MIDDLEWARE SERVICES FOR STRUCTURAL HEALTH MONITORING USING SMART SENSORS

T. Nagayama,[1] B. F. Spencer, Jr.,[2] K. A. Mechitov,[3]

and G. A. Agha[3]

[1)]Department of Civil Engineering, University of Tokyo
7-3-1, Hongo, Bunkyo-ku, Tokyo 113-8656, Japan

[2)]Department of Civil Engineering, University of Illinois at Urbana-Champaign
205 N. Mathews Ave. Urbana, Illinois 61801, USA

[3)]Department of Computer Science, University of Illinois at Urbana-Champaign
201 N. Goodwin Ave. Urbana, Illinois 61801, USA

ABSTRACT: Smart sensors densely distributed over structures can use their computational and wireless communication capabilities to provide rich information for structural health monitoring (SHM). Though smart sensor technology has seen substantial advances during recent years, implementation of smart sensors on full-scale structures has been limited. While users often postulate that wired sensing systems can be simply replaced by wireless systems, off-the-shelf wireless systems are unlikely to provide the data users expect. Sensor component characteristics limit the quality of data collected; data from smart sensors may be inadequate due to packet loss during communication, time synchronization errors, and slow communication speeds. This paper addresses these issues common to smart sensor applications for structural health monitoring by developing corresponding middleware services, i.e., reliable communication, synchronized sensing, and model-based data aggregation. These middleware services are implemented on the Imote2 smart sensor platform, and their efficacy demonstrated experimentally.

1. INTRODUCTION

Dense arrays of smart sensors have the potential to improve Structural Health Monitoring (SHM) dramatically using onboard computational and wireless communication capabilities. These sensors provide rich information sources, which SHM algorithms can utilize to detect, locate, and assess structural damage produced by severe loading events and by progressive environmental deterioration. Information from densely instrumented structures is expected to result in the deeper insight into the physical state of a structural system.

Though smart sensors have seen several demonstrative applications (Lynch and Loh 2006, Kim et al 2007), detailed validation of the data from the smart sensor networks and subsequent use for damage detection has been limited. Measured signals are oftentimes lost during wireless communication or are not synchronized with each other, limiting subsequent analyses. The amount of data to be collected is extremely large, prohibiting spatially dense measurements and/

or the frequency of operation (Nagayama et al. 2007a). The lack of these functionalities has prevented SHM application users from analyzing structures in detail based on smart sensor measurement data.

This paper addresses these issues by developing corresponding middleware services that are common to many SHM applications. Among the middleware services are reliable communication, synchronized sensing, and data aggregation. These middleware services are developed employing a smart sensor platform, the Imote2, which is designed for data-intensive applications such as SHM applications. The middleware services are then implemented on the Imote2 running TinyOS. The idea behind these services are generally applicable to other smart sensor platforms, and the middleware service can be ported to platforms which have similar hardware characteristics, e.g. RAM size, as the Imote2. The implemented middleware services are evaluated and validated experimentally.

2. BACKGROUND

2.1 Reliable data transfer

RF communication is not as reliable as wired communication, due to packet loss. The most common cause for packet loss results from the fact that as distance between nodes increases, the signal-to-noise ratio drops, causing bit errors in the transmitted packet. When the receiver detects a bit error, it drops the entire packet. Interference between multiple simultaneously transmitting nodes, called a collision, can likewise result in the loss of one or both packets. These communication failures may take place for both packets carrying commands to sensor nodes and packets carrying measured data. If packets carrying commands are lost, destination nodes fail to perform certain tasks. When smart sensors are designed to perform complex sets of tasks, collaborating with other sensors nodes, command packet loss may cause sensor nodes to behave unexpectedly, possibly leading to network failure. If packets carrying measurement data are lost, destination nodes cannot fully reconstruct the sender's data. SHM applications employing smart sensors must address this packet loss problem. Retransmission- and acknowledgment-based approaches are candidates to reduce the packet loss rate during wireless communication.These two approaches are first explained briefly.

retransmission without acknowledgment can statistically improve the reliability of communication. If the packet loss rate is expected to be approximately constant over time, retransmission can virtually eliminate data loss; the number of repetitions can be dynamically adjusted based on measured packet loss rates. When the packet loss rate is high, the number of repetitions is increased. Such protocols, however, cannot guarantee communication success rate deterministically. In smart sensor networks, burst packet loss may take place, which undermines the effectiveness of this approach. Interference from other nearby RF transmission devices operating in the same frequency range, for example, may cause a large number of packets to be dropped. If the packet loss rate is high during all the successive retransmissions, the loss of many packets is unavoidable.

Acknowledgment-based approaches can potentially guarantee reliable communication between nodes, by continuing packet retransmission until acknowledgment is received. However, a poorly-

designed communication protocol involving acknowledgment messages can be notably inefficient. Many acknowledgment messages may be required, waiting times may be long, and the same packets may need to be sent many times. Furthermore, RF components on many smart sensor platforms including the Imote2 are in either a listening mode or transmission mode, making the need for frequent switch between the two modes unwelcome. During transmission, the Imote2 cannot receive packets. Nonetheless, transmission and reception are deeply interwoven in many acknowledgement-based approaches. Interwoven transmission and reception of acknowledgement-based approaches may result in slow data transfer.

Instead of acknowledging each packet, the reliable communication protocol developed by Mechitov et al. (2004) sends a set of packets and then waits for acknowledgment. If the receiver does not receive acknowledgment, the same set of packets is sent again. Upon reception of acknowledgment, the sender moves on to the next set of packets. The size of a set of packets needs to be optimized. A small size of packet set necessitates a larger number of acknowledgement packets; a large size of packet set, on the other hand, results in retransmitting large numbers of packets even when only one packet is lost. Therefore, the number of acknowledgement packets cannot be drastically reduced.

2.2 Synchronized sensing

The lack of a global clock in smart sensor network is problematic for SHM applications. If modal analysis is conducted without accurate time synchronization, the identified mode shape phase will be inaccurate, possibly falsely indicating structural damage. Large synchronization errors make accurate estimation of cross spectral densities and transfer functions almost impossible.

Time synchronization for smart sensor networks has been widely investigated. By communicating with surrounding nodes, smart sensors can assess relative differences among their local clocks. Reference Broadcast Synchronization (RBS; Elson et al., 2002), Flooding Time Synchronization Protocol (FTSP; Maroti et al., 2004) and Timing-sync Protocol for Sensor Networks (TPSN; Ganeriwal et al., 2003) are among the well-known synchronization methods. Mica2 motes employing TPSN are reported to synchronize with each other to an accuracy of 50 μsec. Mechitov et al. (2004) implemented FTSP on Mica2 motes as a part of wireless data acquisition system for SHM. This system can maintain better than 1ms synchronization accuracy for minutes. Clock rate difference(i.e., drift) from node to node increases time synchronization error over time. When clock drift is significant, synchronization is applied periodically before the error becomes too large.Thus, fine synchronization among sensor nodes has been shown achievable.

However, measured signals may not be synchronized with each other, even when clocks are precisely synchronized. Sampling timing is not necessarily controlled precisely based on these clocks. Whereas time synchronization protocols have been intensively studied, synchronized sensing needs further investigation.

2.3 Data aggregation

The amount of data typically collected by SHM applications employing wired sensors normally exceeds practical communication capabilities of smart sensor networks. In such approaches, data needs to be acquired at an appropriate sampling rate for a sufficient period of time at various

locations and stored at a central location. Central collection of vibration measurement data is reported to take more than ten hours (Kim et al. 2007). One approach to overcome this problem has been to use an independent data processing strategy (i.e., without internode communication). This approach, however, cannot fully exploit information available in the sensor network (e.g., much of spatial information is neglected). Data aggregation is an important issue to be addressed before an SHM system employing smart sensors can be realized.

Distribution of data processing and coordination among smart sensors play a central role in addressing many smart sensor implementation issues, including data aggregation. Data processing is application dependent. Application specific knowledge can be incorporated into data aggregation strategies to efficiently collect information from smart sensor network.

2.4 Intel Imote2 smart sensor platform

The Imote2 is a new smart sensor platform developed for data-intensive applications (Crossbow, Inc., 2008). The main board of the Imote2 incorporates a low-power Xscale processor, the PXA271, and an 802.15.4 radio, ChipCon 2420. The processor speed may be scaled based on the application demands, thereby improving its power efficiency. One of the important characteristics of the Imote2, which separates it from other commercially available wireless sensing nodes, is the memory size. The Imote2 has 256 KB of integrated SRAM, 32 MB of external SDRAM, and 32 MB of Strataflash memory, which is particularly important for the large amount of data required for dynamic monitoring of structures and associated data processing. The computational power and storage capacity make the Imote2 stand unequaled among commercially available smart sensors. Intel has created a basic sensor board to interface with the Imote2. This basic sensor board can measure 3-axes of acceleration, light, temperature, and relative humidity. All of the sensors on this board are digital, thus no analog to digital converter (ADC) is required. The 3-axis digital accelerometer (LIS3L02DQ; STMicroelectronics, 2008) has a ±2g measurement range and a resolution of 12-bits or 0.97 mg. The Imote2 with the basic sensor board are employed in this paper as the smart sensor platform.

3. RELIABLE COMMUNICATION

While reliable command packet transfer is clearly a significant help in developing SHM systems requiring complex, internode, data processing, the need for reliable data transfer requires further consideration. Nagayama et al. (2007a) theoretically derived expressions for the effect of data loss on power spectral density estimates and showed that data loss degrades measurement signals similar to that of observation noise. When packet loss is not negligibly small, data needs to be reliable transferred.

To estimate the packet loss rate in smart sensor communication and assess the need for reliable communication middleware services, experiments are first conducted. Then reliable communication protocols are proposed to transfer a large amount of data as well, as protocols to send a single packet.

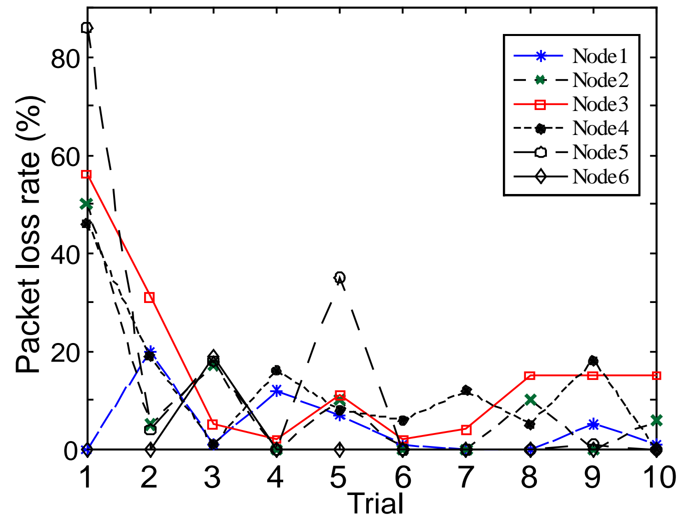

Figure 1. Packet loss rate estimation.

3.1 Packet loss estimation in RF communication

First, packet loss rate is experimentally estimated. A node sends a large number of packets to the receiver nodes and packets received by the receivers are counted; this procedure is repeated several times to estimate the packet loss rate. Eight Imote2s, one programming board, and a PC were configured to perform this experiment. One of the Imote2s is programmed as the base station. Another Imote2 works as the sender. The other six Imote2s are configured to be receivers. A java program running on the PC reads parameters from an input file and sends commands to the base station node. The base station receives information about the number of receivers, node IDs of the sender and receivers, the number of packets to be sent, and the number of repetitions. The base station forwards this information to the sender. On reception, the sender starts broadcasting packets. In this experiment, 100 packets are broadcast. After the sender completes transmission of the 100 packets, the sender tells the receiver that transmission is complete and queries each receiver regarding how many packets were received. This procedure is repeated 10 times.

This experiment is conducted on the lawn in front of Newmark Civil Engineering Laboratory at the University of Illinois at Urbana-Champaign. The sender and receiver nodes are held at the height of about 1 m. The distance between the sender and the receiver is 3.3 m. Figure 1 summarizes the results. While many rounds of data transfer were achieved without any lost packets, the maximum packet loss rate reached 86 percent. In many cases, the packet loss is smaller than 20 percent. The packet loss rate does not show a clear trend among the six nodes. One node without any packet loss in one round of data transfer may suffer from severe data loss in the subsequent round of communication. Woo et al. (2003) also reported that a large packet loss rate is expected, especially when communication distance becomes large. While a loss of only a few packets is likely to take place in communication, such a low packet loss rate cannot be guaranteed.

These levels of packet loss are much larger than those discussed in Nagayama et al.(2007a) and is not acceptable for SHM applications. A reliable communication protocol suitable for transfer of a large amount of data, as well as a protocol for command packets, is proposed herein.

3.2 Reliable communication protocol

Reliable communication protocols for long data records and for commands are developed in this section. In SHM applications, most communication takes place as unicast. Nonetheless, multicast of command packets is oftentimes needed. Some SHM applications can efficiently achieve data aggregation, taking advantage of multicast of long data records as explained later. Both unicast and multicast protocols are, therefore, supported in the proposed protocols.

3.2.1 Communication protocol for long data records

Communication protocols suitable for transfer of long data records reliably are developed based on acknowledgement approaches. In the proposed protocols, the sender transmits all of the data packets before acknowledgment for these packets is expected. The receiver stores all of the received data in a buffer. Once the receiver gets a message indicating the end of data transmission, the receiver replies to the sender, indicating which packets are missing. Then, only missing packets are resent. In this way, the number of acknowledgments and retransmissions, as well as the need to switch between the two modes, can be greatly reduced.

Both the unicast and multicast protocols are designed to send either 64-bit double precision data, 32-bit integers, or 16-bit integers. Many ADCs on traditional data acquisition systems have a resolution less than 16 bits, supporting the need for transfer of 16-bit integer format data. Some ADCs have a resolution better than 16 bits, necessitating data transfer in 32-bit integer format. Once an acceleration record is processed, the outcome may need more precision. Onboard data processing such as FFTs and SVDs is usually performed using double precision calculations. Even when the effective number of bits is smaller than 32, debugging of onboard data processing greatly benefits from transfer of double precision data; data processing results on Imote2s can be directly compared with those on a PC, which are most likely in double precision format. Inclusion of 64-bit double precision data transfer is based on such needs.

The maximum data length supported by the protocols is set as 11,264 for 16-bit integer data, though this length can be adjusted by changing parameters. SHM usually involves FFT calculations; 11,264 is 11 times 1,024;1024 is often employed as the length of the FFT. As explained later, the longer the maximum supported data length is, the larger the RAM needed for the protocol is. Therefore the supported data length cannot be arbitrarily increased. 11,264 data points in 16-bit integer format needs about 22.5 kB of RAM while the Imote2 RAM size is 256 kB. When an extremely large amount of data needs to be transferred, data can be split into smaller blocks and sent block by block.

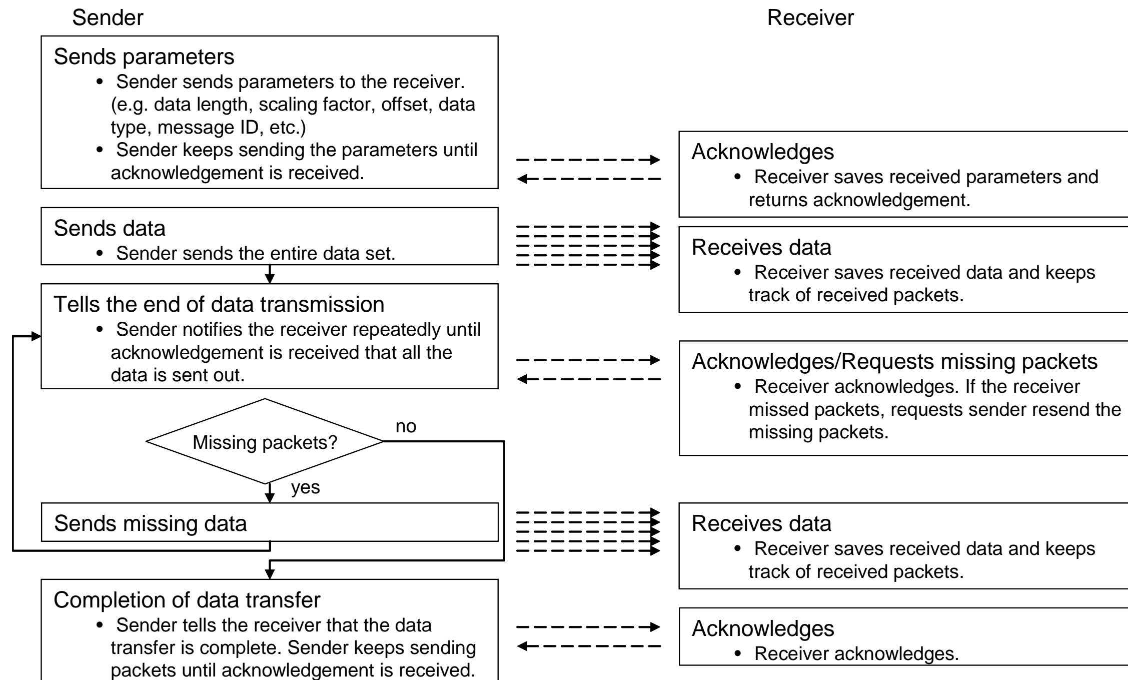

Figure 2. Block diagram of the reliable unicast protocol for long data records.

Unicast

In the unicast protocol, the sender reliably delivers long data records to a single receiver node. Figure 2 shows a flowchart of this protocol. The sender first performs a handshake and transfers the necessary parameters. Then all the data is sent without requesting acknowledgements. After this transfer, the receiver checks for missing packets and requests retransmission. Following retransmission, the handshake is terminated. The details of the protocol are explained in the following paragraphs.

On main program's start-up, the communication middleware service requests that the main program assign a buffer space to keep data to be transferred. This buffer needs to be large enough to hold the entire data record. The main program returns the pointer to this buffer to the middleware service. The need for this large buffer size is a drawback of this approach. However, the buffer is allocated in the main program and only the pointer is given to the communication program; with knowledge of the application, the main program can utilize this memory space for other purposes when communication is not taking place.

When the application is ready to send data, the sender sends a packet and informs the receiver of the necessary parameters (e.g., data length, data type, message ID, destination node ID, source node ID, etc.). If the data is in integer format, a scaling factor and offset of the data are also conveyed to the receiver so that the receiver can reconstruct the original data. All these parameters are packed in the 28 B payload of a packet. After this transmission, the sender periodically checks whether an acknowledgment from the receiver has arrived. If not, this packet is sent again. Once two nodes exchange this packet and engage in a round of reliable communication, packets with different message IDs, destination node IDs, and source node IDs

are disregarded. The message ID of the sender is incremented when the sender engages in a new round of reliable communication.

For Imote2 nodes to appropriately interpret the received packet, 1 byte of the payload of every packet is designated for packet interpretation instruction. For example, a packet from the sender with the instruction '1' is interpreted as the first packet with parameters such as data length, data type, etc. On reception of a packet, tasks predetermined for the instruction are executed.

Also, the current state of the sender and receiver are internally maintained using a 1-byte variable on each node. For example, the sender's current state is '1' after sending the packet with the instruction '1'. This state changes to '2' when the sender finds that an acknowledgment for this packet is received. The tasks to be executed in events such as packet reception and timer firing are determined based on the current state variable and the instruction byte of the most recently received packet.

Upon reception of a packet with the instruction of '1' from the sender, the receiver checks whether it is ready to receive data. Unless this node has already executed the handshake with another node (i.e., agreed to receive data from another node or send data to another node), this node replies to the sender with an acknowledgment packet. The two nodes are then ready to transfer data.

When the sender notices that the acknowledgment has arrived, the periodical check for acknowledgment packets stops and the entire data record is sent to the receiver. One packet contains either three double precision data points, six 32-bit integers, or twelve 16-bit integers. Each packet also contains a 2-byte node ID for the sender and a 2-byte packet ID, which the receiver uses to sort received packets and reconstruct the original data in the buffer. To keep track of received packets, the receiver maintains a bitmap of received packets, which is initialized to zeroes on reception of a packet with the instruction '1'. One bit of the bitmap is assigned to each data packet, which necessitates allocation of 235B RAM space to keep track of 11264 data points in integer format. Upon reception of each packet, the corresponding bit is changed to one. By examining this bitmap, the receiver determines which packets are missing.

When the sender finishes transmitting all the data, a packet conveying the end of the data transmission is sent to the receiver. An instruction-byte of '2' for this packet is interpreted as the end of the data transmission. The sender periodically checks the response from the receiver. If nothing is received, this packet is sent again.

Knowing that all of the data was transmitted from the sender, the receiver checks the bitmap to keep track of received packets. If all of the packets were received, an acknowledgment is returned to the sender. The sender then tells the receiver that the data transfer is complete by sending a packet with the instruction-byte of '4'. This message is also resent until an acknowledgment packet is returned from the receiver. When the receiver first acknowledges the packet with the instruction-byte of '4', the receiver signals to the main program on the receiver that a set of data has been successfully received, thus completing the receiving activity on this node. Once the sender node receives this acknowledgment, the main application on the sender is notified that the data has been sent out successfully, thus completing the sending activity. The handshake is called off and the two nodes are ready to start another round of data transfer.

If the receiver notices that some packets are missing on reception of the packet with the instruction-byte of ‘2’, the receiver responds by sending a packet with the missing packets’ IDs. The first eight missing packets’ IDs are packed in the payload of an acknowledgment packet and sent to the sender. The sender packs these missing packets and sends them to the receiver. At the end of transmission, the packet with the instruction-byte of ‘2’ is sent again. If there are still missing packets, the same procedure is repeated. If all of the data is received, an acknowledgment is returned, and the communication activity is completed as described in the previous paragraph.

One of the difficulties with this protocol is judging when communication ends This difficulty is also known as the Byzantine Generals Problem (Lamport et al. 1982). The sender can clearly determine the end of communication when the acknowledgment message with the instruction-byte of ‘4’ is received. However, the receiver cannot judge the end of communication deterministically. The receiver does not know whether the acknowledgment with the instruction-byte of ‘4’ reaches the sender or not. If the receiver node does not receive any packet after the transmission of this acknowledgment packet, there are two possible situations: the acknowledgment packet reached the sender and this data transfer was successfully completed; and the acknowledgment packet did not reach the sender and the sender is periodically transmitting packets with the instruction-byte of ‘4’, none of which arrived at the receiver node. There is no way for the receiver node to judge between the two possibilities.

This inability to judge the end of communication may cause a serious problem. If only the receiver calls off the handshake, then the sender will keep sending the last packet to the receiver; the receiver never replies because the receiver is already disengaged from this round of data transfer.

This problem is alleviated by storing message and source IDs of the last several rounds of reliable communication and separating the last acknowledgment process from the rest of the process. When the receiver gets the packet with the instruction-byte of ‘4’ for the first time, this receiver node sends an acknowledgment back and is disengaged from this communication activity. The message and source IDs are stored on this node. If this receiver node later receives the same packet, the structure storing the message and source IDs is checked. Once the receiver finds in the structure the pair of the message and source IDs that are the same as those of the received packet, an acknowledgment is sent back to the sender. This procedure to acknowledge the end of data transfer may take place while the receiver node is engaged in another round of data transfer, giving concerns over overwriting packet buffer. For example, consider a situation where a node is engaged in a round of data transfer and about to send a packet with the instruction-byte of ‘2’. If this node receives a packet with the instruction-byte of ‘4’ requesting acknowledgment from another node, this node fills the packet buffer with the necessary parameters for acknowledgement (e.g., the instruction-byte of ‘4’). If the packet buffer is shared for the two processes, the buffer is overwritten. To avoid such overwriting, the packet buffer space for this acknowledgment with the instruction-byte of ‘4’ is assigned separately from the packet buffer space for others. Even when the receiver is engaged in the next round of communication, this process of acknowledgment can be done without affecting the subsequent or ongoing communication activities.

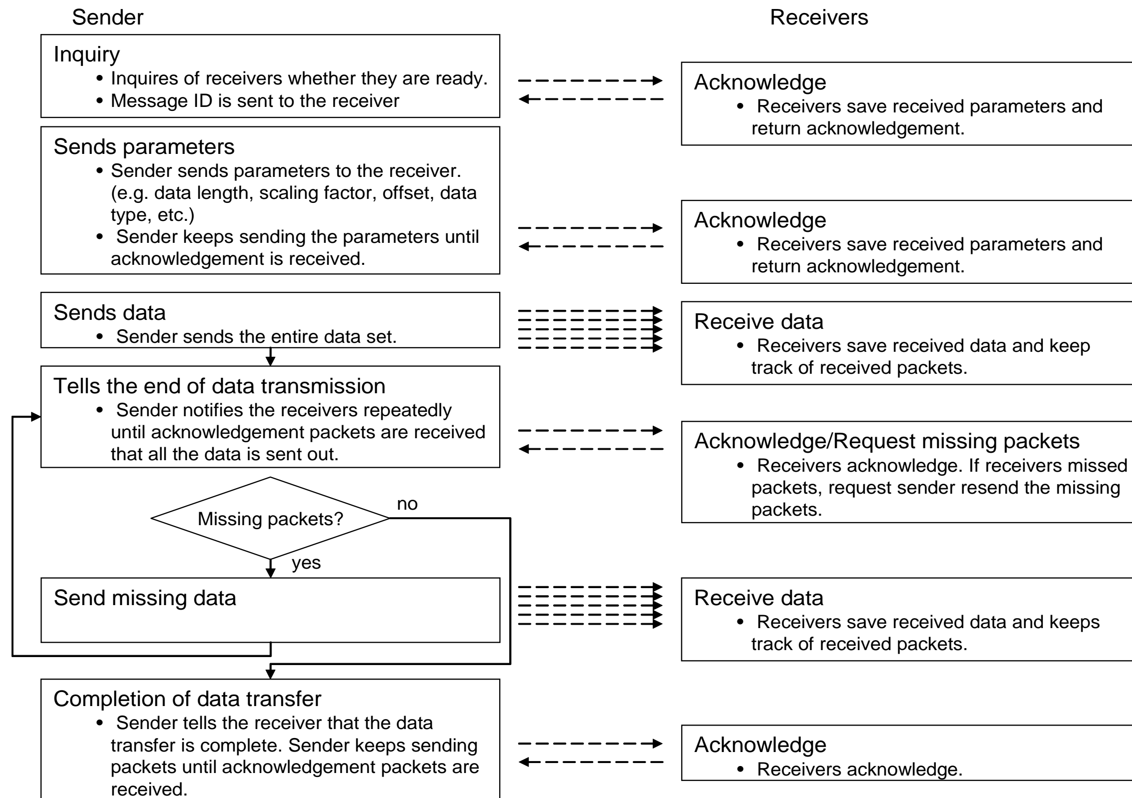

Figure 3. Block diagram of the reliable multicast protocol for long data records.

Multicast

In the multicast communication protocol, the sender reliably transmits long data records to multiple receivers utilizing acknowledgment packets. Figure 3 shows a flowchart of this protocol. At the beginning of communication, the sender searches for receiver nodes. Then handshaking, data transfer, and retransmission follow before calling off the handshake, as in the unicast communication protocol. The details of this protocol are explained in the following paragraphs.

The primary difference between the multicast and unicast protocols is the first step, searching for the receivers. Before the sender sends the required parameters, such as data length and data type, the node IDs of the receivers are broadcast, as well as the source and message IDs. The instruction byte of this packet is set to ‘f’ in hexadecimal. A maximum of eight receiver node IDs are sent in a single packet. When data is sent to more than eight nodes, multiple packets containing the receiver node IDs are broadcast. As is the case for unicast communication, the sender periodically sends these packets until the receivers reply.

Upon reception of this initial packet, receivers compare their own node IDs and the those contained in the packets. If a node is not one of the destination nodes, this node simply disregards the packet. If the node is one of destination nodes, this node sends an acknowledgment to the sender and commits itself to this round of communication.

Note that if multiple receivers reply to the sender at the same time, packet collision may take place; timing for replies to the sender needs to be scheduled appropriately. The order of node IDs in the packets with the instruction-byte of ‘f’ is used to schedule the timing for replies. For example, the node corresponding to the eighth node ID in the packet with the ‘f’ instruction-byte waits 7 x β seconds before the acknowledgment is sent back to the sender; β is unit waiting time. The order of reply and β are stored on each receiver; all of the following acknowledgment messages in this multicast protocol use this scheduling. Optimization of the waiting interval, β, may result in faster communication; such optimization has not been pursued yet.

Once the sender receives an acknowledgment from all of the receivers, the parameters such as data length, data type, scaling factor, and offset are broadcast. Although the broadcast transmission reaches all of the nodes in the neighborhood, only the engaged nodes process the received packet. The source ID, the destination ID (i.e., node ID of ‘ffff’ in hexadecimal, which corresponds to broadcast address in TinyOS), and the message ID are used on receivers to distinguish packets to be processed from other packets.

Another difference between unicast and multicast is that the sender needs to check the acknowledgment messages from all of the receivers. The sender keeps track of acknowledgments from the receivers using a bitmap. One bit of the bitmap corresponds to one receiver. Before the sender transmits a packet needing acknowledgement from the receivers, this bitmap is initialized to zeroes. Upon reception of an acknowledgment from a receiver, the bit corresponding to this receiver is changed to one. While the size of this bitmap currently is set so that up to 32 nodes can be receivers, parameters can be easily adjusted to accommodate a larger number of receivers at the expense of the memory required for storing the bitmap.

Another point to consider is the way that requests for retransmission are handled. After sending the end-of-transmission notice packet to the receivers, the sender waits for a reply from the respective receivers. Within the assigned time slot, each receiver sends back an acknowledgment packet containing missing packet IDs. After the time assigned for all of the nodes to reply has passed, the sender broadcasts the requested packets. At the end of broadcast of these requested packets, a packet with the instruction-byte of ‘2’ is sent, asking for packets still missing.

The end of one round of communication is achieved in a similar manner to that implemented for unicast reliable communication. In addition to the message and source IDs, this multicast protocol requires the receiver nodes store the waiting time before replying with acknowledgment.

3.2.2 Communication protocol for commands

Communication protocols suitable to transfer a single packet reliably are also needed. SHM applications involve sending and receiving many commands, each of which fits in one packet. These commands need to be delivered reliably. If a packet containing a command to start sensing is lost, the destination node does not start sensing. The sender node cannot deteministically judge if a certain command was delivered and executed at the receiver nodes. The current states of smart sensor nodes are hard to estimate without reliable communication. Ensuring the performance of SHM systems without reliable command delivery is extremely complex if not impossible. The reliable communication protocol developed for long data records is, however, not efficient for

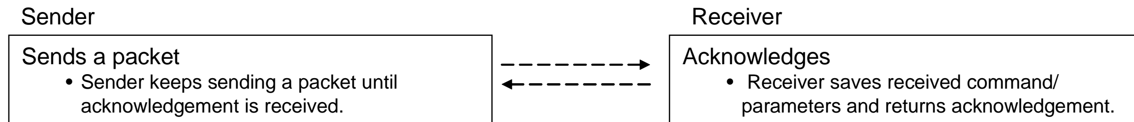

Figure 4. Block diagram of the reliable unicast protocol for commands.

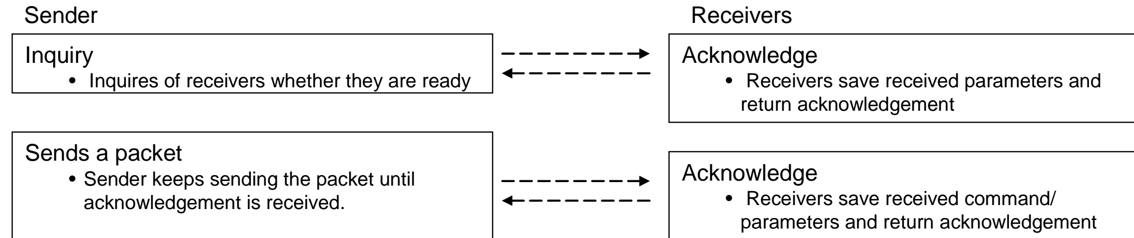

Figure 5. Block diagram of the reliable multicast protocol for commands.

single packet transfer. Unicast and multicast reliable communication protocols suitable for single-packet messages are developed.

This protocol is again similar to the ARQ protocol. Because only one packet is sent, an acknowledgment is returned to the sender for each packet. The protocol is designed to send commands and parameters in 8- and 16-bit integers. The payload of one packet can hold nine 16-bit integers and one 8-bit integer in addition to communication parameters such as source and message IDs.

Unicast

The sender transmits a packet, including the destination ID, source ID, message ID, command, and other parameters. Until this packet is acknowledged, the sender continues to transmit this packet. The receiver extracts information from the packet and sends an acknowledgment back to the sender (see Figures 4).

The end of one round of communication is implemented in a manner similar to that for the reliable unicast protocol for long data records. The source and message IDs for the last several rounds of communication are stored so that acknowledgment can be sent even after the receiver is disengaged from this round of communication.

Multicast

The multicast protocol is implemented in a similar manner to the unicast protocol. As is the case for protocols for long data records, the primary difference is the first step, searching for the receivers. The sender first transmits a packet including the receivers' node IDs, as was the case for the multicast communication protocol for long data records. Once all of the receivers return acknowledgment packets to the sender, the sender transmits a packet containing a command and associated parameters. The receivers also acknowledge this packet. To judge the end of communication, the receiver keeps the source ID, message ID, and waiting time of the last several rounds of communication (see Figures 5).

4. SYNCHRONIZED SENSING

Measured signals from a smart sensor network with the intrinsic local time differences need to be synchronized. If not appropriately addressed, time synchronization errors can cause inaccuracy in SHM applications, particularly in the mode shape phases (Nagayama 2007). Time synchronization accuracy realized on the Imote2 is first estimated and evaluated for the SHM applications. Time synchronization among smart sensors does not necessarily offer synchronized measurement signals. Even when the clocks of sensor nodes are perfectly synchronized with each other, measured signals may not be accurately time-aligned. Issues critical to synchronized sensing are then investigated and synchronized sensing is realized utilizing a resampling approach (Nagayama 2007; Nagayama et al. 2006a, 2007a, & 2007b; Spencer and Nagayama, 2006).

4.1 Estimation of time synchronization error

The accuracy of the Flooding Time Synchronization Protocol (FTSP; Maroti et al., 2004) implementation on the Imote2 is evaluated. This protocol is first reviewed briefly, and then the accuracy is examined in terms of synchronization errors and clock drift.

FTSP utilizes time stamps on the sender and receivers. A beacon node broadcasts a packet to the other nodes. At the end of the packet, a time stamp, t_{send}, is appended just before transmission. Upon reception of the packet, the receivers stamp the time, $t_{receive}$, using their own local clocks. The delivery time, $t_{delivery}$, between these two events includes interrupt handling time, encoding time, and decoding time. $t_{delivery}$ is usually not small enough to be ignored; the variance of $t_{delivery}$ over time is usually small. $t_{delivery}$ can be estimated in several ways. An oscilloscope connected to both nodes and on-board clocks can keep track of the communication time stamp. In this study, $t_{delivery}$ is first assumed to be zero and then adjusted so that Imote2s placed on the same shake table give synchronized acceleration data. If these nodes are synchronized, the phases of the transfer functions among Imote2 signals should be constant over wide frequency range. $t_{delivery}$ is determined to give a constant phase of zero. Using this value, the offset between the local clock on the receiver and the reference clock on the sender is determined as $t_{receive} - t_{send} - t_{delivery}$. This offset is subtracted from the local time when global time stamps are requested afterward.

To evaluate time synchronization error, a group of nine Imote2s are programmed as follows. The beacon node transmits a beacon signal every 4 seconds. The other eight nodes estimate the global time using the beacon packet. Time synchronization is thus performed. Two seconds after the beacon signal, the beacon node sends the second packet, requesting replies. The receivers get time stamps on reception of this packet and convert them to a global time stamp using the offset estimated 2 seconds before. These time stamps are subject to two error sources: First, time synchronization error, and second, delay in time stamping upon reception of the second packet. The receivers take turns to report back these time stamps. This procedure is repeated more than 300 times. These time stamps from the eight nodes are compared. Figure 6 shows the difference in the reported global time stamps using one of the eight nodes as a reference. The time synchronization error seems to be less than 10 μs for most of the time. Scattered peaks may indicate large synchronization error. Note that the time synchronization error is one of the two

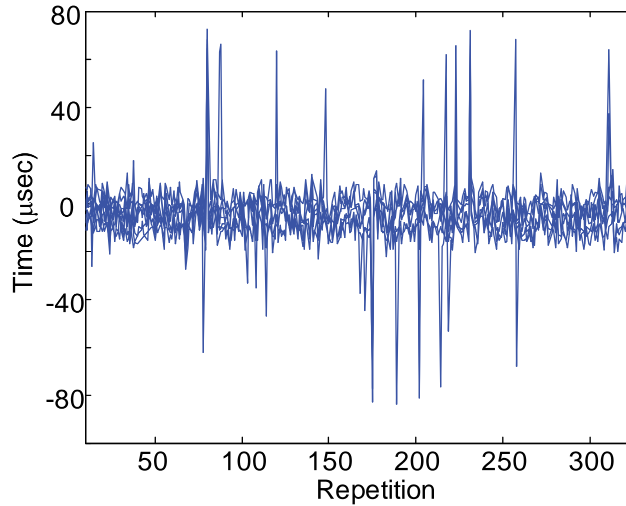

Figure 6. Time synchronization error estimation.

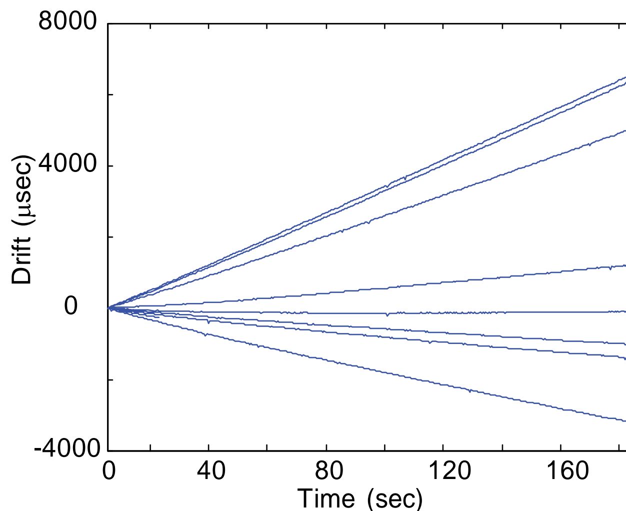

Figure 7. Drift estimation.

above-mentioned factors explaining the error in the time stamps. This figure indicates that an upper-bound estimate of time synchronization error is 80 µs.

The time synchronization error estimated above is considered small for SHM applications. The delay of 10 µsec corresponds to a 0.072-degree phase error for a mode at 20 Hz. Even at 100 Hz, the corresponding phase error is only 0.36 degree.

The same approach is utilized to estimate clock drift. Upon reception of the second packet, which requests replies, the receivers return to the sender their offsets to estimate the global time, instead of global time stamps. If clocks on nodes are ticking at exactly the same rate, the offsets should be constant. This experiment, however, did not show constant offsets. Figure 7 shows the offsets of eight receiver nodes. This figure shows that the drift rate is quite constant in time. The maximum clock drift rate among this set of Imote2 nodes is estimated to be around 50 µs per second. Note that this estimate from the eight nodes is not the upper limit of clock drift because of the small

sample size. This drift is small but not negligible if long measurement records are taken. For example, after a 200 second measurement, the time synchronization error may become as large as 10 ms.

One solution to address this clock drift problem is frequent time synchronization. Time synchronization could be performed often to prevent the synchronization error from accumulating. However, frequent time synchronization is not always feasible. When other tasks are running (e.g., sensing), time synchronization may not perform well. Time synchronization requires precise time stamping while sensing requires precise timing and needs higher priority in execution. Scheduling more than one high priority tasks is challenging, especially for an operating system such as TinyOS, which does not support strict real-time control. If the time synchronization interval is shorter than the sensing duration, another solution needs to be sought to avoid interference between time synchronization and sensing tasks.

The slopes of the lines in Figure 7 are nearly constant and provide an estimate of the clock rate difference. If time synchronization offset values can be observed for a certain amount of time, the slope can be estimated using a least-squares approach. The difference in clock rate is estimated and taken into account in the subsequent data processing as described in Section 4.2.

4.2 Issues toward synchronized sensing

Accurate synchronization of local clocks on Imote2s does not guarantee that measured signals are synchronized. Measurement timing cannot necessarily be controlled based on the global time. To better understand this situation, the sensing process on the Imote2 is first described.

The sensing application program on the Imote2 calls sensor driver commands to perform sensing and obtains measurement data in the following way. A sensing application posts a task to prepare for sensing. Parameters such as the sampling frequency and the total number of data points are passed to the driver. Once the sensor driver becomes ready, sensing starts. The sensing task continues running until a predefined amount of data is acquired. During this sensing, the acquired data points are first stored in a buffer. Every time the buffer is filled, the driver passes the data to the sensing application. This block buffered data is supplied with a local time stamp marked when the last data point of the block is acquired. The clock used for the time stamp runs at 3.25 MHz. If time synchronization is performed prior to sensing, the offset between the global and local times can be utilized to convert the local time stamp to a global time stamp when needed. The data and time stamps passed are copied to arrays in the sensing application, and the buffer is returned to the driver to be used for the next block of data.

Building synchronized sensing on this sampling process is challenging. The difficulties are explained next.

1. Uncertainty in start-up time

Starting sensing tasks at all of the Imote2 nodes in a synchronous manner is problematic. Even when the commands to start sensing are queued at exactly the same moment, the execution timing of the commands is different on each node. Thus, measured signals are not necessarily synchronized to each other.

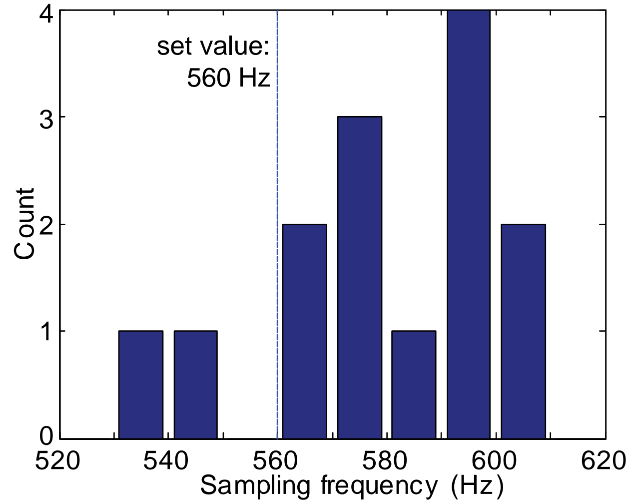

Figure 8. Sampling frequencies of 14 sensor boards.

TinyOS has only two types of threads of execution: tasks and hardware event handlers, leaving users little control to assign priority to commands; if sensing is queued as a task, this task is executed after all the tasks in the front of the queue are completed. As such, the waiting time cannot be predicted. If the command is invoked in a hardware interrupt as an asynchronous command, this command is executed immediately unless other hardware interrupts interfere. However, invocation of commands as a hardware interrupt from a clock firing at very high frequency is not practical; firing the timer at a frequency corresponding to the synchronization accuracy, tens of microseconds, is impossible.

In addition, the warm-up time for sensing devices after the invocation of the start command is not deterministic. Even if the commands are invoked at the same time, sensing will not start simultaneously.

2. Difference in sampling rate among sensor nodes

The sampling frequency of the accelerometer on the available Imote2 sensor boards has nonnegligible random error. According to the data sheet of the accelerometer, the sampling frequency may differ from the nominal value by at most 10 percent (STMicroelectronics, 2008). Such variation was observed when 14 Imote2 sensor boards were calibrated on a shake table (see Figure 8). Differences in the sampling frequencies among the sensor nodes result in inaccurate estimation of structural properties unless appropriate post processing is performed. If signals from sensors with nonuniform sampling frequency are used for modal analysis, one physical mode may be identified as several modes spread around the true natural frequency (Nagayama et al., 2006a).

3. Fluctuation in sampling frequency over time

Additionally, nonconstant sampling rate was observed with the Imote2 sensor boards, which if not addressed, results in a seriously degraded acceleration measurement. When a block of data is available from the accelerometer, the Imote2 receives the digital acceleration signal and obtain the time stamp of the last data point. By comparing differences between two consecutive time stamps,

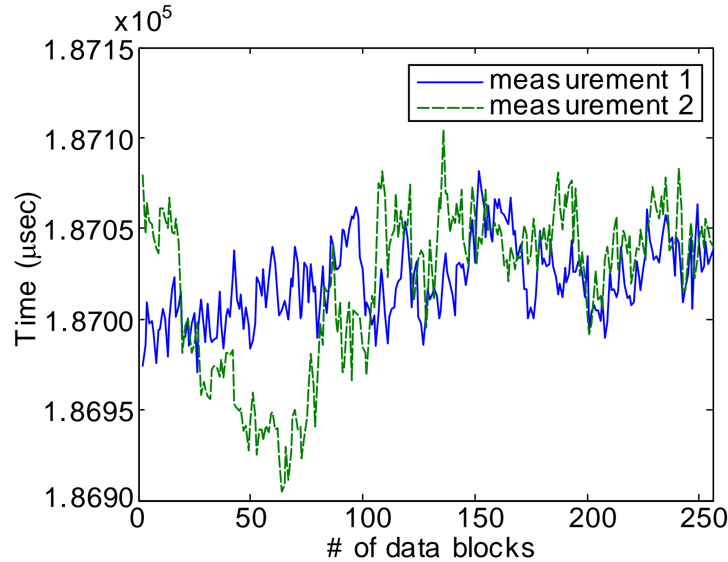

Figure 9. Variation in the sampling frequency over time.

the sampling frequency of the accelerometer is estimated. Figure 9 shows the difference between the time stamps when the block size is set at 110 data points. This figure provides an indication of the variation in the sampling frequency. The difference in two consecutive timestamps fluctuates by about 0.1 percent. Though imperfect time stamping on the Imote2 is a possible source of the apparent nonconstant sampling rate, the slowly fluctuating trend suggests the variable sampling frequency as a credible cause of the phenomenon. With this fluctuation, measurement signals may suffer from a large synchronization error and a nonconstant sampling rate.

4.3 Realization of synchronized sensing

Strict execution timing control is a possible solution for synchronized sensing. Real-time operating systems implemented for industrial systems with ample hardware resources can manage command execution timing precisely. Small embedded systems without operating systems can also be configured to manage execution timing precisely; however, implementation of real-time operation makes the system large and complex. Real-time control of wireless sensors in a network is particularly challenging; the situation is exacerbated for the high sampling rates required in SHM applications. Also, even when a sensor node itself has real-time control, the peripheral devices such as sensor chip may have execution time delay or uncertainty in timing. Instead of pursuing real time control, synchronized sensing is realized herein using post-processing on the non-real-time OS of the Imote2, TinyOS.

Resampling based on the global time stamps addresses the three problems: (a) uncertainty in start-up time, (b) difference in sampling rate among sensor nodes, and (c) time fluctuation in the sampling frequency. The basics of resampling and polyphase implementation of resampling are first reviewed. The resampling approach is then modified to achieve a sampling rate conversion by a noninteger factor. Finally, this proposed resampling method is combined with time stamps of measured data to address concurrently the three issues.

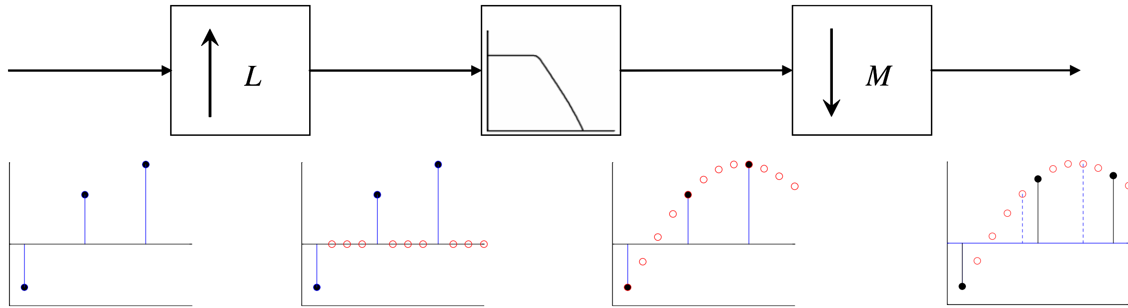

Figure 10. The basic idea of resampling.

Resampling

Resampling by a rational factor is performed by a combination of interpolation, filtering, and decimation. Consider the case in which the ratio of the sampling rate before and after resampling is expressed as an rational factor, L/M. The signal is first upsampled by a factor of L. Then the signal is downsampled by a factor of M. Before downsampling, a lowpass filter is applied to eliminate aliasing and unnecessary high-frequency components (see Figure 10). Using this approach, the signal components below the filter's cut-off frequency are kept unchanged through the resampling process, though imperfect filtering such as ripples in the passband slightly distort the signal.

During upsampling, the original signal $x[n]$ is interpolated by L-1 zeroes as in Eq. (1),

$$v[n] = \begin{cases} x[n/L], & n = 0, \pm L, \pm 2L, ... \\ 0, & otherwise \end{cases} \quad (1)$$

where $v[n]$ is the upsampled signal. In the frequency domain, insertion of zeroes creates scaled mirror images of the original signal in the frequency range between the new and old Nyquist frequencies. A discrete-time lowpass filter with a cutoff frequency, π/L, is applied so that all of these images except for the one corresponding to the original signal are eliminated. To scale properly, the gain of the filter is set to be L.

Before decimation, all of the frequency components above the new Nyquist frequency need to be eliminated. A discrete-time, lowpass filter with a cutoff frequency π/M and gain 1 is applied. This lowpass filter can be combined with the one in the upsampling process. The cutoff frequency of the filter is set to be the smaller value of π/L and π/M. The gain is L.

Decimation by a factor of M is then performed. Thus, the combination of upsampling, filtering, and decimation completes the resampling process.

One of the possible error sources of this resampling process is imperfect filtering. A perfect filter, which has a unity gain in the passband and a zero in the stopband, needs an infinite number of filter coefficients. With a finite number of filter coefficients, passband and stopband ripples cannot be zero. A filter design with 0.1 to 2 percent ripple is frequently used. Figure 11 shows signals before and after resampling. A signal analytically defined as a combination of sinusoidal

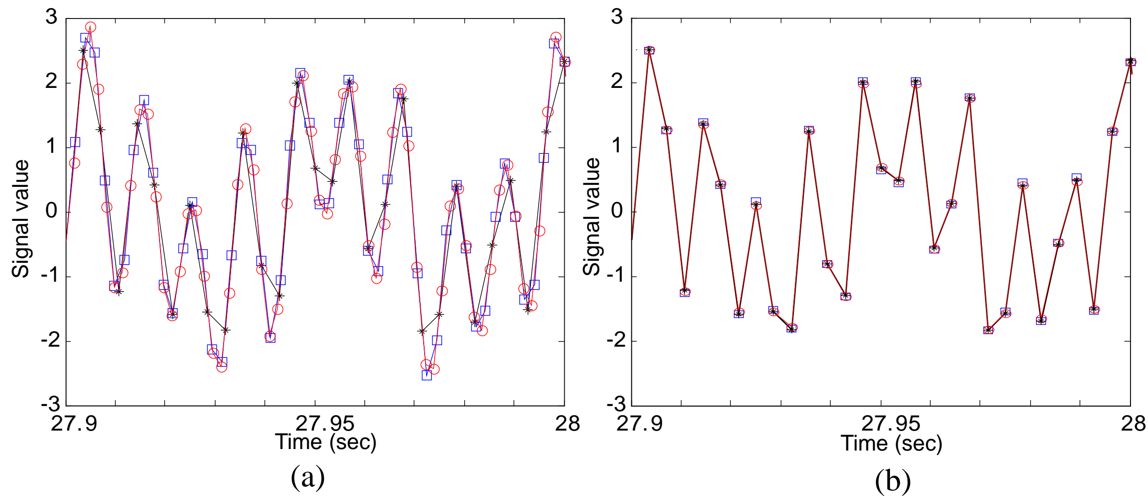

Figure 11. Signals (a) before and (b) after resampling.

waves is sampled at three slightly different sampling frequencies. Two of the signals are then resampled at the sampling frequency of the other signal. As can be seen from Figure 11, after resampling, the three signals are almost identical. These signals are, however, not exactly the same due to the imperfect filtering. Though this signal distortion during filtering is preferably suppressed, this resampling process is not the only cause of such distortion. Other digital filters and AA filters also use imperfect filters. The filter in the resampling process needs to be designed so that it does not degrade signals as compared with the other filters.

The resampling process is considered to be extremely challenging if the upsampling factor, L, is large. This issue is explained herein with examples. When a signal sampled at 100 Hz is resampled at 150 Hz, the rational factor, L/M, is 3/2. The original signal is upsampled by a factor of 3. A lowpass filter with a cutoff frequency of $\pi/3$ can be easily designed with a reasonable number of filter coefficients. Note that the cutoff frequency of the filter is expressed in radian normalized to the sampling frequency of an upsampled signal. This filter is applied to the upsampled signal, which is three times longer than the original one, and then downsampled by a factor of 2. When a signal sampled at 101 Hz is resampled at 150 Hz, on the other hand, the rational factor, L/M, is 150/101. Upsampling by a factor of 101 greatly increases the data size. A lowpass filter with a cutoff frequency of $\pi/150$ requires a large number of filter coefficients. If such a filter is applied to the upsampled signal, the upsampled signal is 101 times longer than the original one. Direct implementation of such resampling on resource-limited smart sensors is intractable.

Polyphase implementation

A Finite Impulse Response (FIR) filter can be computationally much more inexpensive than an Infinite Impulse Response (IIR) filter as a filter for resampling if the polyphase implementation is employed. The polyphase implementation leverages knowledge that upsampling involves inserting many zeroes and that an FIR filter does not need to calculate the output at all of the upsampled data points. This implementation of resampling is explained in this section mainly in the time domain because the extension of the method to achieve synchronized sensing is pursued

using a time domain analysis. Oppenheim et al. (1999) provided a detailed description of the polyphase implementation in the Z-domain.

Upsampling and lowpass filtering with an FIR filter can be written in the following manner:

$$y[n] = \sum_{k=0}^{N-1} h[k]v[n-k]$$
$$v[l] = \begin{cases} x[m], & l = Lm,\ m = 0,\ \pm 1, \pm 2,\ \ldots \\ 0, & otherwise \end{cases} \tag{2}$$

where $x[n]$ is the original signal, $v[l]$ is the upsampled signal, and $y[n]$ is the filtered signal. $h[k]$ is a vector of the filter coefficients for the lowpass FIR filter. The length of the vector of filter coefficients is N. L is, again, an interpolation factor. By combining the two relationships in Eq (2) and the following,

$$Lm = n-k \tag{3}$$

upsampling and filtering can be written as

$$y[n] = \sum_{m=\lceil n-N+1/L \rceil}^{\lfloor n/L \rfloor} h[n-Lm]x[m]. \tag{4}$$

where $\lceil\ \rceil$ and $\lfloor\ \rfloor$ represent the ceiling and floor functions, respectively. The number of algebraic operations is reduced in this equation using the knowledge that $v[l]$ is zero at many points.

Downsampling by a factor of M is formulated as

$$\begin{aligned} z[j] &= y[jM] \\ &= \sum_{m=\lceil jM-N+1/L \rceil}^{\lfloor jM/L \rfloor} h[jM-Lm]x[m] \qquad j=0,\ \pm 1, \pm 2, \ldots \end{aligned} \tag{5}$$

where $z[j]$ is the downsampled signal. The outputs do not need to be calculated at all of the data points of the upsampled signal; rather, they should be calculated only at every M-th data point of the upsampled signal. If an IIR filter was employed, the filter would need outputs at all of the data points of the upsampled signal. This polyphase implementation, thus, reduces the number of numerical operations involved in resampling. However, implementation is still challenging if the upsampling factor, L, is exceedingly large.

Resampling with a noninteger downsampling factor

FIR filter design becomes quite difficult when sampling frequencies need to be precisely converted. For example, resampling of a signal from 100.01 to 150 Hz requires a lowpass filter with a cutoff frequency of $\pi/15000$. The filter needs tens of thousands of filter coefficients. Design of such a filter is computationally challenging. Also a large number of filter coefficients may not fit in the available memory on smart sensors. This problem is addressed mainly by introducing linear interpolation.

First, the resampling is generalized by introducing an initial delay, l_i. Eq. (5) is rewritten with l_i as follows:

$$\begin{aligned} z[j] &= y[jM + l_i] \\ &= \sum_{m=\lceil (jM + l_i - N + 1)/L \rceil}^{\lfloor (jM+l_i)/L \rfloor} h[jM+l_i\text{-}Lm]x[m] \qquad j=0, \pm 1, \pm 2, \ldots \end{aligned} \tag{6}$$

where l_i is integer. By introducing l_i, the beginning point of the downsampled signal can be finely adjusted. If the time difference at the start of sensing is taken into account by l_i, the synchronization accuracy of signals is not limited by the original sampling period.

Resampling is then combined with linear interpolation to achieve the necessary accuracy. The integer, M is replaced by a real number, M_r, and the integer l_i is replaced by a real number, l_{ri}. The upsampling rate, L_a, must remain an integer. In this case, M_r and L_a are not uniquely determined. These values are chosen so that L_a is not too large. A large value of L_a requires a high-order lowpass filter, as is the case for the normal polyphase implementation of resampling. Using these upsampling and downsampling factors, resampling is performed. Upsampling is the same as previously described. However, the downsampling process shown in Eq. (6) cannot be directly applied, due to the noninteger downsampling factor. Output data points to be calculated do not necessarily correspond to points on the upsampled signal. Output data points often fall between upsampled data points. Linear interpolation is used to calculate output values as follows:

$$\begin{aligned} z[j] &= y[p_l] \cdot (p_u - p_j) + y[p_u] \cdot (p_j - p_l) \\ &= \sum_{m=\lceil (p_l - N + 1)/L_a \rceil}^{\lfloor p_l/L_a \rfloor} h[p_l - L_a m]x[m] \cdot (p_u - p_j) \\ &+ \sum_{m=\lceil (p_u - N + 1)/L_a \rceil}^{\lfloor p_u/L_a \rfloor} h[p_u - L_a m]x[m] \cdot (p_j - p_l) \\ p_j &= jM_r + l_{ai} \\ p_l &= \lfloor jM_r + l_{ai} \rfloor \\ p_u &= p_l + 1 \end{aligned} \tag{7}$$

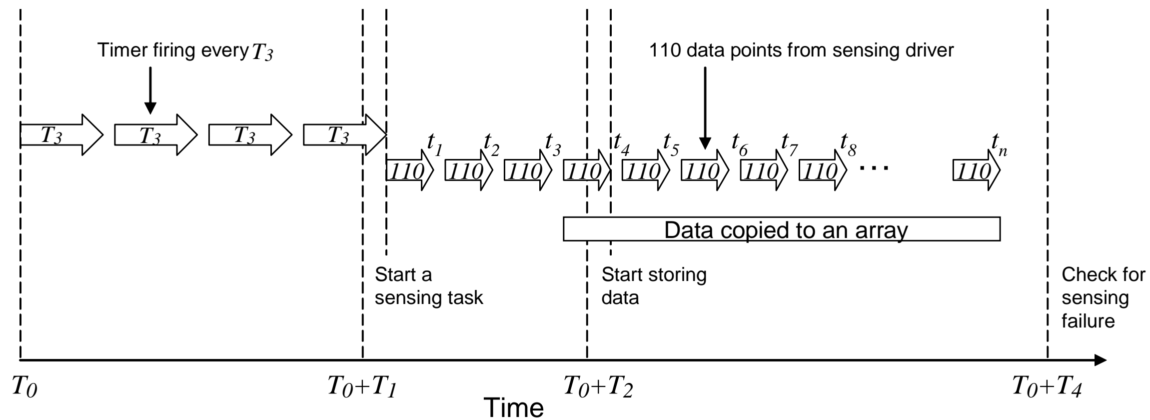

Figure 12. Time stamps and waiting times used for synchronized sensing.

In this way, resampling of an arbitrary noninteger rational factor can be achieved. When L_a is not too small, linear interpolation is effective. A value of L_a ranging from 20 to 150 is employed in algorithmic testing and shown to give reasonable results.

Imote2 implementation using piecewise resampling of time stamped blocks of data

The proposed resampling approach is employed to address issues toward synchronized sensing. Implementation of this approach is first overviewed.

The time stamp to indicate the beginning of the entire sensing process, T_0, and the predetermined waiting times, T_1, T_2, and T_3, are utilized to implement this resampling approach as well as the time stamp at the end of each block of data, $t_i (i = 1, 2, ...n)$. T_1 is the waiting time before the sensing task is posted. A timer that fires every T_3 checks whether the waiting time T_1 has passed. T_2 is the waiting time before the Imote2 starts storing the measured data. The time stamps, t_i, are utilized later to compensate for misalignment of the starting time, as well as for individual differences in sampling frequency and the time varying sampling frequency. Figure 12 illustrates the use of these time stamps and waiting times. This figure also shows the waiting time T_4 for sensing failure detection. If sensing has not finished at T_0+T_4, the Imote2 judges that the sensing process has failed and restarts the process. The details of this approach are explained in the subsequent paragraphs.

When smart sensors are ready to begin sensing, a sensor node managing synchronized sensing in the sensor network gets a global time stamp, T_0, and multicasts it to the other network members. The predetermined waiting times, T_1 and T_2, are also multicast. All of the nodes then start a timer, which is set to be fired every T_3. When the timer is fired, the global time is checked. If the global time is larger than T_0+T_1, a task to start sensing is posted, and the periodic timer stops firing. When a block of measured data becomes available, an event is signaled. In this event, the global time is checked again. If the global time is greater than T_0+T_2, then the block of data is stored in memory. Otherwise, the block of data is discarded. From the time stamp of the last data point of the current block, $t_{current}$, that of the last block, t_{last} , and T_0+T_2, the time misalignment of the first data point of the block and the sensing start time are estimated. When the number of data points in a block is N_{data}, the time misalignment of the first data point, t_{ini} , is estimated as follows:

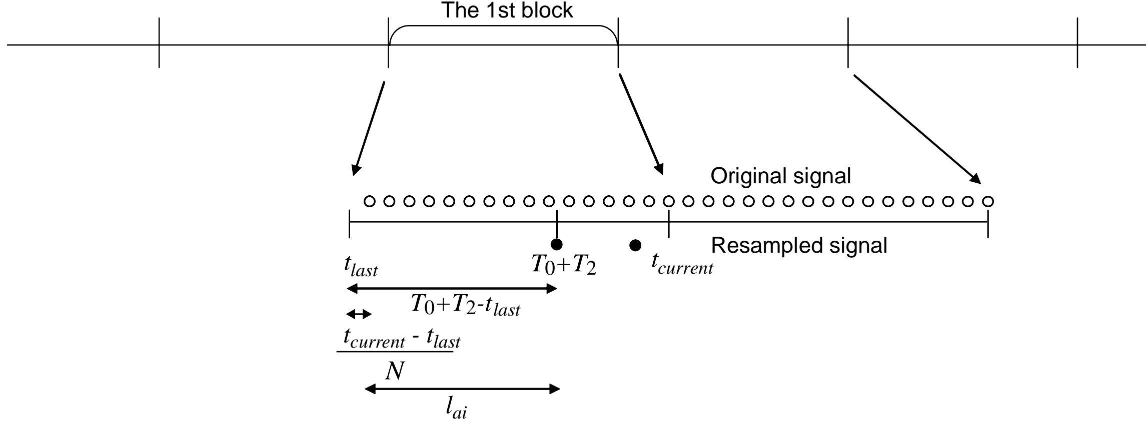

Figure 13. Resampling of the first block of data.

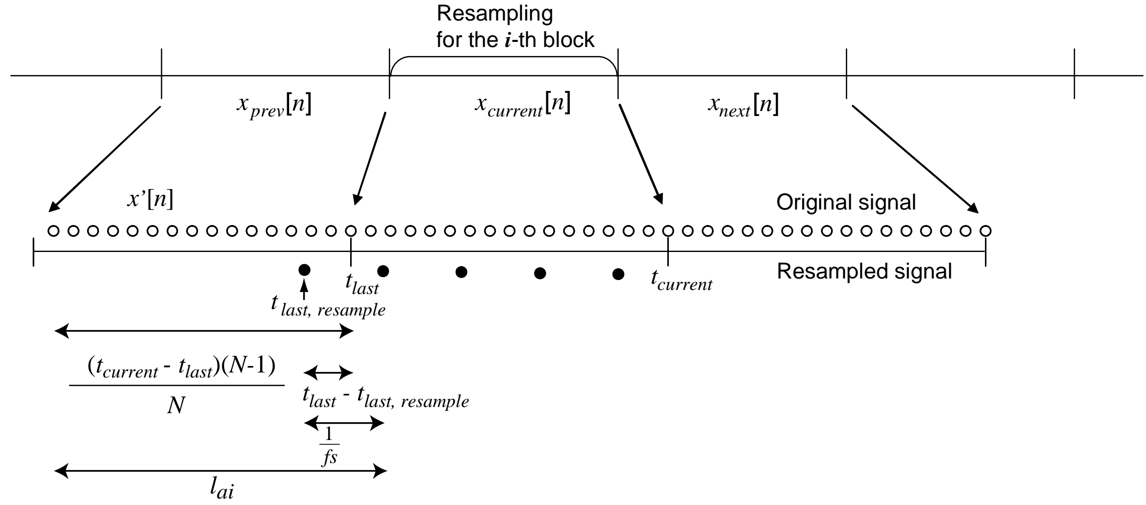

Figure 14. Resampling of the *i*-th block of data (*i*>1).

$$t_{ini} = T_0+T_2-t_{last}-\frac{t_{current}-t_{last}}{N_{data}} \tag{8}$$

where t_{ini} is later used in resampling as l_{ai} in Eq. (7) for the first block of data to compensate for misalignment of the starting time (see Figure 13).

Timestamps and resampling also compensate for difference and fluctuation in the sampling frequency (see Figure 14). The sampling frequency of the current data block, $fs_{current}$,is estimated from $t_{current}$ and t_{last} .

$$fs_{current}= \frac{t_{current}-t_{last}}{N_{data}} \tag{9}$$

The sampling rate conversion by a factor $fs/fs_{current}$ is applied to the block of data. The downsampling factor is determined by

$$M_r = fs_{current} \cdot L_a/fs \tag{10}$$

where fs is the sampling rate after the rate conversion. l_{ai} in Eq. (7) for the first block of data is t_{ini}. For the subsequent blocks, l_{ai} is determined by the time of the last resampled point of the previous block, $t_{last,resample}$. Because Eq. (7) requires $x[m]$ in blocks before or after the current block, data in these blocks are also utilized. To be more specific, when resampling is applied to the current block of data, data from the block before and after the current block are used as part of the input signal for the resampling, $x'[n]$, i.e.,

$$x'[n] = \begin{cases} x_{prev}[n] & 0 \le n < N_{data} \\ x_{current}[n\text{-}N_{data}] & N_{data} \le n < 2N_{data} \\ x_{next}[n\text{-}2N_{data}] & 2N_{data} \le n < 3N_{data} \end{cases} \tag{11}$$

where x_{prev}, $x_{current}$, and x_{next} are data in the previous, current, and next blocks, respectively. The sampling frequency of $x'[n]$ is assumed to be same as that of $x_{current}[n]$. l_{ai} for $x'[n]$ is then calculated as

$$l_{ai} = (t_{current} - t_{last}) \cdot (N_{data} - 1)/N_{data} - (t_{last} - t_{last,\ resample}) + \frac{1}{fs} \tag{12}$$

Using these parameters, Eq. (7) is applied to each data block.

This approach can thus address uncertainty in start-up time, differences in sampling rate among nodes, fluctuation of the frequency over time while the upsampling factor L_a is kept moderate. This resampling-based approach, however, cannot be applied on-the-fly by the Imote2s. The resampling is applied after all of the Imote2s have acquired signals.

While this algorithm can be implemented on Imote2s to achieve synchronized sensing, numerical operations are nontrivial. Eq. (7) still needs a large number of multiplications, additions, etc.; the number of coefficients is still large, frequently greater than a thousand. One thousand filter coefficients in double precision, for example, occupy 8 kB of memory space. These issues are addressed by employing integer operations when applicable.

As is apparent from Eq. (7), scaling FIR filter coefficients results in filter outputs scaled by the same factor. FIR filter coefficients are multiplied by a large constant, η, so that these coefficients can be well approximated by 16-bit integers.

$$h'[n] = \lfloor h[n] \cdot \eta + 0.5 \rfloor \tag{13}$$

These 16-bit integers representing $h'[n]$ are stored on the Imote2s instead of 64-bit double precision data, saving considerable memory space. The scaled output, $y[p_l]$ and $y[p_u]$ in Eq. (14) are estimated only by integer operations; $h'[n]$ and $x[n]$ are both integer type variables.

$$
\begin{aligned}
z[j] &= (y[p_l] \cdot (p_u - p_j) + y[p_u] \cdot (p_j - p_l)) / \eta \\
&= \left(\sum_{m=\lceil (p_l - N + 1)/L_a \rceil}^{\lfloor p_l / L_a \rfloor} h'[p_l - L_a m] x[m] \cdot (p_u - p_j) \right. \\
&\quad + \left. \sum_{m=\lceil (p_u - N + 1)/L_a \rceil}^{\lfloor p_u / L_a \rfloor} h'[p_u - L_a m] x[m] \cdot (p_j - p_l) \right) / \eta \\
p_j &= jM_r + l_i \\
p_l &= \lfloor jM_r + l_i \rfloor \\
p_u &= p_l + 1
\end{aligned} \tag{14}
$$

Then linear interpolation is performed by casting all associated integers to double precision data. The final outcome is adjusted to account for the scaling factor of the filter coefficients. In this way, implementation of the proposed resampling approach becomes less numerically challenging.

5. DATA AGGREGATION

This section demonstrates that distribution and coordination can exploit application specific knowledge so that the data aggregation problem is addressed without sacrificing performance of the SHM algorithms. This data aggregation method is scalable to networks of large numbers of smart sensors (Nagayama et al., 2006b, 2007a; Spencer & Nagayama, 2006).

5.1 Model-based data aggregation

The Natural Excitation Technique (NExT; James et al. 1993) is a widely used modal analysis technique to obtain modal information from output-only measurement of structural vibration. Because the input force to civil infrastructure is usually difficult to measure or estimate, this output-only technique is well-suited for civil infrastructure monitoring. NExT estimates the correlation functions of structural response, which can be further analyzed using modal analysis methods such as Eigensystem Realization Algorithm (ERA; Juang and Pappa 1985) and Stochastic Subspace Identification (SSI; Hermans and Auweraer 1999)].

Correlation functions and their frequency domain representation, spectral density functions, are commonly used analysis tools with a variety of applications. Correlation functions can be exploited to detect periodicities, to measure time delay, to locate disturbance sources, and to identify propagation paths and velocities. Spectral density function applications include identification of input, output, or system properties, identification of energy and noise source, and optimum linear prediction and filtering (Bendat and Piersol 2000).This study employs correlation functions to develop data aggregation strategies for SHM.

Cross spectral density functions are, in practice, estimated from finite length records as in the following equation (Bendat & Piersol, 2000):

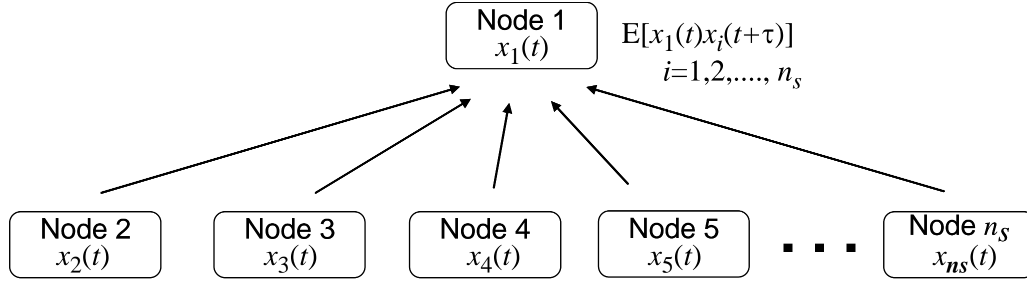

Figure 15. Centralized correlation function estimation.

$$\hat{G}_{xy}(\omega) = \frac{1}{n_d T} \sum_{i=1}^{n_d} X_i^*(\omega) Y_i(\omega) \tag{15}$$

where $\hat{G}_{xy}(\omega)$ is an estimate of cross spectral density function, $G_{xy}(\omega)$, between two stationary Gaussian random processes, $x(t)$ and $y(t)$. $X(\omega)$ and $Y(\omega)$ are the Fourier transforms of $x(t)$ and $y(t)$; * denotes the complex conjugate. T is time length of sample records, $x_i(t)$ and $y_i(t)$. Oftentimes, $x_i(t)$ and $y_i(t)$ are windowed and individual sets of signals may overlap in time. The normalized RMS error, $\varepsilon(|\hat{G}_{xy}(\omega)|)$, of the spectral density function estimation is given as

$$\varepsilon(|\hat{G}_{xy}(\omega)|) = \frac{1}{|\gamma_{xy}| \sqrt{n_d}} \tag{16}$$

$$\gamma^2_{xy}(\omega) = \frac{|G_{xy}(\omega)|^2}{G_{xx}(\omega) G_{yy}(\omega)} \tag{17}$$

where $\gamma^2_{xy}(\omega)$ is the coherence function between $x(t)$ and $y(t)$, indicating the linear dependence between the two signals. The random error is reduced by computing an ensemble average from n_d different or partially overlapped records. Averaging 10 to 20 times is common practice. The estimated spectral densities can then be converted to correlation functions via the inverse Fourier transform.

Correlation function and spectral density function estimation thus requires data from two sensor nodes. Measured data needs to be transmitted from one node to the other before data processing takes place. Associated data communication can be prohibitively large without careful consideration of the implementation. Two approaches for the estimation of these functions are explained next.

An implementation of correlation function estimation in a centralized data collection scheme is shown in Figure 15, where node 1 works as a reference sensor. Assuming n_s nodes, including the reference node, are measuring structural responses, each node acquires data and sends it to the reference node. The reference node calculates the spectral density as in Eq. (15). This procedure is repeated n_d times and averaged. For simplicity, no overlap between individual data sets is assumed. After averaging, the inverse Fourier transform is taken to calculate the correlation

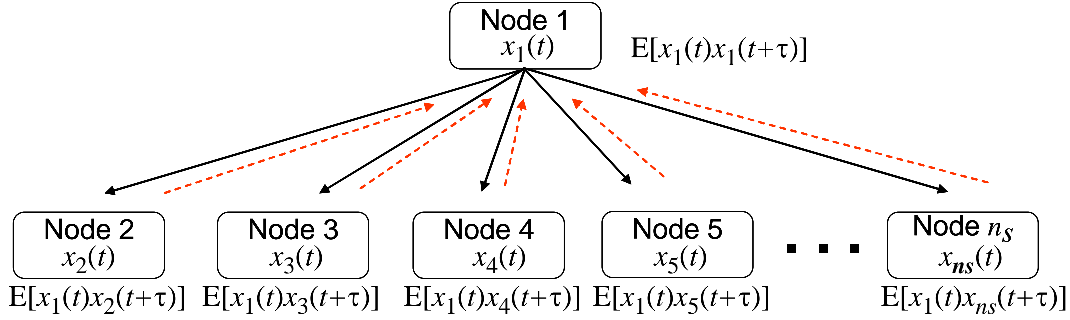

Figure 16. Distributed correlation function estimation.

function. All calculations take place at the reference node. When the spectral densities are estimated from discrete time history records of length N, the total data to be transmitted over the network using this approach is $N \times n_d \times (n_s - 1)$.

In the next scheme, data flow for correlation function estimation is examined and data transfer is reorganized to take advantage of computational capability on each smart sensor node. After the first measurement, the reference node broadcasts the time record to all of the nodes. On receiving the record, each node calculates the spectral density between its own data and the received record. This spectral density estimate is locally stored. The nodes repeat this procedure n_d times. After each measurement, the stored value is updated by taking a weighted average between the stored value and the current estimate. In this way, Eq. (15) is calculated on each node. Finally the inverse Fourier transform is taken of the spectral density estimate locally. The resultant correlation function is sent back to the reference node. Because subsequent modal analysis algorithms such as ERA uses, at most, half of the correlation function data length, $N/2$ data points are sent back to the reference node from each node. The total data to be transmitted in this scheme is, therefore, $N \times n_d + N/2 \times (n_s - 1)$ (see Figure 16).

As the number of nodes increases, the advantage of the second scheme, in terms of communication requirements, becomes significant. The second approach requires data transfer of $O(N \cdot (n_d + n_s))$, while the first one needs to transmit data to the reference sensor node of the size of $O(N \cdot n_d \cdot n_s)$. The distributed implementation leverages knowledge regarding the application to reduce communication requirements as well as to utilize the CPU and memory in a smart sensor network efficiently.

The data communication analysis above assumes that all the nodes are in single-hop range of the reference node. This assumption is not necessarily the case for a general SHM application. However, Gao (2005) proposed a DCS approach for SHM that supports this idea. Neighboring smart sensors in single-hop communication range form local sensor communities and perform SHM in the communities. In such applications, the assumption of nodes being within single-hop range of a reference node is reasonable.

Further consideration is necessary to accurately assess the efficacy of the distributed implementation. Power consumption of smart sensor networks is not simply proportional to the amount of data transmitted. Acknowledgment messages are also involved. The radio listening mode consumes power, even when no data is received. However, the size of the measured data is usually much larger than the size of the other messages to be sent and should be considered the primary factor in determining power consumption. Reduced data transfer requirements realized

Figure 17. Three-dimensional, 5.6 m-long truss structure.

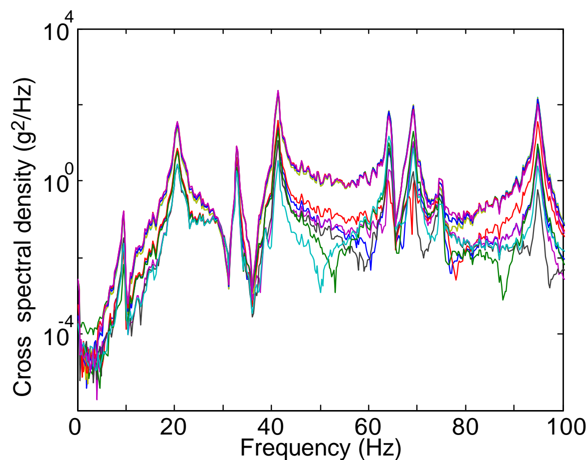

Figure 18. Cross-spectral density estimates from Imote2 data.

by the proposed model-based data aggregation algorithm will lead to decreased power consumption.

6. EXPERIMENTAL EVALUATION

The three middleware services are implemented on the Imote2 and their performance is experimentally evaluated. Six Imote2s are placed on the three-dimensional truss model located at the Smart Structure Technology Laboratory of the University of Illinois at Urbana-Champaign (see Figure 17; http://sstl.cee.uiuc.edu). These Imote2s measure the acceleration response of this truss under band-limited white noise excitation, using the synchronized sensing middleware service. As explained in the model-based data aggregation section, the data is then transferred from a reference node to the other nodes using the multicast data transfer middleware service. Correlation functions are then calculated and sent back to the reference and base station nodes using the reliable communication middleware service. Associated command transfer is accomplished by the unicast and multicast reliable command transfer middleware services.

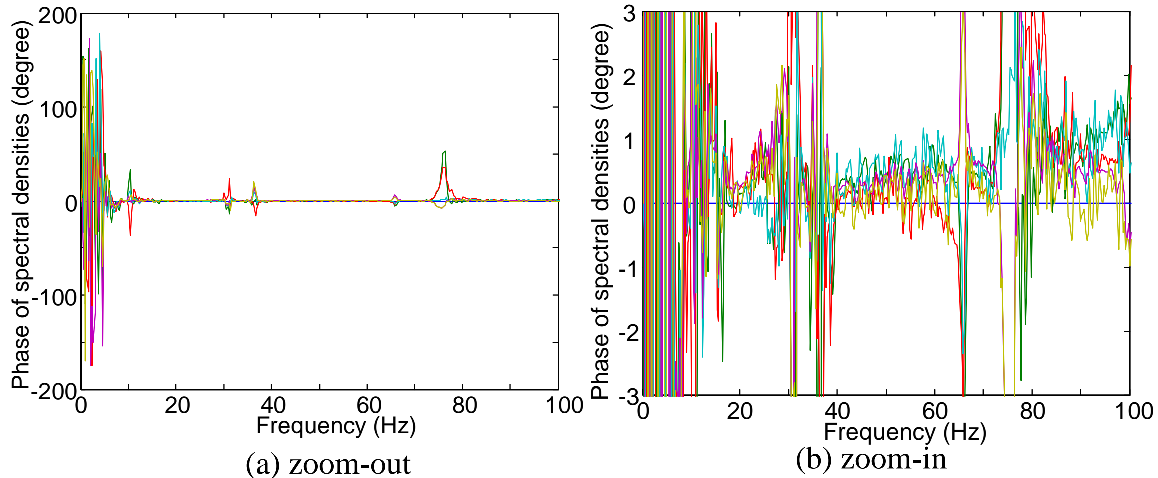

Figure 19. Phase of cross spectral densities.

The collected correlation functions are converted to spectral density functions and shown in Figures 18 and 19. The amplitude of cross spectral density functions show clear peaks corresponding to structural vibration modes. The phase of the spectral density functions is considered to be constant over a wide frequency range if the sensors are installed on a rigid body and if signals are synchronized with each other. The slopes of the phase indicate synchronization errors. Though the truss does not behave as a rigid body, the slopes still indicate the synchronization errors. The Imote2s were installed on neighboring truss nodes. Because the mode shapes of the truss do not have a mode shape node in this neighborhood in the frequency range below 100Hz, the flat phase in Figure 19 demonstrates accurately synchronized sensing.

7. SUMMARY

Middleware services for SHM applications using smart sensors were developed. The data loss problem, which has adverse effects on an SHM algorithm, was addressed by developing reliable communication protocols. To realize synchronized sensing, a resampling-based approach was taken; the synchronized sensing enables highly accurate distributed sensing. Model-based data aggregation, including distribution of data processing and coordination among sensor nodes, provided scalability to a large number of smart sensors while preserving vibration signal information to be used in many of subsequent structural vibration analyses. These middleware services are implemented on the Imote2 and validated through a structural vibration measurement experiment. This development of the middleware services is expected to allow many SHM application users to obtain reliable structural response information from smart sensor networks smoothly and perform detailed structural analysis. Current efforts are underway to package these middleware services so that they can be widely distributed.

REFERENCES

Bendat, J. S., & Piersol, A. G. (2000). *Random data: analysis and measurement procedures*, New York: Wiley.

Crossbow Technology, Inc. <http://www.xbow.com>

Elson, J., Girod, L., & Estrin, D. (2002). “Fine-grained network time synchronization using reference broadcasts.” *Proceedings of 5th Symposium on Operating Systems Design and Implementation*, Boston, MA.

Ganeriwal, S., Kumar, R., & Srivastava, M. B. (2003). “Timing-sync protocol for sensor networks.” *Proceedings of 1st International Conference On Embedded Networked Sensor Systems*, Los Angeles, CA., 138-149.

Gao, Y. (2005). Structural health monitoring strategies for smart sensor networks (Doctoral dissertation, University of Illinois at Urbana-Champaign, 2005).

Hermans, L. & Auweraer, H. V. (1999). “Modal testing and analysis of structures under operational conditions: industrial applications.” *Mechanical Systems and Signal Processing*, 13(2), 193-216.

James, G. H., Carne, T. G., & Lauffer, J. P. (1993). “The natural excitation technique for modal parameter extraction from operating wind turbine,” *Report No. SAND92-1666*, *UC-261*, Sandia National Laboratories, NM.

Juang, J.-N. & Pappa, R. S. (1985). “An eigensystem realization algorithm for modal parameter identification and model reduction.” *Journal of Guidance, Control, and Dynamics*, 8(5):620-627.

Kim S., Pakzad S., Culler D., Demmel J., Fenves G., Glaser S. & Turon, M. (2007). “Health monitoring of civil infrastructures using wireless sensor networks.”*Proceedings of the 6th international conference on Information processing in sensor networks*, Cambridge, MA, 254-263.

Lamport, L., Shostak, R., and Pease, M. (1982). “The Byzantine Generals Problem.”, ACM Transactions on Programming Languages and Systems, 4(3), 382-401.

Lynch, J. P. & Loh, K. (2006). “A summary review of wireless sensors and sensor networks for structural health monitoring.” *Shock and Vibration Digest*, 38(2), 91-128.

Maroti, M. Kusy, B., Simon, G., & Ledeczi, A. (2004). “The flooding time synchronization protocol.” *Proceedings of 2nd International Conference On Embedded Networked Sensor Systems*, Baltimore, MD, 39-49.

Mechitov, K. A., Kim, W., Agha, G. A., & Nagayama, T. (2004). “High-frequency distributed sensing for structure monitoring.” *Proceedings of 1st Int. Workshop on Networked Sensing Systems*, Tokyo, Japan, 101–105.

Nagayama, T. (2007). Structural health monitoring using smart sensors. (Doctoral dissertation, University of Illinois at Urbana-Champaign, 2007).

Nagayama, T., Rice, J. A., & Spencer, B. F., Jr. (2006a). “Efficacy of Intel's Imote2 wireless sensor platform for structural health monitoring applications.” *Proceedings of Asia-Pacific Workshop on Structural Health Monitoring*, Yokohama, Japan.

Nagayama, T., Spencer, B. F., Jr., Agha, G. A., & Mechitov, K. A. (2006b). “Model-based data aggregation for structural monitoring employing smart sensors”, *Proceedings of 3rd Int. Conference on Networked Sensing Systems (INSS 2006)*, Chicago, IL, 203-210.

Nagayama, T. Sim, S-H., Miyamori, Y., & Spencer, B.F. Jr. (2007a). “Issues in structural?health monitoring employing smart sensors.” Smart Structures and Systems, 3(3), 299-320.

Nagayama, T., Spencer, B. F. Jr., & Fujino, Y. (2007b). “Synchronized sensing toward structural health monitoring using smart sensors.” Proceedings of World Forum on Smart Materials and Smart Structures Technology (SMSST 2007), Chongqing & Nanjing, China.

Oppenheim, A. V., Schafer, R. W., & Buck, J. R. (1999). *Discrete-time signal processing*. Upper Saddle River, NJ: Prentice Hall.

Spencer, B. F., Jr. & Nagayama, T. (2006). “Smart sensor technology: A new paradigm for structural health monitoring.” *Proceedings of Asia-Pacific Workshop on Structural health Monitoring*, Yokohama, Japan.

STMicroelectronics. <http://www.st.com/stonline/>.

Woo, A., Tong, T. & Culler, D. (2003) “Taming the underlying challenges of reliable multihop routing in sensor networks.” *Proceedings of the 1st international conference on Embedded networked sensor systems*, Los Angeles, California, 14-27.