlled flooding network component that acts as a consumer service in *Score*, we refer to this component as **R**edundancy-**A**ware **C**ontrolled flooding (denoted as **RAC**). **RAC** exploits topology redundancy information provided by the topology discovery service in order to reduce the total number of transmissions required to disseminate a data message.

In a basic flooding protocol (denoted as **Blind**), the sink starts the process by broadcasting the message to its neighbors, which in their turn, re-broadcast the message again. The broadcast/re-broadcast process is repeated recursively until all the nodes in the network receives the message. Making all the receiving nodes to re-broadcast results in many unnecessary transmissions. In contrast, among the receiving nodes in **RAC**, only the node that has the least communication redundancy with the sender re-broadcasts the message. This increases the chance that the data message will reach more nodes that have never seen the message before, and so, reduces the required number of transmissions overall. We also compare **RAC** to **R**andom **C**ontrolled flooding protocol (denoted as **RC**), in which, the forwarder node is chosen randomly. We refer to **RAC** and **RC** together as controlled flooding protocols.

4.2.1 Controlled flooding protocols

Based on who (i.e., sender versus receivers) decide(s) the node that should re-broadcast a message, we can differentiate two approaches to controlled flooding (i.e., sender-based and receiver-based). In sender-based flooding, the sender node chooses the node that should pick up the flooding process, while in the receiver-based flooding, receiver nodes decide among themselves a single node to pickup the flooding.

The sender-based controlled flooding approach maintains a single thread of flooding in the network at any time, which potentially keeps the total number of transmissions as low as possible. However, if the selected node to re-broadcast has already seen a copy of the message, it will not be interested to re-broadcast, and therefore, the only thread of flooding will vanish leaving some nodes in the network unaware of the message. Therefore, sender-based approach does not provide enough guarantee that all nodes get the message.

```
module TopologyDiscoveryM{
 uses interface Score;
 ...
}
implementation{
 // pointer to the current neighbor record
 NeighborRecordPtr p;
 int16_t redundancy, freshness;
 ...
 event TOS_MsgPtr Receive.receive(TOS_MsgPtr m){
   // Receive message from y and
   // save neighbor list into NS(y)
   ...
   // Start calculating redundancy and freshness
   Score.first()
 }
 event result_t nextDone(uint16_t neigborID){
   // Got the next neighbor in my neighbor list
   ...
   for all e in NS(y){
     if (e == neighborID)
       redundancy++;
   }
   // move to next neighbor in my neighbor list
   if (!call Score.next()){
     // end of neighbor list, then find freshness
     freshness = |NS(y)| - redundancy;
     //put the pointer at y's record
     call Score.seek(y);
   }
 }
 event result_t seekDone(result_t suc){
   // update values of y's record
   p->redundancy = redundancy;
   p->freshness = freshness;
   call Score.write(p);
   ...
 }
}
```

Figure 4: Topology discovery skeleton implementation in *Score*. Upon receiving a neighbor list from node y, current node loops over the neighbor set using *Score* and calculates communication redundancy and freshness. Finally, current node updates relevant bytes in y's neighbor record.

After receiving a message in the receiver-based controlled flooding approach, each one of the receiver nodes backs off for a specific amount of time before re-broadcasting. During the back off period, a node suppresses its transmission once the node gets another copy of the message from another node in the vicinity. It is clear that several flooding threads may exist in the network at the same time; however, this approach makes sure that at least one node eventually picks up the flooding process and so, all nodes in the network get at least one copy of the message. The remaining question is how the receiver nodes calculate their back off time? this is where we differentiate **RAC** from **RC**. In **RAC**, each receiver node sets the back off time proportionally to its communication redundancy with the sender node, so that receiver nodes with least communication redundancy with the sender re-broadcast first, as a result, more new nodes get the message. In **RC**, back off times are chosen randomly.

4.2.2 Simulation Setup and Evaluation

We implement **Blind**, **RC**, and **RAC** using *Score*, Fig. 6 shows the skeleton of **RAC** implementation. Note that **RAC** has access to the communication redundancy infor-

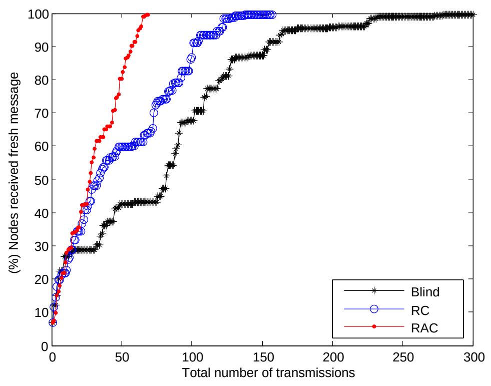

Figure 5: Comparing the performance of Blind, RAC, and RC protocols.

mation through *Score Access* interface. To show the ultimate advantage of **RAC** cross-layer design, we compare its performance to that of **Blind** and **RC** protocols in terms of the number of nodes that receive a fresh message copy for each message transmission per flooded message.

Using each flooding protocol, we disseminate 100 messages from the sink in a wireless network of 300 nodes distributed randomly over a (100x100) units squared sensor field. The nodes' nominal communication range is 15 units. We take average over the 100 message floodings and present the data in Fig. 5. The Figure shows the cumulative number of fresh messages received, shown as the y-axis, for each message transmission in the network, shown as the x-axis. We observe two important results. First, **RAC** outperforms **RC** and **Blind** as for each transmission, there are always some nodes that receive the message for the first time (i.e., the line consistently increasing), while, in **RC** and **Blind**, many transmissions are useless, shown when the line moves horizontally. Second, **RAC** requires around 70 transmissions in total to disseminate a message compared to over 150 and 300 for **RC** and **Blind** respectively.

4.3 Link Quality Service

As another example on a service provider in *Score*, we present asymmetry-aware link quality service. This component basically measures and estimates the packet reception rate with the neighbors and the timelessness link quality information and annotates the corresponding neighbor records for other components in *Score*.

In lossy wireless sensor network, if a node, A, can receive packets from a neighbor, B, node B will be identified as an inbound neighbor of node A. If node A can send packets to a neighbor, C, node C should be identified as an outbound neighbor of node A. However, to identify node C as its outbound neighbor, node A should receive acknowledgement from node C. Due to asymmetric links, if node A can not receive acknowledgement from node C, then node A can not identify node C as its outbound neighbor. Furthermore, the link quality, in term of packet reception rate, from node A to node B is different from that from node B to node A. Therefore, every node should distinguish inbound neighbors and outbound neighbors.

This service makes use of the combination of active probing and passive listening techniques to measure the link quality. Passive listening technique intercepts incoming messages to update the measurement results and the estimated link quality to outbound neighbors. Active Probing sends link quality measurement messages to probe the neighborhood and the estimated link quality for the inbound neighbors. It leverages Window Mean Exponentially Weighted Moving Average Estimator (WMEWMA) to estimate the link quality based on current and history measured results. WMEWMA uses a time window to observe the received packet and it adjusts the estimation result using latest average value of packet reception rate.

The asymmetry-aware link quality service can be leveraged by several higher-level protocols. For example, a routing protocol could consider the link quality of neighbors when building a routing tree to improve delivery rate. A MAC protocol, on the other hand, could treat inbound and outbound neighbors differently.

```
module RACM{
 uses interface Score;
 ...
}
implementation{
 // pointer to the current neighbor record
 NeighborRecordPtr p;
 ...
 event TOS_MsgPtr Receive.receive(TOS_MsgPtr m){
   // Receive message m from y
   if (m is never been received before){
     // use Score to find out redundancy with y
     //and use it as backoff time
     call Score.seek(y)
   }else{// need to suppress forwarding
     call BackoffTimer.stop();
   }
 }
 event result_t seekDone(result_t suc){
   // y's record is found, read record to find
   // redundancy with y
   call Score.read_full_record(p);
   ...
 }
 event result_t read_full_record(p){
   // backoff proportionally to redundancy
   call BackoffTimer.start(p->redundancy);
 }
 event result_t BackoffTimer.fired(){
   // re-broadcast the m
 }
}
```

Figure 6: RAC skeleton implementation in *Score*. Upon receiving a message, current nodes finds redundancy with the sender and backoff accordingly.

5. REFERENCES

[1] S. Al-Omari and W. Shi. Rat: Redundancy-aware topology control in wireless sensor networks. Technical Report MIST-TR-2005-012, Wayne State University, Nov. 2005.

[2] M. Chiang. To layer or not to layer: Balancing transport and physical layers in wireless multihop networks. In *Proc. of INFOCOM'04*, Mar. 2004.

[3] D. Gay, P. Levis, R. Behren, M. Welsh, E. Brewer, and D. Culler. The nesc language: A holistic approach to networked embedded systems. In *Proc. of PLDI'03*, June 2003.

[4] U. Kozat, I. Koutsopoulos, and L. Tassiulas. A framework for cross-layer design of energy-efficient communication with qos provisioning in multi-hop wireless networks. In *Proc. of INFOCOM'04*, Mar. 2004.