A Software Architecture for Adaptive Modular Sensing Systems

Andrew C. Lyle and Michael D. Naish
Sensing and Mechatronic Systems Lab
Dept. of Mechanical and Materials Engineering
The University of Western Ontario
London, Ontario, Canada
alyle3@uwo.ca, naish@eng.uwo.ca

***Abstract*—In this paper, a software architecture and knowledge representation scheme that enables the combination and reconfiguration of modular sensor and actuator components is described. The proposed software architecture utilizes a real-time operating system with a pre-emptive kernel, which simplifies the implementation of the architecture itself through the modularization and concurrent execution of its various software components. A virtual machine-based middleware layer runs on top of the operating system, enabling platform-independent *logical algorithms* to be written once, and run on any module irrespective of its underlying hardware architecture. Logical algorithms govern the behaviour of a given set of heterogeneous modules, providing them with intelligence and enabling them to behave as a single entity known as a *logical module*.**

I. Introduction

The ability to combine diverse modular sensor and actuator components to produce arbitrarily complex and flexible modular sensor systems is a technique that will prove useful in many applications. Application domains include flexible inspection, mobile robotics, surveillance, and even space exploration. In current practice, a fixed combination of sensors and actuators is typically used, tailored to a specific application. Such systems cannot be cheaply or quickly reconfigured to handle a change in process requirements. The ability to combine modular sensor and actuator components in a simple manner allows for rapid reconfiguration to suit any requirement.

Each modular component provides a core sensing or actuation functionality, and may be connected to other components to form more capable composite sensors. A composite sensor is able to automatically determine its overall geometry and assume an appropriate collective identity. If reconfigured, the composite sensor may then assume a completely different identity to match its new geometry. In order to realize this vision, an architecture and knowledge representation scheme that enables the flexible and reliable combination of modular sensor and actuator components is proposed.

A. Software Architecture Design Criteria

The design of the architectural framework should fulfill the following criteria:

1) Autonomy: Support the autonomous discovery of the capabilities of networked modules, and the autonomous configuration of these modules based on their discovered capabilities.

2) Heterogeneity: Allow the connection of sensor and actuator modules of diverse types.

3) Pose/Geometry Determination: Support the determination of the absolute or relative *pose* (position and orientation) of individual modules, and by extension the overall geometry of a set of connected modules.

4) Process Distribution: Support the splitting and distribution of a complex task among a group of networked modules.

5) Resource Management: Manage the hardware resources on each module in an efficient, intuitive, and simple manner.

6) Robustness: Adapt automatically to the addition, removal, or failure of modules in real-time.

7) Scalability: Maintain reliable operation with an increasing number of connected sensor and actuator modules.

B. Software Architecture Stack

The software architecture outlined in this paper is a distributed architecture loosely based on the *Open Systems Interconnection* (OSI) reference model [1], and consists of five layers and two sub-layers as shown in Fig. 1. Using an layered architecture model, where each layer is encapsulated, allows the implementation of any layer to change independently of the others. The function of each layer is defined as follows:

1) Hardware Layer: Contains the physical components of a module needed for execution of the operating system.

2) Driver Layer: Contains the low-level software routines used by the operating system to manipulate and manage the hardware resources present in the module.

3) Data Link Layer: Provides an interface to the wireless transceiver driver that automatically accounts for transmission problems such as packet loss.

4) Real-Time Operating System: Provides resource management functionality as well an environment for concurrent task execution.

5) Middleware Layer: Provides the basic services needed for modules to interact and communicate with each other to achieve a specific goal.

6) Virtual Machine (VM): Provides an environment for the execution of platform-independent instructions.

7) Logical Layer: Comprised of one or more platform-independent logical algorithms which enable a group of modules to behave as a logical entity.

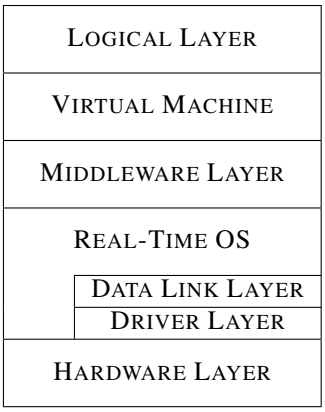

Fig. 1. Software Architecture Stack

II. Related Work

A number of implementations of reconfigurable sensing systems composed of modular sensor and actuator components exist. A popular implementation is the UC Berkeley *Mica* platform [2]. Each Mica node, known as a *mote*, measures 1.25×2.25 inches and runs the TinyOS real-time operating system [3]. Although the motes are capable of collaboration through the use of a peer-to-peer multi-hop wireless networking protocol, no actuation capabilities are supported, and therefore the motes are limited to operating in non-active sensing applications. A similar project is the *Smart-Its* [4] project. Smart-Its are self-contained nodes, as small as 17×25×15 mm, designed to be stuck onto everyday objects. The objects are thus enhanced with sensing and computational capabilities. In addition to its own sensor reading, each node can gather readings from other nodes through a wireless interface. These values may then be processed and transmitted to higher-end devices such as personal computers. However, like the Mica motes, the active operation and automatic reprogramming capability of Smart-Its nodes is limited.

More closely related to the system described in this paper are the *eBlocks* [5] embedded system building blocks. By connecting these blocks together, simple sensor networks may be constructed even by users who are not technically adept. Each block is controlled by a Microchip PIC microcontroller, and is capable of a simple predefined function such as sensing, output, boolean logic, or wireless communication. Although reconfigurable, connected blocks are unable to determine their overall geometry or automatically assume a collective identity to suit new configurations. The possible applications of the system are also limited due to the usage of simple boolean logic functions to generate actions based on sensed events. Similarly, the *I-BLOCKS* [6] project uses DUPLO bricks which are populated with a Microchip PIC microcontroller as well as sensors and actuators. Blocks are able to communicate with each other when physically connected, and may also employ wireless communication if desired. However, like the eBlocks, connected blocks are unable to determine their overall geometry or automatically assume a suitable collective identity. The blocks are also not designed to be easily reprogrammed to suit changing application requirements.

III. Hardware Overview

The basic module used to construct modular sensing systems is the *Transducer Interface Module* (TIM). Each is capable of a single sensing or actuation function, and is uniquely identified by a 64-bit address defined at the time of manufacture. As specified in the IEEE 1451 standard for smart transducers [7], each module possesses a *Transducer Electronic Datasheet* (TEDS) in non-volatile memory, from which a description of the characteristics of its associated sensor or actuator may be obtained.

TIMs are cubical in shape, and thus each possesses six faces to which up to five other modules may be connected. One face is reserved for use by the transducer associated with the module. The hardware which comprises a TIM, shown in Fig. 2, includes the associated transducer; a high-speed NXP Semiconductors LPC2148 ARM-based microcontroller [8]; a Nordic Semiconductor nRF24L01 [9] wireless transceiver supporting high-speed data transmission, multi-channel operation, and carrier detection; a Secure Digital™ flash memory card providing high-capacity, non-volatile storage for data and algorithms; a power supply capable of providing a voltage of 3.3 volts to 9 volts; and five *module connectors*, which are proprietary interfaces used to physically connect additional modules. Further details may be found in [10].

A. Other Module Types

A modular sensing system may consist of two other types of modules significant to the software architecture. These modules perform tasks unrelated to sensing and actuation; instead, they support the inter-operation of a group of TIMs.

1) Administration Module: An administration module is used by the system user to detect and manage TIMs within its vicinity, and possesses only a power supply, a microcontroller, and a transceiver. It may be integrated into a complete computer system, or be a small, self-contained console with a user interface. Administration modules may also act as a sink for transducer readings, and as a gateway for communication with a larger network, such as the Internet.

2) Interconnect Module: Interconnect modules (see [10]) are each built to assume one of a variety of non-standard shapes, and are used to provide angular and translational offsets between connected TIMs which would otherwise not be possible due to the cubical shape of the TIMs. They possess only a microcontroller and module connectors, and draw power from the TIMs to which they are connected. The nature of the offset provided by a particular interconnect module is stored in its TEDS, and may be accessed through its module connectors.

IV. Operating System

The software architecture utilizes a *real-time operating system* (RTOS), allowing it to be implemented in a modular

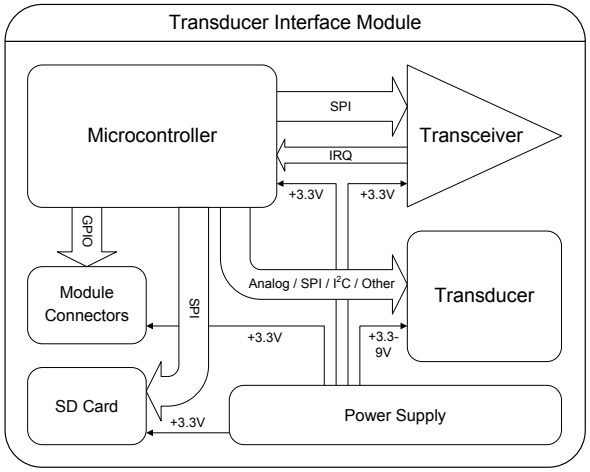

Fig. 2. Transducer Interface Module Hardware Block Diagram

fashion through the concurrent execution of *tasks*. Management of module hardware resources, as well as development and debugging, is simplified as a result.

In an RTOS, concurrently executing tasks may be scheduled using either a *pre-emptive* scheduling policy or a *cooperative* scheduling policy. In pre-emptive scheduling, CPU time is automatically shared between tasks based on their assigned priority, while in cooperative scheduling each task maintains control of the CPU until it explicitly yields control. Pre-emptive scheduling is advantageous since it prevents long-running, low-priority *background* tasks from blocking shorter, higher-priority *foreground* tasks from executing, thus improving system response speed to external events. In the popular *TinyOS* [3] RTOS, which utilizes a cooperative scheduler, all tasks must run to completion. Long-running background tasks are therefore prohibited, and care must be taken to ensure that each task completes in a reasonable amount of time.

The operating system utilized in the software architecture presented herein is *TNKernel* [11]. This RTOS was chosen because it is free, open source, compact, well documented, and contains a priority-based pre-emptive task scheduler. TNKernel also makes provisions for message passing and synchronization between concurrently executing tasks.

Upon startup of a module, standard background tasks are created that execute throughout the duration of its operation. These include the *synchronization task*, which ensures that the local clock on the module stays synchronized with others in its vicinity; the *discovery task*, which examines broadcast packets transmitted by other modules to determine their supported capabilities; and the *beacon task*, which causes the module to periodically transmit a special broadcast packet to the network indicating its presence.

V. Data Link Layer

The purpose of the *data link layer* is to provide *logical link control* in the form of a reliable, connection-oriented service, *medium access control* to prevent channel access conflicts, and a mechanism for *time synchronization* between modules. The data link layer accepts *messages* from the middleware layer, splits them into discrete *packets*, and transmits them through the transceiver driver. Conversely, the data link layer also accepts incoming packets from the transceiver driver, merges them into messages if necessary, and submits them to the middleware layer.

A. Packet Format

Data is transferred to and from the transceiver driver in 40-byte packets. The packet format, shown in Table I, is designed to be compatible with that of the nRF24L01 transceiver, which manages the *preamble*, *broadcast address*, and *checksum* fields within its firmware. The 32-byte payload field defined within the nRF24L01 packet format is sub-divided into smaller fields for the purposes of the software architecture. The data link layer packet fields are described as follows:

- **Preamble** (1 byte) — A pattern of alternating ones and zeroes used by a receiving transceiver to synchronize its clock with that of the transmitting transceiver.
- **Broadcast Address** (5 bytes) — A constant value common to all modules. Any received packet that does not specify this address is automatically rejected by the transceiver firmware.
- **Source Address** (8 bytes) — Identifies the physical or logical module that transmitted the packet.
- **Destination Address** (8 bytes) — Identifies the physical or logical module that should receive the packet.
- **Packet Type** (1 byte) — Indicates the type of the packet. Currently only four bits are utilized; the remaining four bits are reserved for future expansion.
- **Sequence Number** (1 byte) — Used to detect packets retransmitted due to loss. The sequence number is incremented modulo 256 before each packet transmission.
- **Data Field** (14 bytes) — Contains the data to be transmitted in the packet. The data field may be further subdivided into *parameter fields* used for transmitting various types of data specific to the indicated packet type.
- **Checksum** (2 bytes) — Used to detect packet transmission errors. The checksum is automatically generated by the transceiver firmware using the *Cyclic Redundancy Check* (CRC) algorithm [1].

B. Channels and Packet Types

The nRF24L01 transceiver is able to transmit and receive packets on any one of up to 125 distinct radio frequency channels at a time, one of which is reserved by the software architecture for use as a *control channel*. All modules listen to the control channel by default when not transmitting data, and each can detect the presence of others in its vicinity by listening for packet transmission activity on the channel.

The other 124 channels are utilized as *data channels*. Upon successful reservation of a data channel through the use of the RTS and CTS medium allocation packets, the transmitting and receiving modules switch to the agreed channel and carry out the transmission. As depicted in Table II, lengthy

TABLE I
DATA LINK LAYER PACKET FORMAT (FIELD SIZES IN BYTES)

Pre (1)	Broadcast Address (5)
Source Address (8)	
Destination Address (8)	
Type (1)	Seq (1)
Data Field (14)	
	Chk (2)

TABLE II
MULTI-CHANNEL OPERATION (ACK PACKETS NOT SHOWN)

Control Ch.	CTS	RTS	CTS		SYQ	SYR
Data Ch. 1		MSG	DAT	DAT	END	
Data Ch. 2				MSG	DAT	END
⋮						
Data Ch. 124	DAT	DAT	DAT	END		

transmissions may occur simultaneously on different channels without interfering. The various types of packets transmitted on the control and data channels are briefly described below.

1) Control Channel Packet Types:

- **PRE** — *Presence* packets are regularly broadcast by all modules to indicate their continued presence in the sensing system. The following module attributes are specified in the data field: *type* (sensor or actuator), *class* (temperature, pressure, display, etc.), *return* (any one of the *data objects* described in Section VI), and *reference flag* (indicates if the local clock of the module is a reference clock to which other modules may synchronize).
- **RTS** — *Request To Send* packets are used to request transmission of a middleware layer message. The data channel to be used for the transfer, and the message length, are specified within the data field.
- **CTS** — *Clear To Send* packets are issued by the receiving module to the transmitting module to indicate that it may proceed with the transmission.
- **SYQ** — *Synchronization Query* packets are used to initiate time synchronization with a reference module. Modules attempt to synchronize on startup, and assume the role of a reference if synchronization fails. References relinquish their role if another reference is detected with a higher address. The synchronization protocol used is based on the *Simple Network Time Protocol* (SNTP) [12].
- **SYR** — *Synchronization Response* packets are issued by reference modules, and contain timestamps within the data field used by unsynchronized modules to calculate their relative time offset.

2) Data Channel Packet Types:

- **MSG** — *Message* packets are used to indicate the start of a new middleware layer message.
- **DAT** — *Data* packets contain consecutive fragments of a middleware layer message in the data field.
- **END** — *End* packets are issued by the transmitting module to indicate that the transmitted DAT packets comprise a complete middleware layer message.
- **ACK** — *Acknowledgement* packets are used to indicate that transmitted packets have been successfully received. They may also be used to pause lengthy transmissions. A boolean value indicating if the transmission should be paused is specified in the data field.
- **RES** — *Resume* packets are issued by the receiving module to indicate that a paused transmission may be resumed.

VI. MIDDLEWARE LAYER

The purpose of the *middleware layer* is to facilitate interoperability between the various modules in a modular sensing system. At the middleware layer, the *application programming interface* (API) for physical and logical modules is defined, which is comprised of a variety of *API service functions*. API service functions are the interface through which modules request services from, and information about, each other. Data is transferred between modules in the form of variable-length *messages*.

A. Message Format

A message consists of a 38-byte header, followed by a single variable-length block containing the data to be transferred in the message, as shown in Table III. Modules request services from other modules by issuing *call* messages to API service functions. The results are returned in the form of *return* messages. The middleware layer message fields are described as follows:

- **Source Address** (8 bytes) — Identifies the physical or logical module that transmitted the message.
- **Destination Address** (8 bytes) — Identifies the physical or logical module that should receive the message.
- **Deadline** (8 bytes) — If applicable, indicates the time at or before which an API service call should be completed.
- **Timestamp** (8 bytes) — Indicates the time at which the API service call was completed.
- **Message Type** (1 byte) — Indicates the message type. Messages may be a synchronous or asynchronous service call to be completed before, at, or without a deadline, or be a return to a service call.
- **API Service Function** (1 byte) — Indicates what API service function should be invoked if the message is a service call, or what API service function was invoked if the message is a return.
- **Data Length** (4 bytes) — Indicates the length of the data field in bytes.
- **Data Field** (up to 2^{32} bytes) — Arbitrary data may be embedded within the data field in the form of *data objects*. Data objects are loosely based on the concept of *data chunks* described in the *Interchange Format File* (IFF) standard [13], and contain a *type* field, zero or more *parameter* fields, and a *content* field. The data object types which may be specified are *char*, *short*, *int*, *long*, *float*, *double*, *string*, *error* (also a string), and *array* (a container for other data objects).

TABLE III
MIDDLEWARE LAYER MESSAGE FORMAT (FIELD SIZES IN BYTES)

Source Address (8)		
Destination Address (8)		
Deadline (8)		
Timestamp (8)		
Msg (1)	API (1)	Data Length (4)
Data Field (0 - 2^{32})		

B. API Service Functions

Service functions on a module may be invoked manually through an administration module, or automatically by other modules within the network. The mechanism used to invoke service functions, known as a *service call*, is loosely based on the *Remote Procedure Call* (RPC) protocol [14]. Short-running calls are promptly handled as foreground tasks, while longer-running calls execute concurrently as background tasks. Service calls are issued either *synchronously* or *asynchronously* to a remote module through the use of middleware messages. A synchronous service call causes the RTOS to suspend the calling task until the corresponding return message is received from the remote module, or the service call times out, while during an asynchronous service call the task is allowed to concurrently continue execution. If a return message contains an error string, the corresponding *error handler* is invoked if present, otherwise a default action is performed.

Service functions are provided which enable various properties of a module to be obtained (*get* functions), modified (*set* functions), or transmitted (*send* functions). These properties include the value of its associated transducer, the value in its TEDS corresponding to one or more specified keys, its *pose*, its *reference flag*, and its *face enumeration* (a list of addresses corresponding to the modules connected to each of its faces). Various administrative API service functions are also provided. These include functions that instruct a module to join or leave a specified logical entity, allow the module to transmit information about new connections to its faces, or cause the module to shut down and leave the network of modules entirely.

VII. VIRTUAL MACHINE

A *virtual machine* (VM) is a program which interprets and executes high-level, hardware-independent abstract bytecodes. Algorithms defined using these bytecodes are therefore completely decoupled from the underlying hardware architecture on which they execute. A VM was designed and implemented for the proposed software architecture, enabling logical algorithms to be specified once and then used, without recompilation, in the dynamic reprogramming of a variety of heterogeneous modules and hardware architectures as application requirements change. Dynamically modifying the behavior of a module by uploading new binary code to be executed directly, as is done in the *Contiki* [15] and *SOS* [16] systems, results in algorithms being hardware and instruction set dependent. Another benefit of utilizing a VM is that frequently-occurring sequences of low-level operations may be implemented natively and abstracted to a single VM bytecode, making algorithms easier to create, debug, and maintain. A popular VM for high-end computer systems is Sun Microsystems' *Java Virtual Machine* [17], while VM implementations for embedded systems include *Maté* [18] and *Scylla* [19].

TABLE IV
VIRTUAL MACHINE INSTRUCTION FORMAT

Bytecode	Rd	Ra	Rb	Ca	Cb

Current VM implementations are either *stack-based*, in which data is manipulated on a common stack, or *register-based*, in which data is manipulated using discrete registers. The VM implemented for the software architecture is register-based since fewer instructions are required, on average, to implement a particular behaviour in these VMs. The VM instruction format is as shown in Table IV. Instructions are variable-length and specify an 8-bit *bytecode* which indicates the operation to carried out on up to five *operands*: the 8-bit destination register index *Rd*, the 8-bit source register indices *Ra* and *Rb*, and the 32-bit constant fields *Ca* and *Cb*.

The registers available within the VM architecture are a 32-bit program counter register *pc*, which points to the next instruction to be executed, and 256 general-purpose dynamic registers *r0* to *r255*, which store *data objects* of the types described in Section VI and are allocated as needed. The type of data stored is determined when accessed at runtime, similar to variables in dynamically-typed programming languages such as *Python* [20]. For increased flexibility the VM is supplemented by a *data stack*, used to store data objects of any size, and a *call stack*, used by the VM to keep track of function return addresses. Both stacks are implemented as linked lists. Since the VM architecture closely mirrors that of modern CPU architectures, optimization of the bytecode interpreter is simplified.

Algorithms are developed using a custom-written assembler targeted to the *instruction set architecture* (ISA) of the VM. The assembler accepts a text file containing instruction mnemonics as input, and outputs a binary file containing their corresponding bytecode representations. Instructions are provided for performing data acquisition, data manipulation, arithmetic operations, trigonometric operations, bitwise logic, branching, stack manipulation, subroutine calls, service calls, and time delays. For increased error handling flexibility, *error handlers* may be defined within each algorithm in the form of specially-labelled subroutines. Error handlers assume a label of the form *err[str]*, where *str* is the relevant error string. An example application that uses a portion of the VM instruction set may be seen in Table V.

VIII. LOGICAL LAYER

Physically or wirelessly connected TIMs may utilize the intelligence provided to them by platform-independent logical

TABLE V
SAMPLE LOGICAL MODULE TEMPLATE SPECIFICATION

Template Name	Generic Average
Algorithm Count	2

Algorithm Index	0
Algorithm Type	Primary

Algorithm Listing

```
sample:  wait #5000              ; wait 5 seconds
         membercount r0, #1      ; count for alg. 1
         beq r0, #0, sample      ; recheck if none
         waitforall #1           ; push alg. 1 readings
         mov r1, #0              ; r1 = 0
         mov r2, r0              ; r2 = member count
next:    pop r3                  ; r3 = next reading
         add r1, r1, r3          ; accumulate reading
         dec r0                  ; decrement members
         bgt r0, #0, next        ; continue if more
         div r1, r1, r2          ; calculate average
         setlogical r1           ; update reading cache
         jmp sample              ; repeat algorithm
```

Algorithm Index	1
Algorithm Type	Secondary
Assignment Limit	Unlimited
Assignment Type	Sensor
Assignment Class	Any
Assignment Return	float, double
Assignment Pose	Any
Failure Tolerance	Soft

Algorithm Listing

```
sample:  wait #5000      ; wait 5 seconds
         getlocal r0     ; get reading
         sendlocal r0    ; send to primary
         jmp sample      ; repeat algorithm
```

algorithms at the *logical layer* in order to function as a composite entity known as a *logical module*. The concept of a logical module abstraction was first proposed by Henderson and Shilcrat [21]. A TIM may be a member of up to 255 logical modules, each locally represented in memory as a structure and locally assigned an 8-bit index. The upper eight bits of the 64-bit TIM addresses are reserved to allow the mapping of a logical module index to that area. Thus, TIMs are effectively identified by 56-bit addresses.

Administration modules possess databases in which logical algorithms are stored. These logical algorithms are grouped into *logical module templates*, each of which represents intelligence that may be applied to a particular configuration of connected modules. Newly connected modules send information about the connection to nearby administration modules, which search their databases to determine if the constituent modules of the composite entity satisfy the assignment criteria outlined within any of the templates present. Matches are presented to the system user. In the absence of an administration module, connected modules are able to automatically request information about each other, and may assume a built-in, default behaviour based on the information obtained.

Each logical module template consists of a single *primary algorithm* and zero or more *secondary algorithms*. The specification of multiple algorithms per template facilitates the distribution of processing requirements among the members of a derived logical module. Since algorithms are able to be executed concurrently, modules are able to function as a member of multiple logical modules simultaneously. Typically, primary algorithms provide the majority of the intelligence of logical modules, while secondary algorithms complement and provide services to primary algorithms. Within a logical module template, six constants are associated with each secondary algorithm, described as follows:

- **Assignment Limit** — Indicates the maximum number of modules (a negative value signifies no limit) which may execute an instance of the secondary algorithm.
- **Assignment Type** — Indicates the type of modules (*sensor*, *actuator*, or *any*) to which the secondary algorithm may be assigned.
- **Assignment Class** — A list indicating the classes of modules (temperature, camera, speaker, display, linear actuator, etc.) to which the secondary algorithm may be assigned. A class of *any* signifies that the algorithm may be assigned to any class.
- **Assignment Return** — Indicates the data objects (*any* signifies any type) returned by the modules to which the secondary algorithm may be assigned.
- **Assignment Pose** — Indicates the pose type (*relative*, *absolute*, or *any*) and pose value (x, y, z coordinates and α, β, γ Euler angles) of the modules to which the secondary algorithm may be assigned. The pose of the primary module is used as the reference for relative poses. If pose type *any* is specified, the pose value is ignored.
- **Failure Tolerance** — Indicates the tolerance of the secondary algorithm (*soft*, in which normal operation is continued, or *hard*, in which operation of the entire logical module is halted) to the failure or departure of modules assigned to it. A secondary algorithm with soft failure tolerance may be automatically assigned to detected modules which meet its assignment criteria if its specified assignment limit has not been met.

An example of an simple averaging template for arbitrary types of simple sensors is shown in Table V. A logical module derived from this template would consist of a primary module which continually accepts and averages readings from a number of secondary sensor modules, each of which submits a new sample every five seconds.

The constituent modules of a logical entity maintain a local copy of the state and composite TEDS of the logical module as a structure in memory. A queue within the structure is used to store sub-structures containing the address, availability, and assigned algorithm of each member module. The module at the front of the queue is designated the *primary module*, which executes the primary algorithm, while the other modules are designated *secondary modules*, which execute secondary algorithms. The 64-bit address of the logical module is defined as the effective 56-bit address of the primary module, combined with the local 8-bit index it assigned to the logical entity. If the primary module is lost, it is removed from the queue and the next queued module assumes the role of the primary module. The former primary module deletes its copy of the logical

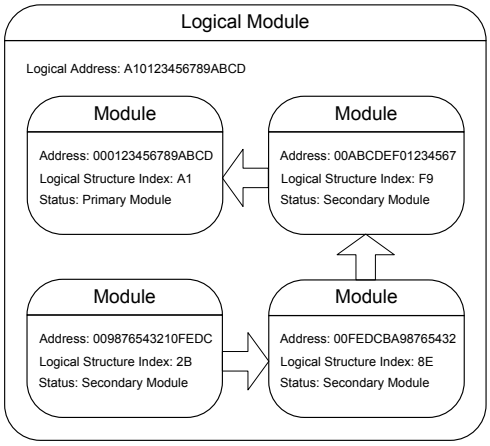

Fig. 3. Logical Module Block Diagram (values in base 16)

module structure if it rejoins the network.

An example of a logical module consisting of four physical modules is depicted in Fig. 3. The arrows point towards the front of the member queue. Module $000123456789ABCD_{16}$ is thus currently the primary module. The local logical module structure on the primary module was assigned the index $A1_{16}$, therefore the resulting address of the logical module is $A10123456789ABCD_{16}$. Should the primary module fail, entity $A10123456789ABCD_{16}$ will also be lost. Module $00ABCDEF01234567_{16}$ would then assume the role of the primary module, resulting in the appearance of a logical module with the address $F9ABCDEF01234567_{16}$. Secondary algorithms with soft failure tolerance will immediately utilize this new logical entity in place of the original.

IX. Conclusions and Future Work

In this paper, a software architecture and knowledge representation scheme that enables the combination of heterogeneous modular sensor and actuator components was described. This technique facilitates the creation of arbitrarily complex and flexible modular sensing systems, and will prove useful in many application domains.

The software architecture utilizes a pre-emptive RTOS, which provides concurrent foreground and background task execution and simplifies the implementation of the architecture. A VM-based middleware layer runs on top of the RTOS, enabling platform-independence of the algorithms which facilitate the collective operation of a set of modules.

Future work involves constructing TIM prototypes on which the performance of the software architecture may be evaluated. To allow the creation of logical algorithms using a higher-level representation than assembly mnemonics, the retargeting of an optimizing compiler, such as *lcc* [22], to the VM ISA will also be investigated.

Acknowledgments

The financial support of the Natural Sciences and Engineering Research Council of Canada (NSERC) is gratefully acknowledged.

References

[1] A. S. Tanenbaum, *Computer Networks*. Prentice Hall, 4th ed., 2003.

[2] J. L. Hill and D. E. Culler, "Mica: A wireless platform for deeply embedded networks," *IEEE Micro*, vol. 22, pp. 12–24, November 2002.

[3] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, "System architecture directions for networked sensors," in *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, vol. 34, (Cambridge, MA, USA), pp. 93–104, ACM, December 2000.

[4] L. E. Holmquist, H.-W. Gellersen, G. Kortuem, A. Schmidt, M. Strohbach, S. Antifakos, F. Michahelles, B. Schiele, M. Beigl, and R. Mazé, "Building intelligent environments with Smart-Its," *IEEE Computer Graphics and Applications*, vol. 24, pp. 56–64, January 2004.

[5] S. Cotterell, K. Downey, and F. Vahid, "Applications and experiments with eBlocks - electronic blocks for basic sensor-based systems," in *Proceedings 2004 First Annual IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks*, (Santa Clara, CA, USA), pp. 7–15, IEEE, October 2004.

[6] T. D. Ngo and H. H. Lund, "Modular artefacts," in *Component-Oriented Approaches to Context-aware Computing*, (Oslo, Norway), June 2004.

[7] K. Lee, "IEEE 1451: A standard in support of smart transducer networking," in *Proceedings of the 17th IEEE Instrumentation and Measurement Technology Conference*, vol. 2, (Baltimore, MD, USA), pp. 525–528, IEEE, May 2000.

[8] NXP Semiconductors, "LPC2141, LPC2142, LPC2144, LPC2146, and LPC2148 device highlight." [Online]. Available: http://www.standardics.nxp.com/products/lpc2000/lpc214x/, 2007 [Accessed: June 30, 2007].

[9] Nordic Semiconductor, "nRF24L01 preliminary product specification." [Online]. Available: http://www.nordicsemi.no/, March 2006 [Accessed: July 11, 2007].

[10] A. Jain and M. D. Naish, "Building blocks for adaptive modular sensing systems," in *Proceedings of the 2007 IEEE International Conference on Systems, Man and Cybernetics*, (Montréal, QC, Canada), October 2007.

[11] Y. Tiomkin, "TNKernel real-time kernel." [Online]. Available: http://www.tnkernel.com/, 2006 [Accessed: June 29, 2007].

[12] D. L. Mills, "Simple network time protocol (SNTP) version 4 for IPv4, IPv6 and OSI." [Online]. Available: http://www.faqs.org/ftp/rfc/pdf/rfc4330.txt.pdf, January 2006 [Accessed: March 4, 2007].

[13] J. Morrison, "EA IFF 85: Standard for interchange format files." [Online]. Available: http://www.szonye.com/bradd/iff.html, January 1985 [Accessed: March 4, 2007].

[14] R. Srinivasan, "RPC: Remote procedure call protocol specification version 2." [Online]. Available: http://www.faqs.org/ftp/rfc/pdf/rfc1831.txt.pdf, August 1995 [Accessed: March 13, 2007].

[15] A. Dunkels, B. Grönvall, and T. Voigt, "Contiki: A lightweight and flexible operating system for tiny networked sensors," in *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks (LCN 2004)*, (Tampa, FL, USA), pp. 455–462, IEEE, November 2004.

[16] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava, "A dynamic operating system for sensor nodes," in *Proceedings of the Third International Conference on Mobile Systems, Applications, and Services (MobiSys 2005)*, (Seattle, WA, USA), pp. 163–176, USENIX Association, June 2005.

[17] Sun Microsystems Inc., "Java technology." [Online]. Available: http://java.sun.com/, 2007 [Accessed: March 4, 2007].

[18] P. Levis and D. Culler, "Maté: A tiny virtual machine for sensor networks," in *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, vol. 37, (San Jose, CA, USA), pp. 85–95, ACM, October 2002.

[19] P. Stanley-Marbell and L. Iftode, "Scylla: A smart virtual machine for mobile embedded systems," in *Proceedings of the Third IEEE Workshop on Mobile Computing Systems and Applications*, (Los Alamitos, CA, USA), pp. 41–50, IEEE, December 2000.

[20] Python Software Foundation, "The Python programming language." [Online]. Available: http://www.python.org/, 2006 [Accessed: March 14, 2007].

[21] T. C. Henderson and E. Shilcrat, "Logical sensor systems," in *Journal of Robotics Systems*, vol. 1, pp. 169–193, 1984.

[22] C. Fraser and D. Hanson, "lcc: A retargetable compiler for ANSI C." [Online]. Available: http://www.cs.princeton.edu/software/lcc/, February 2007 [Accessed: March 4, 2007].