RESEARCH

The Smart Surface Network: A Bus-Based Approach to Dense Sensing

Farha N Ali, ^{1??}, Yvon Feaster², Jiannan Zhai² and Jason O. Hallstrom³
, (561) 297-3000

Abstract

We present the Smart Surface Network (SSN), a hardware and software platform designed for dense sensing. Sensor nodes connected to the SSN communicate using a serial bus integrated within a mountable physical surface. The hardware architecture and bus access and communication mechanisms are implemented in a self-stabilizing manner, providing robust handling of unannounced arrivals and departures of network devices. An associated API supports a peer-to-peer communication paradigm, providing access to the physical, data link, and application layers of the bus. In this paper, we describe the SSN hardware architecture and present the bus access and peer discovery algorithms. We also discuss the design of the API and describe experimental results characterizing the fairness of the bus algorithm, the efficiency of the peer discovery algorithm, and the performance of the SSN system under varying load conditions.

Keywords:

bus-based communication; dense sensing; hardware/software co-design; indoor sensing

1 Introduction

Consider the possibility of being able to control the lighting, HVAC, entertainment center, window coverings, and security system in your home simply by pushing a sensor into a wall, similar to pressing a thumbtack into a corkboard. We envision smart homes, classrooms, warehouses, and other environments with smart walls that allow a variety of sensors to be inserted or removed at will. These sensors will have the ability to communicate with one another, even in dense settings, without the worry of wireless interference or packet loss.

Lifton et al. introduced a hardware and software platform for Pushpin Computing [1, 2] to address the goals of providing power to a dense network of nodes. This effort was the point of departure for our Smart Surface Network (SSN), a hardware and software platform that provides dense sensing capabilities on a mountable “smart surface”. A key differentiating characteristic of our work is the seamless integration of *communication* hardware and software within the surface. We anticipate a future need to place hundreds, if not thousands of sensors in close proximity, in applications like smart floors [3], sensate skin [4], and dense media capture [5].

Power constraints are a common challenge faced by sensor networks, with batteries serving as a common power source. To use limited energy reserves efficiently, sensor systems often use policies to periodically switch the system between passive and active states, saving energy during the passive state. To avoid problems related to battery depletion, reliable power delivery may be achieved through the network architecture, rather than software policies. The SSN uses two conductive layers to deliver power to attached sensor nodes through the smart surface.

Wireless communication presents additional challenges for sensor networks. A node’s transmission range is limited. If the destination node is not within this range, messages must be propagated by intermediate nodes. Multi-hop routing algorithms are essential for efficient packet delivery [6, 7, 8], but introduce complexity, latency, and increased power consumption. The resulting internal interference is exacerbated by the physical layer used by common wireless standards, such as ZigBee [9], ISA 100.1a [10], and others. The physical layer in these standards relies on IEEE 802.15.4 [11], operating in the unlicensed band of 2.4 Ghz, which is also used by many household appliances. Hence, nodes in a wireless sensor network not only suffer from interference from their peers, but also from their surroundings, contributing to both packet and energy loss. A wired bus medium does not suffer from these issues. In our solution, to provide reliable communication between nodes connected to the SSN, the surface adds a third layer of conductivity, which serves as a communication bus. Devices mounted on the surface communicate through this bus. This is in contrast to the Pushpin Computing approach, which relies on capacitive coupling or infrared to support wireless communication [1].

For a sensor device to communicate effectively with its peers using a shared bus, several requirements must be satisfied.

- 1 Each device must be granted access to the bus.
- 2 To ensure the integrity of data sent over the shared medium, collisions must be avoided using schemes such as those described in [12, 13, 14]; or must be detected using techniques such as those described in [15, 16, 17]. The SSN bus uses carrier sensing multiple access with collision avoidance (CSMA/CA)

by electing one device as bus master prior to any data exchange. The leader election problem is addressed in many research areas [18, 19, 20, 21]; our approach relies on lightweight ID bit arbitration. Note that TDMA offers an alternative collision avoidance strategy, but is not appropriate for the SSN bus because: (1) TDMA suffers from a single point of failure, as time slot synchronization requires a shared clock. (2) TDMA-based contention resolution schemes perform poorly if network traffic is bursty. (3) TDMA-based schemes cannot support unannounced insertions and removals of network nodes, as time slot allotment must be done prior to network deployment.

- 3 Each device must be able to identify/address other devices in the network. Wireless networks use peer discovery algorithms to determine their neighbors in order to determine efficient transmission schemes, as in [22, 23, 24]. In a bus network, these schemes are not applicable and/or not needed because of the broadcast nature of the bus mechanism. To identify and discover devices in the network, the SSN relies on a novel peer discovery algorithm.
- 4 Each device must be able to send commands and data to other devices. Hence, a set of protocols must be developed to send data and commands. This is achieved through an API used to program SSN devices, which provides a complete set of peer-to-peer communication primitives. The SSN is a complete system for sensor networking based on an integrated communication bus and power supply.

It is worthwhile to note that the bus-based sensing solution described in this manuscript is complementary to existing wireless sensor networking technologies. In real applications, sensor deployment density can vary significantly, ranging from very sparse networks (e.g., meteorological networks) to very dense networks (e.g., sensate walls and floors). For sparse deployment scenarios, wireless technologies provide strong solutions. However, as density increases, there is a pivot point where our bus-based approach begins to provide value. Beyond the simplicity it affords in terms of power delivery and device installation, the bus-based approach provides significant power, cost, and form-factor savings, as well as protection against external wireless interference. The radio chip and supporting circuitry on most wireless sensing platforms is responsible for (1) the majority of the platform's power consumption, (2) a significant fraction of the overall production cost, and (3) a significant fraction of the overall form-factor. As the number of devices deployed within a limited surface area grows from tens to hundreds to thousands, these factors are increasingly significant. The bus-based approach provides a complementary technology solution that eliminates the whole of the radio circuit, allowing operation from virtually any modern microcontroller, yielding significant power, cost, and form-factor savings. Even in these scenarios, however, the relationship with wireless technologies is complementary. We envision hierarchical applications of the smart surface network where dense, bus-based sensing fabrics are connected to higher-tier sensor networks via wireless links.

Contributions (1) We present a surface-based networking architecture which provides power and a wired-AND communication bus to attached devices, mitigating battery depletion issues and radio interference problems. (2) We present a reference

sensing platform that uses variable length pins to connect to the mountable surface. (3) We present an election algorithm to ensure fair bus access contention resolution. (4) We present a novel peer discovery algorithm to ensure robust network operation in cases of unannounced arrivals and departures of network nodes. (5) We present an API with a complete set of peer-to-peer communication primitives that enable devices to communicate via the SSN bus. (6) We present experimental results characterizing the fairness of the election algorithm, the efficiency of the peer discovery algorithm, and the performance of the SSN system under varying load conditions.

Organization The remainder of the paper is organized as follows. Section 2 discusses related work. Section 3 describes the SSN hardware. Section 4 discusses the election algorithm, the peer discovery algorithm, and the SSN API. Section 5 presents evaluation results detailing the efficiency of the peer discovery algorithm, the fairness of the election algorithm, and the performance of the SSN system under varying load conditions. Finally, section 6 presents a summary of contributions and pointers to future work.

2 Related Work

We conduct a review of the most closely related work in the literature. We focus first on platform support for dense sensing. Next, we discuss available bus communication standards comparable to the SSN approach with respect to device addressing, device discovery, and contention resolution for media access.

2.1 Hardware

Most closely related to our work is the Pushpin Computing initiative introduced by Lifton et al. [1]. In their approach, the pushpin surface is composed of two supply layers, positive and ground, consisting of aluminum foil encased in layers of polyurethane foam. The sensing hardware consists of four stacked printed circuit boards (PCB) for power, communication, processing, and sensing, respectively. The bottom-most PCB has two protruding pins that supply power when pushed into the layers of foam and aluminum foil. The communication modules support capacitive coupling and infrared. The processing module consists of a Cygnal C8051F016 processor and supporting circuitry. The sensor layer provides application-specific sensors. There are several important differences between the SSN and Pushpin Computing approaches. Most obviously, the SSN's hardware architecture provides three layers of conductivity, one each for power, ground, and communication. This results in a fundamentally different communication approach. Infrared communication used in Pushpin Computing requires line-of-sight alignment between the receiver and the sender, and the communication quality depends on the spatial proximity of the nodes. The SSN uses a wired bus and does not suffer from either of these issues.

Laerhoven [25, 26] presents *Pin & Play*, another hardware platform based on a pluggable surface. The Pin & Play hardware platform provides two layers of conductive material separated by a thin layer of cork. The first provides a communication bus and (parasitic) power. The second is a ground layer. Similar to the SSN, the conductive material used in Pin & Play is woven fabric sheets made of 0.1 mm thick silver or nickel-copper plated nylon [27]. The most significant difference between Pin

& Play and the SSN is that Pin & Play devices rely on the Dallas/Maxim 1-Wire protocol [27, 28]; the approach does not support network application programming. Only applications that use components compatible with the 1-Wire network protocol are applicable in their approach, and this set is quite limited. We developed a new communication protocol for the SSN, which supports full network application programming.

It is also useful to consider dense sensing efforts that could benefit from a network platform like the SSN. Conner [29] describes a multi-hop wireless network of motion sensors designed to help workers find empty conference rooms. The devices are wired to the light switches in each room. Data from motion sensors is transmitted to a central server and used by a web-based application to indicate which conference rooms are free. Using the SSN bus would enable the central server to collect data from sensor nodes directly, thus eliminating the need for data transmission across multiple hops.

Lifton [2] describes *Tribble* (Tactile Reactive Interface Built By Linked Elements), consisting of 20 hexagonal and 12 pentagonal tiles connected to form a sphere resembling a soccer ball. Each tile is equipped with a circuit board covered by a plastic veneer. The tiles communicate neighbor to neighbor. Tribble was designed as a testbed for dense sensing to investigate applications involving multi-modal electronic skins.

We observe that communication among nodes in the last effort is neighbor-based, i.e., the nodes communicate only with their immediate neighbors. The SSN bus supports both broadcast and unicast communication among all network nodes.

2.2 Communication

Communication among devices connected to the smart surface is supported by the built-in bus, which provides device addressing, device discovery, and contention resolution for media access. In this section, we discuss how these issues are handled in comparable bus communication standards. We focus on 1-Wire, I2C, SMBus, Controller Area Network (CAN) bus, FlexRay [30] and, the Local Interconnect Network (LIN) bus [31].

1-Wire is a device communication bus system designed by Dallas Semiconductor Corporation and provides low-speed data, signaling, and power over a single wire. It has been used in a number of applications [32, 33, 34, 35]. I2C is a bus communication protocol developed by Phillips for attaching low-speed peripherals to embedded systems and motherboards. SMBus is an improvement over I2C used in many applications including [36, 37, 38, 39, 4]. The Controller Area Network (CAN) bus is an ISO-defined serial communication standard developed during the 80's by Robert Bosch for the automotive industry. CAN bus is the focus of many research efforts [40, 41, 42, 43, 44, 45]. FlexRay is an automotive network communication protocol developed by the FlexRay Consortium to govern on-board automotive computing. It is discussed in several research efforts [46, 47, 48]. The Local Interconnect Network (LIN) bus is a serial communication protocol used to connect components in vehicles. This protocol is a product of the LIN Consortium founded by five automakers (BMW, Volkswagen, Audi Group, Volvo Cars, Daimler Chrysler). The technology expertise (networking and hardware) was supplied from Volcano Automotive Group and Motorola. LIN bus was the focus of research efforts presented

in [49, 50]. In the following discussion, we focus on how device addressing, device discovery, and contention resolution for media access services are realized in these bus communication standards.

Device addressing involves assigning a unique identifier to each device. Two factors distinguish addressing schemes: the information content embedded in the identifier and the time of assignment. 1-Wire addresses are assigned during the manufacturing process and are 64 bits long, consisting of an 8-bit family code, a 48-bit device address, and an 8-bit CRC code. The 48-bit device address is guaranteed to be unique. As discussed in [51], the maximum number of devices connected to a 1-Wire network, in worst case current and temperature conditions, is 98 [51]. Although the number of devices can be far greater than 98, reserving 48 bits for each device address is inefficient. The I2C standard uses 7, 8, or 10-bit addresses, also assigned during the manufacturing process. The least significant bit in I2C is reserved to control the direction of communication. SMBus is an improvement over I2C and uses the same address structure, but includes an address resolution protocol for dynamic allocation of the addresses. Default addresses are assigned during manufacturing, but a master can assign a new address to a SMBus device if it chooses, e.g., due to a conflict. CAN bus uses a publish/subscribe model. Rather than device IDs, messages are given IDs. These IDs are used to signify message importance. FlexRay uses Time Division Multiple Access (TDMA) to resolve contention among the nodes. The LIN bus uses the bus shunt method, or the extra wire daisy chain (XWDC) method [52] to assign IDs to the nodes. Both of these methods require slave nodes to provide two extra pins, one for input (D1) and another one for output (D2). In the XWDC method, a daisy chain of slave nodes is formed by connecting the D2 pin of the each node to the D1 pin of the subsequent node. The first node's input pin is hardwired to ground. Only the nodes that have not yet received an address and have their input set to low can accept the address sent on the bus. As a node accepts the address sent from the master, it pulls its output pin low, thus enabling the subsequent node to receive the next address sent. The limited literature available on the bus shunt method suggests that nodes are connected in a daisy chain, and a measurement of the current is used to select a node and assign an address to it.

Only 1-Wire is able to handle unannounced arrivals and departures of nodes from the network, though the standard has its own limitations, as discussed above. In the remaining bus communication standards, the network structure must be decided before deployment. Any additions or removals will compromise the operation of the network. SSN supports the addition and removal of network nodes without any disruption in the network's functionality.

Novelty Our approach is differentiated from prior work in several important ways. First, the surface hardware architecture provides three layers of conductivity, one each for power, ground, and communication. This results in a fundamentally different communication approach compared to wireless or parasitic power communications. In parasitic communication, bus synchronization is achieved by pulling the power/data line low, forcing all devices to reset and lose state. In our implementation, separate data and power lines allow devices to retain state while synchronizing.

Second, the SSN supports “hot” reconfiguration of the network; i.e., nodes can be added or removed from an operational network without causing disruption. Third, the network peer discovery algorithm is more efficient (and easier to implement) as compared to other bus-based peer discovery algorithms. Fourth, the network API provides support for developing custom applications (instead of restricting the usage of the system to manufacturer-provided services). Fifth, the architecture provides a peer-to-peer communication model, affording each device a chance to act as a sender or a receiver. Finally, the SSN metadata management features afford simple reflection of hardware changes in firmware (e.g., addition and removal of integrated sensors) and provide support for using custom network addressing schemes.

3 System Hardware Architecture

The SSN architecture presents three design challenges. The first is the design of the surface. The second is the design of the communication model. The third is the design of the hardware platform to support SSN-based networking. Here we discuss the development of the surface, the initial testing hardware, and the platform used in the final design. The software design is discussed in Section 4.

3.1 The Surface

The networking surface is realized as a wooden box framing three layers of aluminum foil and conductive fabric, each separated by smooth Styrofoam, as shown in Figure 1a. The box is 11-inches wide and 72-inches long. The conductive layers are contacted using three metal pins of varying length built into the sensor devices developed for the surface. Each conductive layer comprises heavy duty aluminum foil stitched to conductive fabric consisting of 76% nylon and 24% elastic fibers, coated in 99.9% Silver [53]. Each layer has a copper wire running the full length of the aluminum, as shown in Figure 1b, to ensure electrical continuity throughout the surface. The bus layer is held high with a weak pull-up resistor.

3.2 Proof of Concept Device

The first devices created to test the power layers of the SSN were simple LED/resistor pairs, as shown in Figure 2a. In place of perf board, a small, 4-hole button was used to house an LED and a resistor. Since the leads of an LED are pliable, they were replaced with straight pins, though these did not prove to be sturdy enough to penetrate the substrate. Several sizes of brad nails were also tested, none of which worked well. After several attempts, nickel-plated (steel) T-pins were used. These pins were the correct size, diameter and length, and were sturdy enough to penetrate the substrate layers.

As shown in Figure 3, two of the three pins pass through multiple conductive layers, which could cause a short circuit. To avoid this, portions of the pins are coated with an insulating epoxy. Unlike other products tested (e.g., nail polish, paint), marine epoxy [54] was durable enough to withstand numerous insertions and retractions of the device from the surface.

To test all of the conductive layers, a second device was created. As shown in Figure 2b, this device included a 1-Wire temperature sensor [55], and a .10uF ceramic capacitor soldered on triangle-shaped perf board. We wrote a simple program to read the temperature from the device using the 1-Wire protocol. This test was used to test the power and communication layers of the SSN realization.

3.3 Smart Surface Hardware Design

3.3.1 Stackable Hardware Device

Inspired by other stackable sensing platforms [56, 57, 58], we developed a hardware platform consisting of three 45mm x 45mm printed circuit board (PCB) layers to evaluate the peer-to-peer protocol and API developed for the SSN. Each layer is connected using two sets of Hirose connectors [59]. The device stack is depicted in Figure 4d. Below is a description of each layer of the device stack.

Pin (bottom) Layer As shown in Figure 4a, the pin layer consists of three varying length nickel plated steel T-pins protruding from the bottom of the PCB, providing power and communication contacts for the device. The pins provide connections to the appropriate conductive layers of the surface. They are durable enough to withstand numerous insertions and retractions. To prevent short circuits when the device is inserted and retracted from the surface, each pin is carefully coated with marine epoxy, leaving only enough of the pin exposed to make a connection with the appropriate conductive layer.

Controller (middle) Layer As shown in Figure 4b, the controller layer, which serves as the core of the device, provides power regulation and computation support. The regulator is able to take up to 30V and supplies 5V at the output. The 8-bit microcontroller, an ATmega168 [60], is equipped with 1 Kbyte of RAM and 16 Kbyte of in-system, self-programmable flash memory. A 10MHz external crystal provides the microcontroller an accurate ($\pm 30ppm$) and stable clock source. Three LEDs—red, green, and yellow—indicate the status of the device. A standard 6-pin header is used for microcontroller programming.

Sensor (top) Layer To demonstrate Smart Surface applications, the sensor layer consists of three sensors and an open ADC interface, as shown in Figure 4d. A temperature sensor, a passive infrared sensor, and a electret condenser microphone are used to gather common environmental information. Five open ADC ports are available to add additional sensors as needed.

4 System Software Architecture

The SSN software stack includes an election algorithm to provide each device an opportunity to gain access to the bus, a peer search algorithm to identify the devices present in the network, and an API to provide access to the physical, data link, and application layers of the bus. The API also provides functionality to store and modify metadata within each network device.

4.1 Election Algorithm

To provide each device an opportunity to access the bus, a periodic election is held. A device can compete in an election only if it finds the bus inactive for a specified period of time, or if it receives a **reset** command from the device currently controlling the bus.

The election algorithm is presented in Listing 1. Each device begins the election in the **compete** state (line 1) and reads the bus after a short random delay (lines

2-4). If the device finds a high value on the bus (the default floating state), it sets its state to `master` and holds the bus low for a specified period of time (lines 5-10) to signal its claim on the bus. If the device observes a low value on the bus, another device has gained access to the bus, so it defers to the current master, sets its state to `get_command`, and waits for the bus to be released (lines 11-14). Note that this is a wired-AND bus; the value of the bus at a given instant in time is the logical AND of the values sent by all devices.

The device that has claimed the bus must keep it low long enough to allow all other devices to observe the bus and determine that another device has claimed it. To achieve this, we limit the random delay to 100 microseconds (line 3), and the device that pulls the bus low must keep it low for 300 microseconds (line 8). In our trials, the 200 microsecond difference appears to be a reasonable compromise between waiting too long and accommodating variable election start times across the network.

Although unlikely, multiple devices may generate the same (or very close) random numbers, and after waiting, they may claim the bus at the same time, each assuming itself to be the master. To address this (unlikely) problem, a second stage election is held using device IDs after the master candidate devices have released the bus.

To confirm its election as a master, each master candidate repeats the following process for each bit in its ID, beginning with the lowest order bit (lines 15-39). If the ID bit is a 1, the device waits for a duration equal to one-half of a bit period, then checks the bus. If the device finds the bus low, it sets its state to `get_command`, thus deferring in favor of any master candidates with 0 at that ID bit position. The device then waits for the bus to change state to detect a synchronization signal at three quarters of a bit period (discussed later), thus synchronizing itself with the device sending the 0. It then waits for a quarter of a bit period (lines 18-25) to complete the bit period. The intra-bit signal transitions are used for synchronization of the devices (explained later). If the device finds the bus high, it deduces that no other device is sending a 0 for this bit position. It then waits for another quarter of a bit period, sets the bus low, thus providing synchronization for the devices reading the bus, and then keeps the bus low for a quarter of a bit period (lines 26-31). Otherwise, if the ID bit is 0, the device pulls the bus low for three-quarters of a bit period, then releases the bus and waits for a quarter of a bit period before sending the next bit (lines 33-38).

A device not competing in the election observes the bus transitions during the election and synchronizes itself during the transitions at the three-quarter bit periods (lines 40-50). It reads the bus at the half-bit periods, and then waits for the subsequent transition. After observing the transition, the device then waits for one-quarter of a bit period to end the bit. At the end of the ID sending process, the slave nodes set their state to `get_command` (lines 53-55).

The bus transitions at three-quarters of a bit period are used to synchronize devices in the network. Once a device is elected master, it provides a synchronization signal by sending short pulses on the bus to start each bit. During the election process, however, there is no master, hence synchronization is achieved by observing the transitions at the three-quarter marks. The approach is similar to that used in Manchester encoding [61]. Figure 5 shows how the devices synchronize with their

peers. The figure shows a master device sending a 1, and then a 0 on the bus, while three slave devices are reading the bus. To send a 1, the master keeps the bus high for three-quarters of a bit period, then pulls it low for a quarter of a bit period. We see some jitter in the slaves' measurement of half of a bit period. To accommodate the jitter, all slave devices continuously poll the bus for the transition after reading the bus at the half-bit period. The transition indicates three-quarters of a bit period at the master, hence all slave devices synchronize with the master. When sending a 0, the master keeps the bus low for three-quarters of a bit period, and then leaves it high for one-quarter of a bit period.

After all of the ID bits are sent (and read), there is only one master among the devices, and the remaining devices prepare themselves to accept commands from it. When the elected master is ready to release the bus, it sets its state to `get_command` and sends the `reset` command. Requiring a transition from `master` to `get_command` (rather than `compete`) prevents repeated reelection of the same master. On receiving a `reset` command, all slave nodes set their states to `compete` in preparation for the next round of election.

In rare cases, a device wishing to acquire the bus may finish waiting during the first stage (lines 3-4), at an instant when a master candidate is just releasing the bus. The interpretation of the bus value during the transition may be interpreted as either high or low. A late device may read the bus value as high and try to pull the bus low to declare its candidacy. At the same time, the other devices in the network may then assume that master election has begun. As the late device will hold the line low for a duration longer than one bit period, and all devices in the network look for transitions at three-quarters of a bit period for synchronization, the devices will get out of sync with the master candidates. Further, when the late device releases the bus, transitions from the master candidates, and the transition introduced when the late device releases the bus will send the wrong bit values on the bus. All master candidates, including the late node, may assume that another candidate has won the election. In this case, an election may end without a confirmed master, and no commands will be sent on the network. SSN devices are programmed to detect inactivity, and if the period of inactivity exceeds a given value, each device sets its state to `compete` and starts a new election.

4.2 Peer Search Algorithm

To determine the devices present on the bus, and to support unannounced arrivals and departures from the network, the SSN software provides a peer search algorithm. The algorithm relies on the fact that each device is configured with a unique ID.

The master search algorithm is shown in Listing 2, and the slave search algorithm is shown in Listing 3. To begin the peer search process, the master device issues a `start_search` command (Listing 2, line 2). All slave devices signal their presence by setting the bus low (Listing 3, line 1). The master device reads the bus. If it finds the bus low, it starts the process of reading the ID bits of devices present

```

1 state=COMPETE
2 i=random_number(1,100)
3 wait_micro_seconds(i)
4 line=read_bus()
5 if(line==1){
6     state=MASTER
7     set_bus_low()
8     wait_microseconds(300)
9     release_bus()
10 }
11 else{
12     state=GET_COMMAND
13     wait_while_bus_low()
14 }
15 for bitCounter:1 to 7{
16     bit=id_bit@bitCounter
17     if(state==MASTER){
18         if(bit==1){
19             wait_half_bit_period()
20             line=read_bus()
21             if(line==0){
22                 state=GET_COMMAND
23                 wait_while_bus_low()
24                 wait_quarter_bit_period()
25             }
26             else{
27                 wait_quarter_bit_period()
28                 set_bus_low()
29                 wait_quarter_bit_period()
30                 release_bus()
31             }
32         }
33         else{
34             set_bus_low()
35             wait_3_quarter_bit_period()
36             release_bus()
37             wait_quarter_bit_period()
38         }
39     }
40     else{
41         wait_half_bit_period()
42         line=read_bus()
43         if(line==0){
44             wait_while_bus_low()
45             wait_quarter_bit_period()
46         }
47         else{
48             wait_while_bus_high()
49             wait_quarter_bit_period()
50         }
51     }
52 }
53 if(state==GET_COMMAND){
54     accept_commands()
55 }

```

Listing 1: Election Algorithm

```

1 line=0
2 start_search()
3 line=read_bus()
4 while(line==0){
5     send_handshake()
6     line=read_bus()
7     if(line==0){
8         for each bit in id{
9             set_bus_low()
10            wait_microseconds(10)
11            release_bus()
12            bit=read_bus()
13        }
14    }
15 }

```

Listing 2: SSN Peer Search Algorithm (master)

in the network (Listing 2, lines 4-15). After indicating their presence, slave devices set variables `done` and `end_search` to *false* (Listing 3, lines 2-3). Consider the three situations that a slave device can encounter during a search: *done*, *not waiting*, and *waiting*. A device is *done* if it has successfully sent its ID and is now waiting for other devices to finish sending their IDs. A device is *not waiting* if it is in the process of sending its ID. Finally, a device is *waiting* when it has not yet sent its ID, but is deferring in favor of other devices until the next round.

Before each read round, the master sends a `handshake` pulse. The slave response is determined by its situation. If a slave is *not waiting*, it will try to send its ID (Listing 3, lines 7-37); otherwise, it will check whether all other slaves are *done* (Listing 3, lines 38-47).

If a slave device is *not waiting*, it responds to the `handshake` by pulling the bus low (for a short duration) to indicate (to the master) that it has not been successful in sending its own ID, so the master should read the ID bits now (Listing 3, lines 8-10). On reading the low value, the master starts reading the ID bits (Listing 2, lines 8-13). Both master and slave devices use a counter to keep track of the current bit position. For each bit, the master sends a short synchronization pulse (Listing 2, lines 9-11), and then reads the next bit value from the bus. To send a bit, a device checks its state by checking the value of the `wait` variable (Listing 3, line 12). If the `wait` value is *true*, the device waits for a full bit period (Listing 3, lines 27-29). If the `wait` value is *false*, the device checks the bit value. If it is 0, it pulls the bus low for one bit period and then releases the bus (Listing 3, lines 13-17). If the bit value is 1, the device waits for one-half of a bit period and then observes the bus. If it finds the bus low, it declares itself *waiting* by setting `wait` to *true* for the remaining bits in this round (Listing 3, lines 18-26), thus deferring in favor of another device sending a 0 for this bit.

When all the bits are sent (and read), a slave device that participated throughout the round sets its state to *done*. All *waiting* slave devices then transition to *not waiting* (Listing 3, lines 31-36).

```

1 set_bus_low()
2 end_search=false
3 done=false
4 while(!end_search){
5     wait_for_handshake()
6     wait=false;
7     if(!done){
8         set_bus_low()
9         wait_full_bit_length()
10        release_bus()
11        for each bit in id{
12            if(!wait){
13                if (bit==0){
14                    set_bus_low()
15                    wait_full_bit_length()
16                    release_bus()
17                }
18                else{
19                    wait_half_bit_length()
20                    line=read_bus()
21                    if(line==0){
22                        wait=true
23                    }
24                    wait_half_bit_length()
25                }
26            }
27            else{
28                wait_one_bit_length()
29            }
30        }
31        if(!wait){
32            done=true
33        }
34        else{
35            wait=false
36        }
37    }
38    else{
39        line=read_bus()
40        if(line==0){
41            wait_one_bit_length()
42        }
43        else{
44            done=false
45            end_search=true
46        }
47    }
48 }

```

Listing 3: SSN Peer Search Algorithm (slave)

When a slave device is *done*, it observes the bus transition in response to the **handshake** signal. If the bus is pulled low, the search is not complete, so the device will wait through the remainder of the ID read cycle, synchronizing itself with each bit. If the bus is not pulled low, the search is complete, and the device prepares to accept the next command (Listing 3, lines 38-47).

At the end of one ID read cycle, exactly one device has successfully sent its ID. Since this device is *done*, the number of *done* devices increases in each round, ensuring termination of the search.

4.3 The Smart Surface API

The Smart Surface API is based on a layered architecture and provides access to the physical, data link, and application layers of the SSN bus. The lowest layer of the API handles communication among devices at the physical and data link layers of the SSN bus. The second layer of the API provides protocols for interpreting the bytes into meaningful commands and data, thus providing access to the application layer of the SSN bus network. Developers write programs in the application layer using commands from the second layer of the API. Here we describe each of the API layers.

4.3.1 SSN API, Layer 1

Layer 1 of the API provides access to the physical and data link layers of the SSN bus. The physical layer of the bus is responsible for transmitting and receiving individual bytes between devices. Layer 1 provides support for reading and writing individual bytes on the bus, as well as access to the data link layer of the bus, providing functions for metadata initialization, master election, and CRC checksum calculations.

4.3.2 SSN API, Layer 2

The second layer of the API provides protocols to send commands that can be used in the application layer of the SSN bus. To avoid collisions on the bus, a device must compete with its peers in an election to gain exclusive access to the bus prior to sending any data, as discussed in Section 4.1. A device becomes the master if it wins the election; otherwise, it is considered a slave. The master can send commands to all devices, or it can select an individual device. Slave devices do not send commands to other devices; they send data only when the master requests it. Each device in the SSN can communicate with any other device in the network and is capable of acting both as a master and a slave.

To send a broadcast command (or data), the master first pulls the bus low for a duration longer than the bit period. This pulse causes each slave to start a timer to track how long the bus is low. Upon recognizing a broadcast command, the slaves get ready to accept a command byte from the master. After sending the command byte, the master sends a short pulse on the bus when it is ready to receive the response.

To communicate with an individual device, the master sends the **select** command using a broadcast pulse; the ID of the target device is then sent. If a device recognizes its ID, it watches for subsequent commands. After selecting a device, the master

uses a short pulse to initiate each command (i.e., shorter than the broadcast pulse, but still longer than a bit period). At the end of unicast communication, a master can address all devices by initiating the next command using a broadcast pulse.

The master's broadcast pulse also ensures that newly added devices will not mistake the data and command bit transitions as the start of a command. Each new device first looks for activity. If it detects bus activity, it checks the length of the pulse. If it identifies a broadcast pulse, it starts accepting commands. If it identifies no activity for a specified period of time (chosen by the programmer), the new device may start an election.

4.3.3 Application Layer Programs

The commands provided by the API at the application layer include the functions for peer search, metadata and sensor data collection from peers, packet marshaling, and packet transmission to a PC via UART. To support application-specific node behavior, the SSN API provides the flexibility of adding user-defined commands at the application layer.

The application layer of the SSN comprises multi-command tasks that a smart surface device can perform when it is the elected master. The SSN operating system expects the user to populate two functions. The first function must be populated with registration calls for any user-defined commands (used to link command codes to specific functions). The second function must be populated with the code that a device is expected to execute when it is elected master. The startup routine calls the first function only once during the boot process. The second function is called by the operating system each time a device is elected master. Within this function, a developer can combine application layer API commands to perform customized tasks. The second function should be carefully written because only after exiting this function will a device relinquish the role of master.

4.3.4 Device Metadata

Each SSN device is configured with metadata, including a unique ID and information about the sensors it provides. Currently, this information comprises the number of sensors present on the device, the name of the physical phenomenon that each sensor measures, and the ADC port to which each sensor is attached. The metadata is saved in the microcontroller's EEPROM. In the future, we plan to explore the possibility of extending metadata information to include the units of measure, the supported range of measurement, and the precision of each sensor. This data will be expressed in a format suitable for storage in a resource-constrained device, similar to MoteML [62], yet easy to convert to the SensorML [63] standard for embedded sensor metadata.

5 Evaluation

In evaluating our SSN implementation, we focus on: (i) comparing the efficiency of the SSN peer discovery algorithm with other device discovery algorithms, (ii) evaluating the fairness of the election algorithm, and (iii) observing the effects of varying load conditions and packet length on network throughput.

5.1 Efficiency of Peer Discovery

To determine the efficiency of the SSN peer discovery algorithm, we compare the number of operations needed by the SSN bus with two other device discovery algorithms: the 1-Wire search algorithm, and a brute force search algorithm.

The 1-Wire search algorithm works in the following manner. The master performs two read operations for each bit position in the ID. Given the wired-AND nature of the 1-Wire bus, when more than one device responds at the same time, the bus value represents the logical AND of all the bits sent. The first read operation results in all slave devices sending bit values in their ID at the current bit position, and the second results in all slave devices sending the complement of the same bit. The master then writes the logical AND of the two values on the bus. A slave device reads this bit, and if it matches with its own current ID bit, it continues to participate in the search; otherwise it stops participating and waits for the next round. Hence, there are $3 * p$ operations on the bus for one p -bit address. For n devices, there will be $3 * p * n$ total operations on the bus.

The brute force algorithm works as follows. The master device writes all possible addresses on the bus. Each slave device reads the address, and if it matches its address, it informs the master of its presence. Consider a network with an address size of p bits. The master will always perform $2^p * p$ address bit writes, plus 2^p response read operations, regardless of the number of devices present on the bus.

The SSN search requires p reads by the master for a p -bit address and 1 write and 1 read for handshaking per address by the master. Therefore, with n devices in the network, the algorithm requires only $p * n + 2 * n$ operations, performing better than both brute force and 1-Wire search. The 1-Wire slaves also perform three operations to determine the value of one bit, whereas the SSN slaves perform only one write per bit.

Figure 6 shows a comparison of I/O operations needed by the brute force, 1-Wire, and SSN peer discovery algorithms for networks using 8-bit addresses. The x-axis represents the number of devices in the network, and the y-axis represents the number of I/O operations required to discover all devices in the network.

An interesting observation can be made from Figure 6. In a situation where the number of devices in the network is greater than one-third of the possible addresses, a brute force search will require fewer operations than the 1-Wire search algorithm. For example, with 8-bit addresses, and 100 devices in the network, 1-Wire requires $3 * 8 * 100 = 2400$ operations, whereas a brute force search requires only $(2^8 * 8) + 2^8 = 2304$ operations. The SSN search requires $8 * 100$ reads for ID bits; and $1 * 100$ writes and $1 * 100$ reads for handshaking. Thus, there will be $8 * 100 + 2 * 100 = 1000$ operations.

5.2 Fairness of Election

We next consider the fairness of the SSN election algorithm. To conduct this evaluation, we program devices to continuously compete for the bus. When a device successfully claims the bus and becomes the master, it reports its ID to another device (using serial communication) responsible for sending the ID to a desktop computer. During the election phase, there is a possibility that the devices may miss some bus transitions because of the synchronization issues discussed above. In

such a situation, all the devices reset themselves and attempt the election process again after a short delay. We record the number of resets in our experiments as well.

We start our experiments with a two-device network. The devices compete for the bus for a total of 500 elections, recording the ID of the master for each election. Adding one node at a time, network size is increased from 2 to 9 nodes, repeating the experiments with each increase. The results are shown in Figure 7.

Each bar in Figure 7 represents a network of SSN devices and shows the percentage of times a device was able to gain access to the bus. As an example, the first bar from the left represents a two-device network and reports the percentage of times each of the two devices in the network gained access to the bus, along with the number of resets experienced by the devices. We observe that all devices were represented in the master elections. For a perfectly fair election algorithm, a device should be selected $1/\text{network_size}$ times. We observe that the election algorithm is approximately fair, with each device in the network getting a chance to become master. We also observe that device election is seemingly random, with no noticeable pattern. Next, we consider the relationship between the number of network resets experienced in the SSN and network size. In Figure 7, we observe that the network of three devices suffered the most resets.

We repeat the experiment with the same experimental setup. The results are summarized in Figure 8. We observe that the network of 5 nodes suffered the most resets. By comparing the two experimental result sets, we conclude that master election appears to be approximately random, and each node gets an almost equal chance of getting elected. This confirms fairness. Also, the continuation of network operations, despite resets, shows that the network is robust, able to resume normal operation in the presence of unexpected events. We see fairly large numbers of network resets in our results. We attribute this to the fact that each node in the network was competing to gain access in each election with a probability of 1 — an unlikely occurrence in practical applications.

5.3 Throughput

In this section we evaluate the performance of the SSN design. Specifically, we study the effect of varying load conditions on network throughput.

5.3.1 Throughput, Varying Network Density

We first consider the effect of network density on SSN throughput. We note that the size of the prototype device is 4 square inches and we are restricted to deploying in areas larger than or equal to the device's form factor.

Keeping the number of devices constant in the network, we change network density by changing the physical distance between devices, thus increasing the network's physical area. We define network density as the ratio of the number of devices in the network to the physical area of the network, as shown below. Network area is expressed as multiples of the prototype device's area.

$$\text{Network Density} = \frac{\text{Number of Nodes In Network}}{\text{Network Area}}$$

We conduct this experiment using 6 devices. When devices are placed together in a rectangular grid, with no gaps between them, the total area of the network is 24 sq. inches. As noted previously, we express the network area in multiples of device area. We expand the rectangle by increasing the gap between devices and calculate density using this measure. We vary the density from 1 to 0.25. Each time a device is elected master, it sends 20 packets on the network, and also sends its ID using serial communication to a device that forwards it to an application running on a desktop computer. The desktop application collects this data for a 5-minute interval and records it in a file. We then compute the throughput using this information. We repeat the experiment 50 times and use the results to compute an average. We repeat the experiment for packet sizes of 50, 100, 150, and 200 bytes.

Figure 9 summarizes our results. The X-axis represents network density, and the Y-axis represents network throughput in bits/second. The error bars represent standard deviation from the average. From the results shown in Figure 9, we conclude that network throughput under varying network density does not exhibit substantial change. Hence, we conclude that variation in network density does not noticeably affect SSN throughput.

Polastre et al. [64] introduce the Berkeley Media Access Control (B-MAC) protocol for wireless sensor networks and present a throughput comparison with Sensor Media Access Control (S-MAC) [65]. Our results are comparable. We observe that for networks of size 10 or smaller, B-MAC performs slightly better (11 Kbps for a 6-node network). We also notice that B-MAC throughput is inversely proportional to network size. SSN performs better than S-MAC (7 Kbps for a 6-node network). We note that B-MAC suffers from a large preamble. One example shows a preamble overhead of 271 bytes for a data packet of 36 bytes [64].

5.3.2 Throughput, Varying Network Size

To evaluate the effect of network size on an individual device's throughput, we observe the throughput of a selected device for networks of size 2 to 10. We program all devices to send 20 packets of a given size. We start with a two-device network, choosing one device for throughput calculation, and record the number of packets sent by that device in a 5-minute interval. We divide the total number of bits sent (across all packets) by the number of elapsed seconds to calculate throughput. We repeat the experiment 50 times and use the results to determine average device throughput. For each network size, we repeat the experiment for packet sizes of 50, 100, 150, and 200 bytes.

Our results are summarized in Figure 10. The X-axis represents the number of nodes sending data, and the Y-axis represents throughput of a single node in bits/second. The error bars represent standard deviation from the average. We observe that the throughput of a single device drops rapidly as more devices are added to the network. This behavior is consistent with other media access algorithms, such as carrier sense multiple access with collision detection [66]. We also note that greater throughput values are achieved for larger packets.

Next, we determine the effect of gradually increasing network size on overall network throughput. We record the number of bits transmitted across the network during a 5-minute interval in networks of different sizes. We program all devices

to send 20 packets of a given size. Each time a device is elected master, it sends 20 packets on the network, and also sends its ID using serial communication to a device that forwards it to a desktop computer. We vary network size from 2 to 10. For each network size, we repeat the experiment for packet sizes of 50, 100, 150, and 200 bytes. For each scenario, we repeat the experiment 50 times and use the results to determine the average network throughput.

Our results are presented in Figure 11. The X-axis represents the number of nodes sending data, and the Y-axis represents network throughput in bits/second. The error bars again represent standard deviation. We observe that average network throughput experiences minimal fluctuation over network size. We conclude that available network throughput does not change with the addition of devices in the network, although the individual device throughput decreases.

5.3.3 Throughput, Varying Packet Size

We also evaluate the effect of packet size on SSN device throughput. For this evaluation, we keep the network size fixed at two nodes. We observe the throughput of one device for packet sizes ranging from 20 to 200 bytes. For each packet size, we record the number of packets sent during a 5-minute interval. Throughput is then calculated by dividing the number of bits sent (across all packets) by the number of seconds elapsed to calculate the throughput. We repeat the experiment 50 times using the results to determine the average throughput.

Our results are summarized in Figure 12. The X-axis represents packet size in bytes, the Y-axis represents single node throughput in bits/second, and the error bars represent the standard deviation of values from the average. We see that device throughput increases with increasing packet size, though returns are diminishing.

We repeat this experiment to observe the effect of communication distance on throughput. We place two nodes at a distance of 100 feet. The first node is connected to power supply using short jumper wires. The second node is connected to the same power supply using 100-foot wires. Communication is facilitated by a third wire which is also 100-foot. Our results are summarized in Figure 13. Again, the X-axis represents packet size in bytes, the Y-axis represents single node throughput in bits/second, and the error bars represent the standard deviation of values from the average. We observe that our results are identical to the results presented in Figure 12. Hence, we conclude that SSN behavior is independent of bus length.

6 Conclusion and Future Work

In this paper, we presented the Smart Surface Network (SSN), a solution to address the reliable power distribution and wireless interference problems faced by dense sensing systems, while supporting the unannounced addition and removal of network nodes. The nodes in the SSN communicate using a wired-AND bus. The mountable surface hardware provides power and communication access to all nodes in the network. The SSN software provides algorithms for fair bus access and peer discovery. An API provides access to the physical, data link, and application layers of the SSN. We envision use of the SSN in applications like smart homes, sensate skin, and smart floors.

Our evaluation demonstrates that the election algorithm used for media access is approximately fair. We also show that the SSN peer search algorithm performs better than existing peer discovery algorithms. Finally, we study the performance of the SSN under varying load conditions, showing that SSN bus performance is comparable to other mediums, such as Ethernet.

We envision our future work along three dimensions. The first dimension involves improving the performance of the election algorithm and evaluating the reliability of SSN communication. In our current evaluation, all nodes in the network compete in all elections with a probability of 1, which is an unlikely occurrence in practical applications. In the future, we plan to study the effect of lower probabilities of participation on election algorithm performance. We also plan to study the number of packets received in error to evaluate the reliability of the bus. The second dimension of our future work involves extending the expressivity of our metadata representation to capture more information, and to simplify conversion to existing open standards, such as SensorML. The third dimension of our future work involves the design and implementation of an SSN adapter. The hardware interface will physically connect non-SSN devices to the SSN, and the software interface will provide physical and data link layers for the networking stack of the non-SSN device.

Competing interests

The authors declare that they have no competing interests.

Author's contributions

(1) We present a surface-based networking architecture which provides power and a wired-AND communication bus to attached devices, mitigating battery depletion issues and radio interference problems. (2) We present a reference sensing platform that uses variable length pins to connect to the mountable surface. (3) We present an election algorithm to ensure fair bus access contention resolution. (4) We present a novel peer discovery algorithm to ensure robust network operation in cases of unannounced arrivals and departures of network nodes. (5) We present an API with a complete set of peer-to-peer communication primitives that enable devices to communicate via the SSN bus. (6) We present experimental results characterizing the fairness of the election algorithm, the efficiency of the peer discovery algorithm, and the performance of the SSN system under varying load conditions.

Acknowledgments

This work was supported by the National Science Foundation through awards CNS-0745846 and CNS-1126344. Yvon Feaster is an NSF Graduate Research Fellow, DGE-0751278. ...

Author details

¹ Department of Mathematics and Computing, Lander University, 320 Stanley Avenue, 29649, Greenwood, USA. ² School of Computing, Clemson University, Clemson, SC, 29634, Clemson, USA. ³ Institute for Sensing and Embedded Network Systems Engineering, Florida Atlantic University, 777 Glades Road, 33431, Boca Raton, USA.

References

1. Lifton, J., et al.: Pushpin computing system overview: A platform for distributed embedded, ubiquitous sensor networks. In: Proceedings of International Conference on Pervasive Computing, pp. 139–151. Springer, Berlin, Germany (2002)
2. Lifton, J., et al.: Distributed sensor networks as sensate skin. In: Proceedings of the 2003 International Conference on Sensors, pp. 743–747. IEEE Press, New York, NY, USA (2003)
3. Richardson, B., et al.: Z-Tiles: Building blocks for modular, pressure sensing floorspaces. In: Conference on Human Factors in Computing Systems. CHI '04, pp. 1529–1532. ACM Press, New York, NY, USA (2004)
4. Mistree, B., Paradiso, J.: ChainMail: a configurable multimodal lining to enable sensate surfaces and interactive objects. In: Proceedings of the 4th International Conference on Tangible, Embedded, and Embodied Interaction. TEI '10, pp. 65–72. ACM, New York, NY, USA (2010)
5. Laibowitz, M., et al.: Wearable sensing for dynamic management of dense ubiquitous media. In: Proceedings of the 2009 6th International Workshop on Wearable and Implantable Body Sensor Networks. BSN '09, pp. 3–8. IEEE Computer Society, Washington, DC, USA (2009)
6. Nawaz, R., et al.: Beaconless multihop routing protocol for wireless sensor networks. In: 2011 IEEE 3rd International Conference on Communication Software and Networks, pp. 721–725 (2011)
7. Bencan, G., et al.: Multihop routing protocol with unequal clustering for wireless sensor networks. In: ISECS International Colloquium on Computing, Communication, Control, and Management, 2008. CCCM '08., vol. 2, pp. 552–556 (2008)
8. Zhang, X., Riley, G.: An on-demand bluetooth scatternet formation and routing protocol for wireless sensor networks. In: Sixth International Conference on Software Engineering, Artificial Intelligence, Networking and

- Parallel/Distributed Computing, 2005 and First ACIS International Workshop on Self-Assembling Wireless Networks. SNPD/SAWN 2005, pp. 411–418 (2005)
9. Digi International: ZigBee Standards. <http://www.digi.com/technology/rf-articles/wireless-zigbee>. (date of last access) (2013)
 10. The International Society of Automation: ISA100. <http://www.isa.org/InTechTemplate.cfm>. (date of last access) (2013)
 11. IEEE Standards Association: 802.15.4. <http://standards.ieee.org/about/get/802/802.15.html>
 12. Garcia-Luna-Aceves, J., Tzamaloukas, A.: Receiver-initiated collision avoidance in wireless networks. *Wireless Networks* **8**(2/3), 249–263 (2002)
 13. Golas, A., et al.: Hybrid long-range collision avoidance for crowd simulation. In: *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games. I3D '13*, pp. 29–36. ACM, New York, NY, USA (2013)
 14. Šišlák, D., et al.: Decentralized algorithms for collision avoidance in airspace. In: *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems. AAMAS '08*, pp. 543–550. International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC (2008)
 15. Afek, Y., et al.: Beeping a maximal independent set. In: *Proceedings of the 25th International Conference on Distributed Computing. DISC'11*, pp. 32–50. Springer, Berlin, Heidelberg (2011)
 16. Sen, S., et al.: CSMA/CN: carrier sense multiple access with collision notification. *IEEE/ACM Transactions on Networking* **20**(2), 544–556 (2012)
 17. Chen, T., et al.: A protocol for distributed collision detection. In: *Proceedings of the 9th Annual Workshop on Network and Systems Support for Games. NetGames '10*, pp. 6–166. IEEE Press, Piscataway, NJ, USA (2010)
 18. Kim, K., et al.: Distributed cluster head election algorithm using local energy estimation. In: *Proceedings of the 2009 International Conference on Hybrid Information Technology. ICHIT '09*, pp. 256–259. ACM, New York, NY, USA (2009)
 19. Malpani, N., et al.: Leader election algorithms for mobile ad hoc networks. In: *Proceedings of the 4th International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications. DIALM '00*, pp. 96–103. ACM, New York, NY, USA (2000)
 20. Jurdziński, T., et al.: Efficient algorithms for leader election in radio networks. In: *Proceedings of the 21st Annual Symposium on Principles of Distributed Computing. PODC '02*, pp. 51–57. ACM, New York, NY, USA (2002)
 21. Lynch, N., et al.: Asynchronous leader election and MIS using abstract MAC layer. In: *Proceedings of the 8th International Workshop on Foundations of Mobile Computing. FOMC '12*, pp. 3–1310. ACM, New York, NY, USA (2012)
 22. Yin, Z., Leung, V.: Third-party handshake protocol for efficient peer discovery and route optimization in IEEE 802.15.3 WPANs. *Mobile Networks and Applications* **11**(5), 681–695 (2006)
 23. Ni, J., et al.: Coloring spatial point processes with applications to peer discovery in large wireless networks. *SIGMETRICS Performance Evaluation Review* **38**(1), 167–178 (2010)
 24. Sacha, J., et al.: Using aggregation for adaptive super-peer discovery on the gradient topology. In: *Proceedings of the 2nd IEEE International Conference on Self-Managed Networks, Systems, and Services. SelfMan'06*, pp. 73–86. Springer, Berlin, Heidelberg (2006)
 25. Laerhoven, K., et al.: Pin & play: Networking objects through pins. In: *Proceedings of the 4th International Conference on Ubiquitous Computing*, pp. 219–228. Springer, London, UK, UK (2002)
 26. Laerhoven, K., et al.: Pin & play: The surface as network medium. *IEEE Communications Magazine* **41**(4), 90–95 (2003)
 27. Ljungblad, S., et al.: UbiComp challenges in collaborative scheduling: pin & play at the goteborg film festival. *Personal and Ubiquitous Computing* **11**(7), 563–575 (2007)
 28. Maxim Integrated Products: One wire search algorithm. <http://www.maxim-ic.com/app-notes/index.mvp/id/187>. (date of last access) (2002)
 29. Conner, W., et al.: Making everyday life easier using dense sensor networks. In: *Proceedings of the 3rd International Conference on Ubiquitous Computing*, pp. 49–51. Springer, Berlin, Germany (2001)
 30. National Instruments Corporation: FlexRay Automotive Communication Bus Overview. <http://www.ni.com/white-paper/3352/en>. (date of last access) (2013)
 31. National Instruments Corporation: Local Interconnect Network (LIN). <http://www.ni.com/white-paper/9733/en>. (date of last access) (2013)
 32. Gui, Y.: 1-wire search algorithm and its application. In: *2012 International Conference on Computer Distributed Control and Intelligent Environmental Monitoring (CDCIEM)*, pp. 1–3 (2012)
 33. Xue, H.: Research and development of an intelligent temperature-measuring system based on 1-wire bus. In: *2008 International Conference on Intelligent Computation Technology and Automation (ICICTA)*, vol. 2, pp. 30–33 (2008)
 34. Jie, J.S., Zhiwei, H.: The application of 1-wire bus technology in the temperature monitoring on the of rolling mill. In: *IEEE '09. International Conference on E-Learning, E-Business, Enterprise Information Systems, and E-Government, 2009.*, pp. 233–236 (2009)
 35. Chew, M., et al.: Electrical power monitoring system using thermochron sensor and 1-wire communication protocol. In: *4th IEEE International Symposium on Electronic Design, Test and Applications, 2008. DELTA 2008.*, pp. 549–554 (2008)
 36. Zhao, F., et al.: Design of schematic mapping system based on i2c and usb bus. In: *First International Conference on Instrumentation, Measurement, Computer, Communication and Control, 2011*, pp. 180–183 (2011)
 37. Cai, C.: The LPT data acquisition card design of the front-wheel vehicle positioning test equipment based on the i2c bus. In: *IITAW '08. International Symposium on Intelligent Information Technology Application Workshops, 2008.*, pp. 753–756 (2008)
 38. Righetti, X., Thalmann, D.: Proposition of a modular I2C-based wearable architecture. In: *MELECON 2010*.

- 2010 15th IEEE Mediterranean Electrotechnical Conference, pp. 802–805 (2010)
39. Gong, N., et al.: Leveraging conductive inkjet technology to build a scalable and versatile surface for ubiquitous sensing. In: Proceedings of the 13th International Conference on Ubiquitous Computing. UbiComp '11, pp. 45–54. ACM, New York, NY, USA (2011)
 40. Ran, L., et al.: Design method of CAN BUS network communication structure for electric vehicle. In: 2010 International Forum on Strategic Technology (IFOST), pp. 326–329 (2010)
 41. Ran, P., et al.: The design of communication convertor based on CAN bus. In: ICIT 2008. IEEE International Conference on Industrial Technology, pp. 1–5 (2008)
 42. Lihong, Z., Lei, S., et al.: Measurement and control system of soil moisture of large greenhouse group based on double CAN bus. In: 2011 Third International Conference on Measuring Technology and Mechatronics Automation (ICMTMA), vol. 2, pp. 518–521 (2011)
 43. Guo, H., Jiang, Y.: Application layer definition and analyses of controller area network bus for wire harness assembly machine. In: Computational Intelligence for Modelling, Control and Automation, 2006 and International Conference on Intelligent Agents, Web Technologies and Internet Commerce, International Conference On, pp. 11–11 (2006)
 44. Li, X., Li, M.: An embedded CAN-BUS communication module for measurement and control system. In: 2010 International Conference on E-Product E-Service and E-Entertainment (ICEEE), pp. 1–5 (2010)
 45. Gao, J., et al.: Design of intelligent transmitter for pulp consistency based on the CAN bus technology. In: 2010 IEEE 11th International Conference on Computer-Aided Industrial Design Conceptual Design (CAIDCD), vol. 2, pp. 1213–1216 (2010)
 46. Jaouani, H., et al.: Hybrid task and message scheduling in hard real time distributed systems over flexray bus. In: 2012 International Conference on Communications and Information Technology (ICCIT), pp. 21–26 (2012)
 47. Kim, W., et al.: System-level development and verification of the flexray communication controller model based on systemc. In: Second International Conference On: Future Generation Communication and Networking, 2008. FGCN '08., vol. 2, pp. 124–127 (2008)
 48. Klobedanz, K., et al.: Self-reconfiguration for fault-tolerant flexray networks. In: 2011 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW), pp. 207–216 (2011)
 49. K. Lakshmanan, K., et al.: Integrated end-to-end timing analysis of networked AUTOSAR-compliant systems. In: Design, Automation Test in Europe Conference, pp. 331–334 (2010)
 50. Senthilkumar, K., Ramadoss, R.: Designing multicore ECU architecture in vehicle networks using AUTOSAR. In: 2011 Third International Conference on Advanced Computing (ICoAC), pp. 270–275 (2011)
 51. Springbok Digitronics: 1-Wire Design Guide v1.0. <http://www.1wire.org>. (date of last access) (2004)
 52. Wikipedia: Bus Shunt Method. http://en.wikipedia.org/wiki/Local_Interconnect_Network#Bus_Shunt_Method_.28BSM.29. (date of last access) (2013)
 53. Less EMF Inc.: Stretch Conductive Fabric. <http://www.lessemf.com/fabric.html>. (date of last access) (2011)
 54. Ellsworth Adhesives: Epoxy. <http://www.ellsworth.com/Search/epoxy>. (date of last access) (2013)
 55. Futurlec: Datasheet, DS1820. <http://www.futurlec.com/Dallas/DS1820.shtml>. (date of last access) (2011)
 56. Crossbow Tech Inc.: MICA2 Wireless Measurement System. <http://www.eol.ucar.edu/isf/facilities/isa/internal/CrossBow/DataSheets/mica2.pdf>. (date of last access) (2013)
 57. Crossbow Tech Inc.: MICAz Wireless Measurement System. http://www.openautomation.net/uploads/productos/micaz_datasheet.pdf. (date of last access) (2013)
 58. Eidson, G., et al.: The South Carolina digital watershed: End to end support for real time management of water resources. International Journal of Distributed Sensor Networks 2010 (2010)
 59. HIROSE Electric Co.: HIROSE Connectors. <http://www.hirose-connectors.com/>. (date of last access) (2014)
 60. Atmel Atmega: ATmega168. <http://www.atmel.com/Images/doc2545.pdf>. (date of last access) (2013)
 61. Atmel: Manchester Encoding Basics. <http://www.atmel.com/Images/doc9164.pdf>. (date of last access) (2014)
 62. Ali, F., et al.: A metadata encoding for memory-constrained devices. In: Proceedings of the 49th Annual Southeast Regional Conference. ACM-SE '11, pp. 191–196. ACM, New York, NY, USA (2011)
 63. Open Geospatial Consortium: SensorML. <http://www.opengeospatial.org/standards/sensorml>. (date of last access) (2013)
 64. J. Polastre, J., et al.: Versatile low power media access for wireless sensor networks. In: Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems. SenSys '04, pp. 95–107. ACM, New York, NY, USA (2004)
 65. Ye, W., et al.: An energy-efficient mac protocol for wireless sensor networks. In: Proceedings of the 21st International Annual Joint Conference of the IEEE Computer and Communications Societies. INFOCOM '02 (2002)
 66. Tobagi, F., Hunt, V.: Performance analysis of carrier sense multiple access with collision detection. Computer Networks 4, 245–259 (1980)

Figures

Figure 1: Surface Substrate

Figure 2: Testing Devices

Figure 3: Inserted Device Diagram

(a) Pin Layer

(b) Controller Layer

(c) Sensor Layer

(d) Layer Stack

Figure 4: SSN Device Layers

Figure 7: Master Election, Experiment 1

Figure 8: Master Election, Experiment 2

Figure 9: Network Throughput vs Network Density

Figure 10: Single Node Throughput vs Network Size

Figure 11: Network Throughput vs Network Size

Figure 12: Single Node Throughput vs Packet Size

Figure 13: Single Node Throughput vs Packet Size
(100-foot Linear Distance)