A Modular Real-Time Fieldbus Architecture for Mobile Robotic Platforms

Uluç Saranlı, *Member, IEEE*, Akın Avcı, *Student Member, IEEE*, and M. Cihan Öztürk, *Student Member, IEEE*

***Abstract*—The design and construction of complex and reconfigurable embedded systems such as small autonomous mobile robots is a challenging task that involves the selection, interfacing, and programming of a large number of sensors and actuators. Facilitating this tedious process requires modularity and extensibility both in hardware and software components. In this paper, we introduce the universal robot bus (URB), a real-time fieldbus architecture that facilitates rapid integration of *heterogeneous* sensor and actuator nodes to a central processing unit (CPU) while providing a software abstraction that eliminates complications arising from the lack of hardware homogeneity. Motivated by our primary application area of mobile robotics, URB is designed to be very lightweight and efficient, with real-time support for Recommended Standard (RS) 232 or universal serial bus connections to a central computer and inter-integrated circuit (I^2C), controller area network, or RS485 bus connections to embedded nodes. It supports automatic synchronization of data acquisition across multiple nodes, provides high data bandwidth at low deterministic latencies, and includes flexible libraries for modular software development both for local nodes and the CPU. This paper describes the design of the URB architecture, provides a careful experimental characterization of its performance, and demonstrates its utility in the context of its deployment in a legged robot platform.**

***Index Terms*—Distributed control, embedded systems, fieldbus, instrumentation architecture, mobile robots, real-time data acquisition, universal robot bus (URB).**

I. Introduction

COMPLEX embedded instrumentation systems have become an essential part of almost every imaginable application area in today's world. Regardless of whether such systems are deployed in commercial, technological, or scientific settings, there is an increasing need for a rapid development and revision cycle to decrease the time spent on integration problems already solved in numerous other similar situations. This requires architectural support for modularity and extensibility while also preserving real-time performance and long-term reliability.

For commercially successful and large-scale application domains such as computers and computing peripherals, general-purpose networking, instrumentation for process control, and the automotive industry, standardization has been achieved for many of the underlying technological components, resulting in a substantially accelerated development process. However, such standardization has not yet been possible for other fields that either do not yet have sufficient commercial outreach or are based on relatively new scientific discoveries. Among important examples are instrumentation applications such as biomedical and biomechanical applications [1], wireless sensor networks [2], manufacturing facilities [3] as well as home, office, or hotel automation applications [4].

Our focus in this paper is on the equally important area of autonomous mobile robotics, a rapidly emerging field for a number of important application areas such as search and rescue [5], humanitarian demining [6], space exploration [7], and other more specific domains including automated transportation (i.e., automated guided vehicles [8]) and miniature assembly [9]. A common aspect of all these application areas is their need to satisfy very strict weight and power constraints for autonomous operation in the field while incorporating a large array of sensory and actuation modalities to support a reliable and real-time interface to the physical environment. Moreover, the presence of numerous open research problems associated with these applications necessitates flexibility in the quantity and layout of available sensors and actuators. Currently, the lack of a compact, modular, and flexible instrumentational infrastructure that supports a wide variety of connectivity options while preserving real-time performance leads to the development of a new integrated design for every new robot platform. Such integrated custom designs substantially increase the time required for both the initial prototype development as well as further design iterations.

This paper introduces a new modular real-time instrumentation architecture, i.e., the universal robot bus (URB), to address these problems both for the specific domain of autonomous mobile robots as well as other similarly structured domains. Inspired by, but substantially extending the earlier RiSEBus [10] protocol, the new architecture that we describe in this paper incorporates the following three novel contributions.

1) The URB architecture and protocols provide strict real-time performance guarantees (with respect to both deterministic latency and bandwidth) in the presence of a heterogeneous collection of connectivity alternatives such as the Recommended Standard (RS) 232, RS485, universal serial bus (USB), industry standard architecture (ISA), inter-integrated circuit (I^2C), and controller area network (CAN) protocols. Existing architectures are either very

Manuscript received March 10, 2010; revised May 26, 2010; accepted July 19, 2010. Date of publication November 11, 2010; date of current version February 9, 2011. The works of A. Avcı and M. C. Öztürk were supported by the Scientific and Technological Council of Turkey (TUBITAK) under Project 109E032. The Associate Editor coordinating the review process for this paper was Dr. Antonios Tsourdos.

The authors are with the Department of Computer Engineering, Bilkent University, Ankara 06800, Turkey (e-mail: saranli@cs.bilkent.edu.tr; akinavci@gmail.com; m.cihanozturk@gmail.com).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TIM.2010.2078351

rigid in their support for connectivity protocols [11], [12] or cannot provide real-time performance guarantees for high-bandwidth operation [13]–[17].

2) The URB framework enables modularity and flexibility in the deployment of small-scale but complex instrumentation systems through the provision of a standardized set of application programming interfaces (APIs) and associated interface libraries, significantly accelerating both initial development and later modification while still satisfying associated performance requirements. This results in a level of hardware/firmware modularity similar to what unified robotic programming architectures, such as ROS [18], OROCOS [19], and Microsoft Robotics Studio [20], try to achieve for high-level robot programming.
3) URB protocols allow accurate automatic synchronization of instrumentation components distributed within a system, ensuring consistency and synchrony of sensory acquisition and actuator output.

The relevance of the URB framework to the instrumentation and measurement community comes from the fact that all of these features are needed for real-time discrete control systems where accurate measurements (both in terms of timing and sensory accuracy) and quick response to rapidly changing environmental conditions are necessary. Although we use the development of a small mobile robotic platform to illustrate the utility and performance of our design, URB is equally suited for use within similarly structured instrumentation problems such as wearable computing, biomechanical measurement systems or sensor networks. Our review of existing literature in Section II also shows that none of the existing fieldbus architectures in the literature provide a comparable degree of flexibility and real-time performance under a similarly unified design.

The rest of this paper is organized as follows. In Section II, we summarize existing work in this area. We then give an overview of the URB framework in Section III, followed by a description of API components for developers using the URB framework in Section IV. Section V describes novel technical aspects of URB in how deterministic performance and node synchronization is achieved, followed by a careful experimental characterization of performance in Section VI. Section VII concludes this paper.

II. Background and Existing Work

In general, instrumentation architectures with distributed control and data acquisition elements are called *fieldbuses* [21]–[23]. Despite this seemingly unifying definition, there has been numerous application-specific fieldbus designs with different strengths and weaknesses [11]. Among these, the Foundation fieldbus is probably one of the most widely used and standardized alternative [13], [22]. It is widely adopted by numerous commercial field devices and many off-the-shelf interfacing board and component alternatives are available to the designers of instrumentation systems. As a result of this wide adoption, significant research effort has also been devoted to its real-time applications, characterizing and improving predictability of communications [24], [25] and the impact of delays on closed-loop control [26]. Unfortunately, size, power, and weight constraints associated with the design of autonomous mobile robots are much more severe and limiting than those associated with domains in which existing large-scale fieldbus alternatives are intended to be used.

Nevertheless, our domain also exhibits many constraints paralleling problems that led to the development of the fieldbus concept. Most importantly, there is a clear need for a modular and extensible interfacing standard since the use of custom interfaces to every sensor or actuator component impairs extensibility and increases the cost of revisions [16], [22]. In addition to such practical advantages, the provision of a modular instrumentation architecture was also shown to yield substantial gains in the deployment of novel control strategies for robotic platforms [27]. Advantages of similar modularity features were also described in [28], where a possibly heterogeneous set of remotely positioned embedded devices were accessed through a standardized general packet radio service interface to yield a distributed data acquisition system. Finally, the use of shared communication buses adopted by fieldbus systems results in substantially reduced cabling, eliminating one of the most common sources of failure for rapidly moving mobile robot platforms with severe vibrations and other physical disturbances [11].

Since simplicity and deterministic performance are critical for our application domain, physical layer protocols such as the Ethernet [29], [30], designed primarily for non-real-time local and wide area network deployments, are not appropriate for our use. Similarly, the relatively high bandwidth requirements of fast mobile robotic platforms, where control loops typically operate at 1 kHz, preclude the use of slower fieldbus protocols such as Profi-Bus [14], LON [15], [16], BAC-Net [17], ModBus, and FIP. Combined with our need to modularly interface with even very simple miniature microcontroller-based sensor and actuator nodes, direct adoption of most existing fieldbus designs, hence, becomes very difficult.

In this context, one of the more appropriate physical layer protocols is the CAN, a two-wire serial communication bus standard originally developed for the automotive industry to reduce cabling cost and provide immunity to electrical noise [12], [31]. The CAN protocol has been successfully adopted for robotics applications, delivering both real-time performance [32], and modularity properties [12] that we observed were necessary for mobile robotic platform designs. However, *exclusive* adoption of the CAN protocol for the physical layer substantially constrains the usable selection of computing components to only high-end relatively large options that may be inappropriate for miniature sensory interfaces. Our goal is to provide support for as many different connectivity options as possible while maintaining the same modularity and real-time performance properties. A similar argument precludes exclusive adoption of the USB [33], further disadvantaged by the significant software development effort associated with USB devices and their associated operating system drivers. Nevertheless, we recover the utility of both of these protocols through their optional use within a URB system.

III. Overview of the URB Architecture

Physically, the URB is a two-tiered architecture with support for a heterogeneous collection of connectivity standards. As

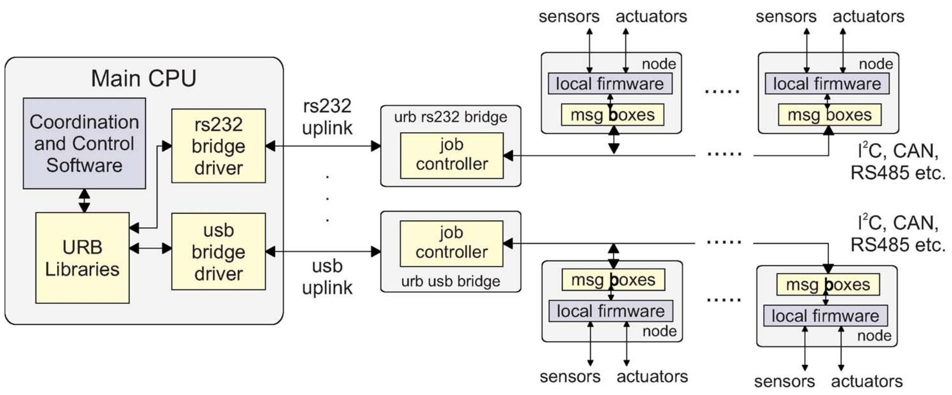


Fig. 1. Physical topology of a URB system. CPU–node connectivity is established through *bridges*, each of which controlling a single downlink bus shared among associated nodes and communicates with the CPU through a dedicated uplink connection.

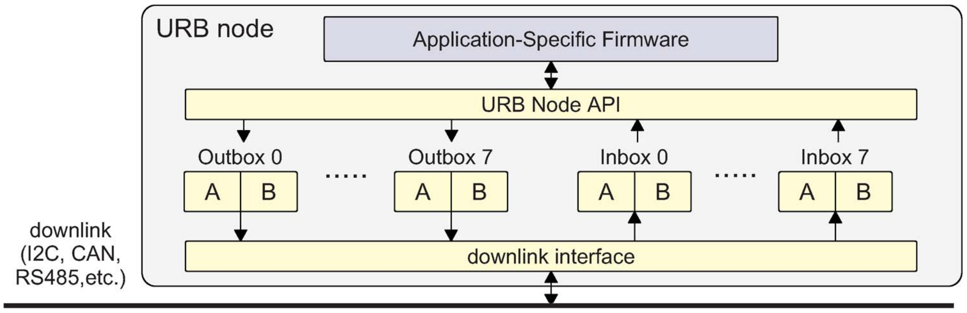


Fig. 2. Logical view and firmware structure of a URB node. A total of 16 (eight inputs, eight outputs) double-buffered message boxes are supported by each node.

shown in Fig. 1, the first tier consists of *URB bridges* connected to a central processing unit (CPU) through a variety of protocols such as RS232, USB, ISA, and peripheral component interconnect. We call the connection between a bridge and the CPU the *uplink*. In contrast, the second tier consists of application-specific *URB nodes* each connected to one of the bridges through a simple shared bus protocol such as I^{2}C, CAN, or RS485. The connection between a bridge and its nodes is called the *downlink*.

The novelty of our design lies in the provision of a standard set of bridge implementations for each uplink/downlink protocol pair, together with a standard set of libraries for both the CPU software and node firmware to provide a *logically centralized* abstraction, wherein each node appears to be directly accessible from the CPU with intermediate bridges handling the necessary translation between different protocols and physical connectivity standards. The biggest challenge in the realization of this architecture is the design of both the low level protocols associated with each connectivity alternative and the associated software libraries to support reliable real-time operation with the necessary modularity properties. Nevertheless, the resulting logical node access model admits reusable node and CPU software components, allowing rapid deployment of new systems with minimal design overhead while preserving real-time performance. In summary, the URB architecture incorporates the following key features:

1) transparent support for a wide variety of connectivity protocols used within the same system;
2) deterministic real-time performance with predictable latency, high bandwidth, and low protocol overhead;
3) automatic synchronization of local node heartbeats across nodes on a single downlink;
4) automatic discovery and identification of nodes;
5) standard APIs for rapid development of both URB node firmware and associated CPU software.

The first three items above represent major contributions of this paper. Subsequent sections will detail associated architectural details and experimentally characterize the performance of our design.

IV. Developer's View of the URB Architecture

A. Logical View of a URB Node

To achieve flexibility and modularity, URB adopts a *logically centralized* communication model that is inspired from the USB [33], where a collection of *endpoints* provide structured communication channels with the CPU. As such, each URB node provides 16 double-buffered *message boxes* (eight inboxes and eight outboxes) with individually configurable fixed sizes of up to 32 bytes available to application programmers. As shown in Fig. 2, application firmware on a URB node can use the URB node API to access these message boxes with all low-level downlink communications handled by the URB node libraries.

Two of these message boxes, i.e., outbox 0 and inbox 0, are reserved for sending node identification information to the CPU and receiving protocol commands from the CPU, respectively. Outbox 0 is of particular importance since, as shown in Fig. 3, it allows each node to be uniquely identified by an 8-bit *class*,

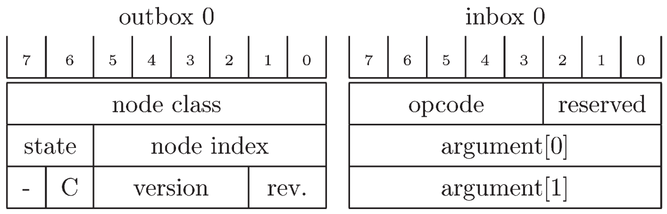


Fig. 3. Left: Outbox 0 used to identify the URB node. Right: Inbox 0 used to send custom commands to the node.

encoding the functionality of the node, and a 6-bit *index*, distinguishing multiple nodes with the same class attached to the same URB system. This structure allows automatic discovery of nodes and runtime queries of node status.

B. Application Software on URB Nodes

To facilitate the development of application-specific nodes, URB provides a standard node API. First, a set of methods is provided to initialize the class and index of a node as well as set fixed message box sizes and access their contents as shown on the left column in Fig. 4. More importantly, however, the API expects the application firmware to instantiate functions listed on the right column of Fig. 4, implementing key functionality necessary to interface with the URB framework as well as application-specific features.

The functions `app_bootup`() and `app_init`() are called when the node is powered up and `app_reset`() is invoked when the node receives a reset command on inbox 0. The URB node library internally implements the main round-robin loop and calls `app_idle`() at every iteration of this loop (not necessarily periodically), allowing the application to do polling. In contrast, `app_update`() is periodically invoked and can be used by the application to implement periodic tasks. The preferred frequency of invocation for this update function is 1 kHz, but any other frequency can also be chosen by the application programmer through appropriate clock and timer configurations on the node microcontroller during firmware initialization. As described in Section V-E, the URB framework also allows calls to `app_update`() to be synchronized across nodes, making it possible to ensure timing consistency across multiple nodes provided that the initial update frequencies of all nodes are similarly configured. Finally, `app_signal`() is called with a signal identifier upon reception of an associated command on inbox 0 and calls to `app_fault`() inform the application of fault conditions. This simple node API provides flexibility while allowing rapid deployment of simple node applications.

C. Application Software on the URB CPU

As described above, message boxes on each node provide structured communication channels between the CPU and URB nodes. Information on the size and structure of each message box has to be shared by both the CPU software and node firmware for which the mechanism is left unspecified by the protocol. Developers can either ensure consistency through shared header files or utilize a dedicated message box for this purpose. Nevertheless, in the presence of this shared knowledge, the URB CPU library, currently implemented in C++, provides a simple means by which nodes attached to the system can be identified and associated message boxes can be accessed.

The singleton `URBInterface` class provides the entry point of the API. During initialization, application software is expected to invoke its `addBusManager`(...) method to register each bridge connected to the system, through manager class instances associated with corresponding types of uplink (i.e., `USBManager`, `RS232Manager`, etc.). This process also initiates node discovery on each downlink bus and identifies all nodes connected to the system. Following this initialization and discovery, the `findNode`(...) method can be used to locate a node with a specific class and index, returning a `NodeAccessor` class instance through which associated message boxes can be accessed. The methods `newRequest`() and `submitRequest`() can then be used to initiate URB transmissions, with user-specified callbacks invoked upon completion.

Fig. 5 illustrates a simple program which periodically reads from a specific URB node. It should be noted that access to a node is independent of which bridge it is connected to and what type of downlinks are in use. Callbacks are asynchronously invoked by bus managers upon reception of requested data and allow modular processing of data flow between different nodes and associated software components. In case of a failure, either in the form of a checksum mismatch or a downlink communication timeout as described in Section V-D, the callback function can check error flags in the response packet to identify the source of the problem and proceed accordingly. No automatic retransmission attempts are done by the URB framework to ensure predictable performance. Naturally, real-time performance critically depends on the implementation details of specific bus managers and associated low level uplink and downlink protocols, which we will detail in the subsequent sections. However, this API provides the level of modularity that we observed to be necessary for rapid deployment and revision of instrumentation within mobile robotic platforms.

V. Internals of the URB Architecture

A. Uplink and Downlink Communication Protocols

As a result of the heterogeneous use of physical connectivity standards within the URB framework, each different type of uplink and downlink connection implements its own physical and transport layer protocols. Nevertheless, URB enforces a common structure on the link-control layer protocols which we summarize in this section.

Taking place across dedicated connections, uplink communications consist of *request* packets addressed to a node on the corresponding downlink bus. Request packets encode a downlink-specific *address* field for the targeted node, together with the message box number to be accessed and its size. Requests are processed in the order that they are received by the bridge, which performs necessary communications on the associated downlink and prepares a response packet, either with the contents of the outbox to be read, or a success flag for writes to a node inbox. Fig. 6 illustrates uplink request and response packet formats.

Functions provided by the API	Functions to be implemented by the application
`URB_SetClass(...)`	`app_bootup()`
`URB_SetIndex(...)`	`app_init()`
`URB_SetupInbox(...)`	`app_reset()`
`URB_SetupOutbox(...)`	`app_idle()`
`URB_LockInbox(...)`	`app_update()`
`URB_ReleaseInbox(...)`	`app_signal(...)`
`URB_LockOutbox(...)`	`app_fault(...)`
`URB_ReleaseOutbox(...)`	

Fig. 4. Left: Functions provided by the URB node API. Right: Functions to be implemented by the application firmware.

```
bool read_callback( NodeRequest *req ) {
  char * data = req->getData();
  // Process message box contents
  return true;
}


void main() {
  URBInterface *intf = URBInterface::instance();
  intf->addBusManager( new USBManager( "/dev/urb_usb0" );
  intf->addBusManager( new RS232Manager( "/dev/ttyS0" );

  NodeAccessor *node = intf->findNode( CLASS, INDEX );
  while (1) {
    NodeRequest *req = node->newRequest();
    req->read( MBOX_NUM, MBOX_SIZE );
    req->setCallback( read_callback );
    node->submitRequest( req );

    sleep(1);
  }
}
```

Fig. 5. Example program that periodically reads data from a given outbox on a specific URB node.

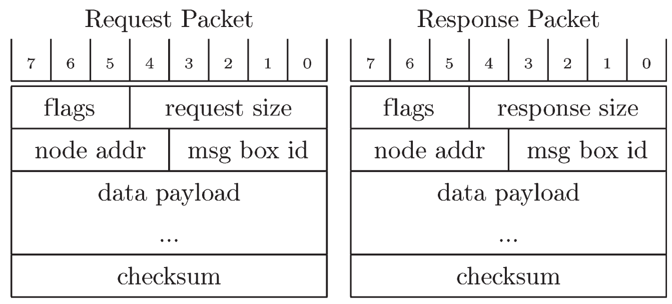


Fig. 6. (Left) Request and (right) response packet formats for uplink and downlink communications.

Downlink communications mirror the structure of the uplink protocol, except with the request size and flag fields removed since transport layer protocols for I^{2}C, CAN, and RS485 provide necessary mechanisms to implicitly infer packet size. Note that protocol overhead is minimized for both uplink and downlink communications, allowing maximum channel utilization and increased bandwidth.

B. RS232 Uplink Implementation

We have designed and implemented a URB bridge with support for RS232 uplink connections that can be used for low-to-medium-bandwidth applications and easy testing of simple node implementations, as illustrated in Fig. 7. Since RS232 is widely available on all desktop and small embedded computers, URB support for an RS232 uplink provides a very versatile benchtop testing facility for node development.

Since the universal asynchronous receiver/transmitter (UART) standard is an asynchronous point-to-point protocol,

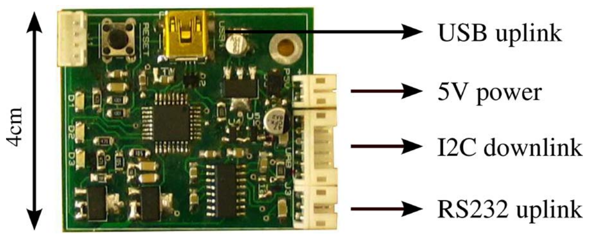


Fig. 7. URB bridge prototype with support for both RS232 and USB uplink connections. Current prototype measures 4 × 5 cm to facilitate debugging, but a much smaller version is in development.

no extensions to the common protocol described in Section V-A is needed. URB RS232 bridges simply buffer incoming request packets from the uplink, process them in the order that they are received, performing necessary downlink transactions with the appropriate downlink protocols, and immediately send the corresponding response packets back to the CPU. The `RS232Manager` class instance on the CPU uses separate read and write threads to asynchronously service requests and their responses. As described in Section VI-B, this yields very low latencies, determined primarily by the UART baud rate and the downlink speed.

C. Real-Time USB Uplink Implementation

In this section, we describe how we can use a high-speed USB connection as the physical connectivity layer of a URB uplink connection. This task entails the following two major challenges: 1) obtaining predictable real-time behavior (primarily in terms of determinacy) through careful management of USB host controller behavior and 2) minimizing latency by proper scheduling of USB request blocks for uplink packet transmission. The methods we propose in this section can achieve both of these goals and represents a novel approach applicable to other real-time applications for the USB standard.

In general, interfacing with USB devices requires an associated kernel driver on the CPU. As shown in Fig. 8, our design involves joint development of a kernel driver that communicates with the bridge through the host controller, together with the `USBManager` class, which controls buffering and delivery of URB requests through the uplink.

During high-speed operation, the USB host controller accepts *request blocks* from kernel drivers and schedules associated transactions within 1-ms time slots [33]. The time at which the actual transmission takes place depends on a variety of factors, including the type of the transmission (control, interrupt, bulk, or isochronous) as well as the existing load associated with other devices on the same bus. Moreover, unlike the bidirectional nature of RS232 communications, the host controller is in charge of initiating all transactions in a USB system, making it impossible for the bridge (which acts as a USB slave device) to signal its need for sending back response packets.

A timeline of activities for our URB USB uplink implementation is illustrated in Fig. 9. First, to maximize bandwidth, a thread created by the `USBManager` class buffers node requests from the application software and sends them to the kernel driver *once* every 1 ms, matching the scheduling frequency of the host controller. This ensures that only one request block, with as much payload as possible, is sent to the host controller at every cycle (assuming no non-URB devices are attached to the same host controller), ensuring both determinacy and maximizing the amount of data that can be sent to the bridge.

Upon reception of this write request, the kernel driver bundles it into a USB request block and issues a write request to the USB host controller, which schedules it for transmission at the next USB cycle. The completion of this request is followed by the invocation of a write callback in the kernel, which then prepares an associated read request block and submits it to the host controller for retrieval of USB bulk response packets. The end of this sequence is marked by the invocation of the read callback function, which then signals the read thread to retrieve and buffer response packets. This carefully scheduled USB data exchange mechanism, coordinated through operating system facilities such as spin locks and wait queues, ensures that the worst-case latency from the time a request is issued to the reception of its response is 3 ms, which is confirmed by our experimental results presented in Section VI-B. Part of this latency, i.e., 1 ms, is caused by the operating system scheduling period for the write and read threads in addition to the 2-ms USB latency detailed in Fig. 9.

A very important feature of our USB uplink implementation is that it allows two transactions to be simultaneously active (e.g., write n and read $(n-1)$ as in Fig. 9), making it possible to utilize USB bandwidth for both a read and a write operation during every 1-ms cycle. Waiting for the write–read sequence to be finished before initiating the next write operation would have wasted 50% of the bandwidth, with further implications on the allowable control loop frequency. Further details on our URB USB uplink implementation can be found in [34].

D. I^{2}C Downlink Implementation

I^{2}C is a two-wire synchronous serial protocol that can achieve speeds exceeding 1 Mb/s with proper bus termination [35]. Its relatively high speed and universal availability in almost all microcontrollers make it a very attractive option for URB downlink communications. Its simplicity compared to more robust but complex protocols such as CAN motivated our prototype downlink instantiation to use the I^{2}C as its underlying physical layer.

Challenges in this adoption primarily arise from the relatively noise-prone nature of I^{2}C communications, particularly in the presence of bidirectional isolator components on the data and clock lines. Since these lines rely on open-drain interfaces, a problematic node can result in the entire bus to become stuck at a low digital level, making any further communication impossible. Our I^{2}C downlink implementation addresses this issue by proper timeout mechanisms that issue a bus reset when such conditions are detected. The inclusion of a checksum field in the packet format of Section V-A also increases robustness by enabling detection of erroneous data transmission. All results presented in Section VI use this I^{2}C downlink implementation, for which further details can be found in [36]. Note, however, that the URB framework allows the use of any shared bus standard for its downlink connections if associated bridge and node libraries are implemented.

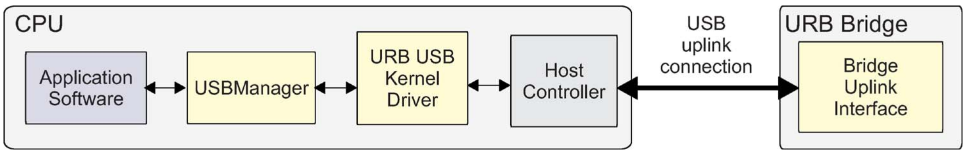


Fig. 8. Organization of the URB USB uplink implementation.

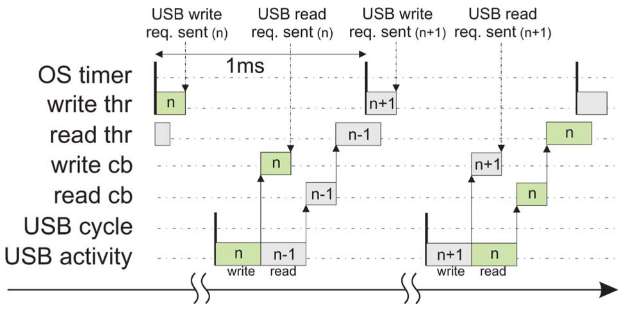


Fig. 9. Timeline of activities for `USBManager` read and write threads, kernel callback functions within the USB driver, and physical USB communications. Request n is sent by the write thread to the kernel driver, which then receives the associated write callback. This callback issues a read request, which is serviced in the next USB cycle and ends with the invocation of the read callback. This wakes up the read thread, which retrieves response packets from the kernel driver.

E. Synchronization of Node Heartbeat Clocks

One of the most important real-time features for a robotic platform is the deterministic accuracy of relative timing for data acquisition and command update across different sensors and actuators in the system. When instrumentation and computation associated with these components is physically distributed on the robot, maintaining this accuracy becomes increasingly difficult [37]. Similar problems have most widely been studied in the context of clock synchronization for multiprocessor systems and sensor networks and can be categorized into *convergence*-, *agreement*-, and *diffusion*-based algorithms [38]. The first two types of algorithms require access to the internal clocks of other components in the system, introducing substantial communication overhead and, hence, is not directly suitable for our architecture. Similarly, diffusion-based algorithms rely on direct communication between processing units [39], which is structurally incompatible with the URB framework where nodes are configured as slaves.

In this section, we describe a novel synchronization algorithm compatible with the topology and operation of the URB framework that allows synchronous data acquisition across all nodes connected to a single shared downlink connection. Our approach closely parallels the receiver–receiver clock synchronization algorithms for sensor networks [40], [41] and relies on the broadcasting of a *SYNC* message on the shared bus, to which local heartbeat timers on each node lock onto using period and phase feedback and ideas from phase-locked loops [42].

Fig. 10 illustrates the local synchronization algorithm used by each URB node to tune its internal heartbeat clock. First, during each node update cycle, we estimate the period of the incoming downlink heartbeat T_d with a simple exponential filter as follows:

$$\hat{T}_d[k+1] = \hat{T}_d[k] - K_d\left(\hat{T}_d[k] - T_d\right) \qquad (1)$$

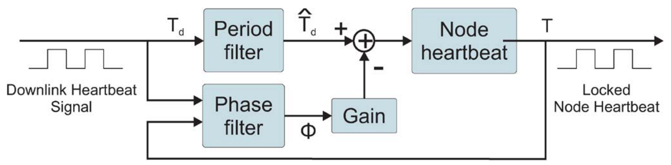


Fig. 10. Local phase-locked loop for the node heartbeat clock. A SYNC message broadcast on the downlink is used as a global clock.

where T_d denotes the measured period of the downlink heartbeat, $\hat{T}_d$ denotes its local estimate, and K_d determines the filter time constant. We then use this smoothed estimate as the period of the local clock, which is adjusted with a feedback term to match its phase to the incoming downlink heartbeat. The computation of the local timer period T on the node, hence, takes the form

$$T[k+1] = \hat{T}_d[k+1] - K_n(t_n - t_d + \Delta t_{\text{adj}}) \qquad (2)$$

where t_n and t_d denote the measured times for the latest node and downlink heartbeat signals, whereas Δt_{adj} is a constant offset that can be used to impose a desired phase lead on the local clock.

In this context, we use the idea of allowing a fixed phase lead Δt_{adj} on the local node heartbeat to substantially decrease data acquisition latency. Consider the scenario illustrated in Fig. 11, where two URB nodes are attached to a single bridge. When the internal update cycles of the two nodes and the data requests from the bridge are not synchronized, at the worst case, data sent back by a node can be up to 1 ms old as is the case with the second data request in Fig. 11. This results in a worst-case increase of 1 ms in data latency.

In contrast, when both nodes are synchronized with the bridge heartbeat, and a sufficient phase lead is introduced, freshly acquired data components can be sent as a response to bridge requests, as illustrated in Fig. 12. The maximum time spent on the node for data acquisition and buffer updates t_{acq} can easily be measured by the node and then used as the desired phase lead with $\Delta_{\text{adj}} = t_{\text{acq}}$ to yield the desired behavior. Different nodes with different values of t_{acq} will adjust themselves accordingly, ensuring synchrony between the end of acquisition at every node and the reception of data with minimum latency by the bridge. Section VI-C details our experimental results on the performance and accuracy of this algorithm. Note also that the same method can be used to synchronize the heartbeat signals for bridges within a URB system, making it possible to synchronize not only nodes one a single downlink, but all nodes connected to the system.

VI. Platform Experiments and Performance

In this section, we present our experimental results on the performance of an *entire URB system*, including the CPU

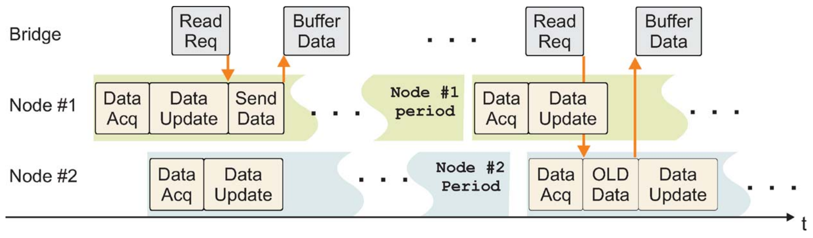


Fig. 11. Timeline of data acquisition and communication transactions for two *unsynchronized* nodes.

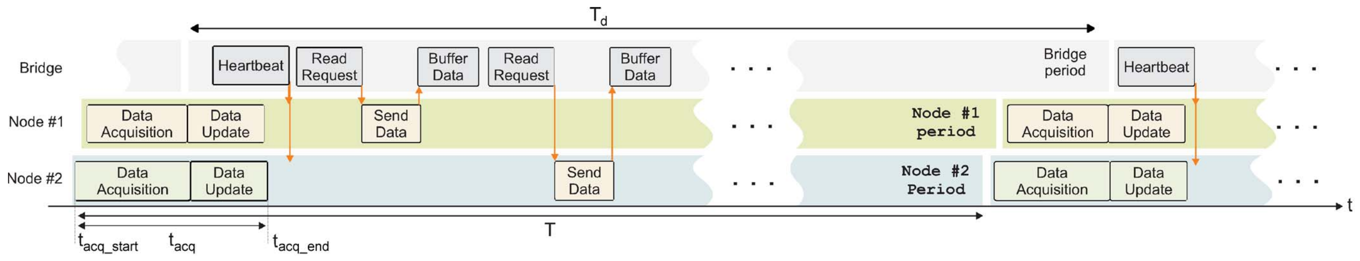


Fig. 12. URB system with one bridge and two nodes with synchronization enabled. Data acquisition on all nodes finishes simultaneously and right before the downlink heartbeat is received. Latency from the end of data acquisition to the shipment of data is minimized.

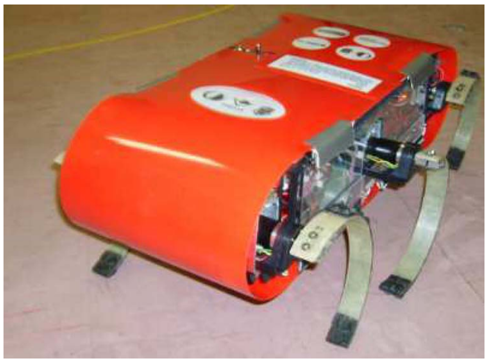


Fig. 13. SensoRHex hexapod robot platform.

software, the uplink connection, the bridge firmware, the downlink connection, and the node firmware, operating at a main loop frequency of 1 kHz. To also illustrate the modularity properties of the system, all of our experiments will be presented in the context of the SensoRHex robot platform, which uses URB as its instrumentational infrastructure.

A. SensoRHex *Hexapod Platform*

RHex is a power and computationally autonomous hexapod robot that has been capable of dynamic locomotion at very high speeds over very difficult rough terrain [43]. However, its relatively old centralized electromechanical design has made it very difficult to go beyond basic behaviors with substantial effort required to incorporate new sensors and revised actuator modules [44]. We have used the URB architecture to design SensoRHex, as shown in Fig. 13, preserving the morphology of RHex but replacing the computational infrastructure with a much more modular and extensible design. In the following sections, we will both describe how the URB architecture has been used to realize this design and present experimental performance figures for components we described in the scope of this paper.

The SensoRHex platform consists of six legs, each attached to a gearhead/motor/encoder unit (see Fig. 17). Each dc motor is controlled by a URB-enabled controller node, which also measures the armature voltage and current as well as the motor case temperature and interfaces to a Hall effect sensor for calibration. The computational infrastructure incorporates two PC104 stacks each with a 500-MHz AMD Geode CPU card. Both CPU units run stripped-down Linux installations with 1-kHz tick frequency. The robot has three onboard Li-Poly batteries, managed by custom electronics that also serves as a URB node and reports voltage, current, and battery status information to the CPU. Finally, there is a small inertial measurement node and an infrared distance measurement node attached to the system as well.

Fig. 14 illustrates the structure and layout of components on the SensoRHex platform. Two URB USB bridges interface with three motor controller nodes each, ensuring that high bandwidth data transfer can be realized. In contrast, low-bandwidth nodes, including the battery management, inertial measurement, and infrared sensor nodes, are attached to the main CPU through a URB RS232 link. This architecture is readily extensible with either the addition of new bridges or by attaching additional nodes to existing downlink buses, provided that their bandwidth requirements do not exceed the associated limits.

Due to the highly dynamic nature of the robot platform, motor control nodes need to be accessed at a minimum frequency of 1 kHz to ensure stability of closed-loop behaviors. In contrast, nodes attached to the RS232 can operate at 100 Hz since the time constants of associated sensors are rather large and filtering can be done on the nodes. Based on these requirements, we configured the USB uplink connections to operate in high-speed mode with 10 Mb/s raw bandwidth and the RS232 uplink to use 115 200 Bd since this was the highest frequency supported by the microcontrollers. The downlink connection for all three buses operates at 400 kb/s due to the presence of isolator chips and large amounts of noise coming from the motor amplifiers. It is important to note that the architectural

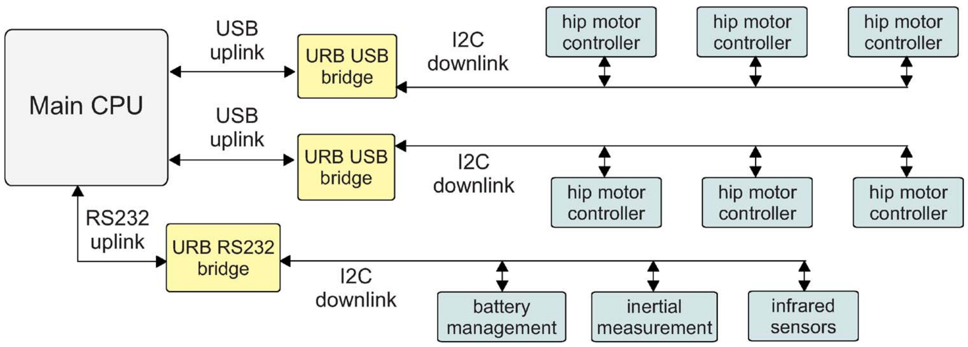


Fig. 14. Structure of the computational infrastructure on SensoRHex.

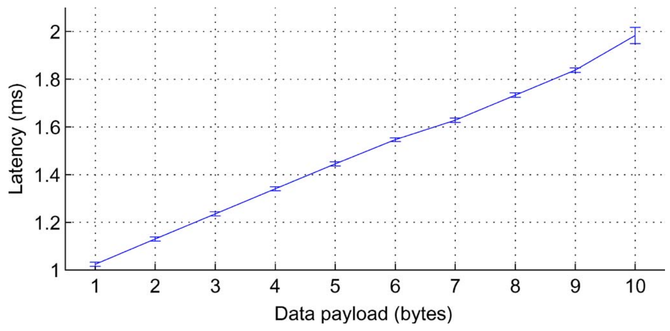


Fig. 15. Average round-trip (from URB request submission to callback invocation) latencies for 1000 read requests across an RS232 uplink operating at 115 200 Bd and an I^2C downlink at 400 kb/s as a function of data payload with a loop frequency of 1 kHz.

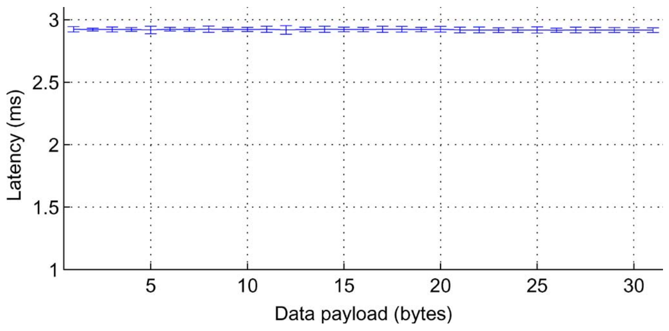


Fig. 16. Average round-trip (from URB request submission to callback invocation) latencies for 1000 data read requests across a USB uplink and an I^2C downlink at 400 kb/s as a function of data payload with a loop frequency of 1 kHz.

support that we provided for different uplink and downlink connectivity options within the URB architecture allowed us to fine tune the computational infrastructure according to our requirements.

B. Bandwidth and Latency

To obtain general performance figures for our uplink and downlink implementations, we first characterize in this section the latency and bandwidth limitations of our RS232 and USB uplink implementations connected to a test node with configurable message box sizes.

Fig. 15 illustrates *API level* average round-trip latencies for 1000 periodic read requests across an RS232 uplink connection at 115 200 Bd with a loop frequency of 1 kHz for different data payload sizes. The RS232 communication speed limits the uplink packet length to 12 bytes within a millisecond, corresponding to a maximum data payload of 10 bytes. Our implementation has been able to achieve this performance with a maximum latency of 2 ms (half of which is caused by the operating system thread scheduling frequency). The linear increase in latency as a function of the data payload is a direct result of the asynchronous nature of RS232 communications, with the request callback being invoked immediately after transmission is finished. Note that larger payload sizes are possible if the loop update frequency is reduced.

In contrast, the USB uplink implementation allows much higher bandwidth, allowing a full size packet with 31 bytes of data content transmitted within a single loop period of 1 ms. As shown in Fig. 16, the USB uplink provides a constant latency of 3 ms as predicted by the analysis of Section V-C. When the USB uplink is used, the I^2C downlink connection and the bridge firmware become bottlenecks for the maximum bandwidth that can be achieved. Note, however, that both the 3-ms latency and the uplink bandwidth are preserved when multiple bridges are connected to the same hub, making the USB uplink very modular and extensible.

C. Synchronization Accuracy

To characterize the performance of the synchronization algorithm, we proposed in Section V-E, we will use three different metrics. First, the *initial convergence delay* is defined as the time it takes for all nodes to converge to the same heartbeat period and phase. Second, the *internode heartbeat difference* is defined as the time difference between any two nodes on the same bus. Finally, *single-node jitter* is defined as the variation in the heartbeat period of a single node. Note that an important factor in all of these figures in the nondeterminacy associated with data acquisition on each node since our algorithm attempts to synchronize the *end of* data *acquisition* rather than just the heartbeat timers themselves.

Table I summarizes our results, obtained with 20 independent experiments on a single I^2C downlink with two nodes. Times of key transition events were measured with an oscilloscope. These figures show that the synchronization algorithm performs very well, with fast convergence and steady-state synchronization errors below 1 μs.

D. Application: High-Bandwidth DC Motor Control

In this section, we describe our final set of experiments for high-bandwidth closed-loop control of SensoRHex's hip motor

TABLE I
PERFORMANCE METRICS FOR NODE SYNCHRONIZATION FOR 20 EXPERIMENTS MEASURING KEY TRANSITIONS WITH AN OSCILLOSCOPE

Metric	Average	Max
initial convergence delay	78.4 ms	88.4 ms
inter-node difference in t_{acq_end}	$0.19\mu s$	$0.56\mu s$
single-node jitter on t_{acq_end}	$0.92\mu s$	$1.33\mu s$

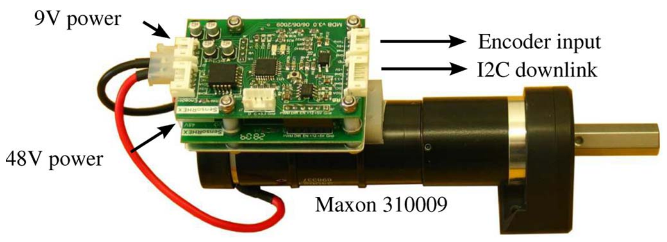


Fig. 17. URB-capable motor driver unit. Our prototype provides encoder, voltage, and current feedback and allows driving dc motors with up to 10 A continuous, 20 A peak current. The unit is shown mounted on one of SensoRHex's hip motors.

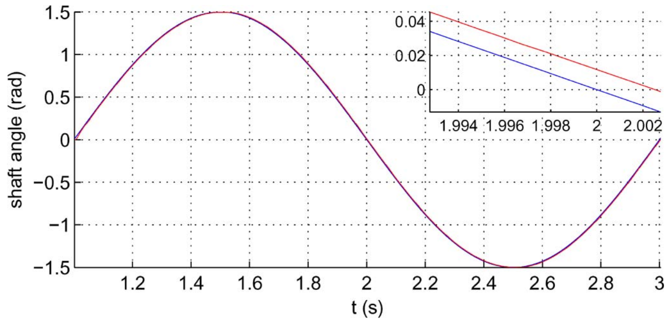


Fig. 18. Closed-loop PD motor control performance for a 0.5-Hz sinusoid reference input (blue line) and an RS232 uplink connection. The magnified plot at the top right corner shows accurate tracking with only a 2-ms lag.

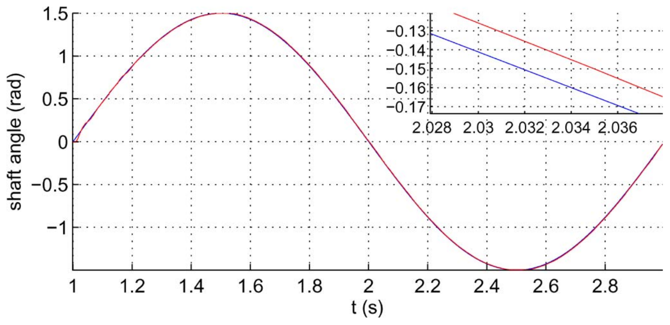


Fig. 19. Closed-loop PD motor control performance for a 0.5-Hz sinusoid reference input (blue line) and a USB uplink connection. The magnified plot at the top right corner shows accurate tracking with only a 3-ms lag.

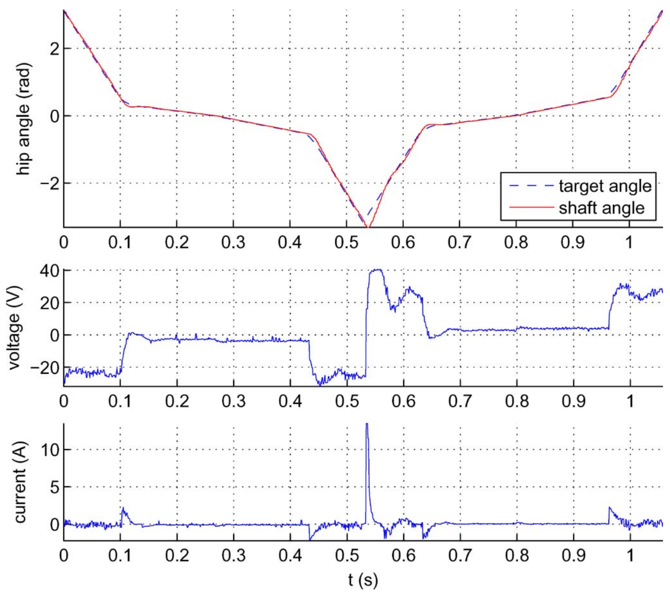


Fig. 20. (Top) Motor shaft angle and (middle and bottom) armature voltage and current measurements from one of SensoRHex's hip units during two steps (one forward and one backward) of an alternating tripod gait under 1-kHz closed-loop PD control through a URB USB uplink connection. The direction of rotation for the leg was changed around $t = 0.53$ s to illustrate transient currents.

units illustrated in Fig. 17. Using the URB node API, the motor driver unit provides two message boxes: Outbox 1 provides high bandwidth sensory information (6 bytes data payload), and inbox 1 accepts motor voltage commands (2 bytes data payload). Normally, the driver node is also capable of performing closed-loop control directly, but we use the CPU to close the proportional–derivative (PD) control loop for the results we present in this section to characterize the performance of the entire URB system.

Fig. 18 shows the motor shaft position control performance at 1 kHz through an RS232 uplink and an I^2C downlink connection for a 0.5-Hz sinusoid reference signal. Consistent with our previous measurements of 2 ms latency, properly tuned PD gains yield a tracking lag of approximately 2 ms for the actual motor shaft position (shown in red) with respect to the reference trajectory (shown in blue). This confirms that the URB framework performs as expected even under the severe noise conditions of the SensoRHex platform.

Fig. 19 illustrates the results of the same experiment performed with a USB uplink condition. This time, the tracking error increases slightly, corresponding to the 3-ms latency associated with our USB uplink implementation. Nevertheless, the USB uplink still preserves real-time performance consistent with our analysis and previous experiments. Note that, here, we use the term real-time to primarily refer to the determinacy of latency and bandwidth performance.

Finally, Fig. 20 illustrates shaft angle and armature current and voltage measurements from one of SensoRHex's hip modules while executing an alternating tripod gait [43]. The overshoots on slowing the leg down (around $t = 0.1$ s) and changing the direction of rotation (around $t = 0.53$ s) are due to the large inertia associated with SensoRHex's legs. The relatively noisy voltage measurements are a result of the fact that a 40-kHz pulsewidth modulation signal is used to drive the motor voltage. Nevertheless, once again, the URB framework works as expected, with deterministic latency and bandwidth performance. These experimental results, hence, show that the architecture we proposed successfully meets both the flexibility and deterministic real-time performance requirements we observed were necessary for instrumentation within resource-constrained mobile robot platforms.

VII. Conclusion

In this paper, we have introduced the URB, a new modular fieldbus architecture with deterministic real-time performance, which is designed for use within instrumentation systems such as autonomous mobile robot platforms where high-bandwidth real-time measurement and control of environmental interactions are needed. The architecture we proposed has a two-tiered structure and supports a heterogeneous collection of communication protocols within the same system, making it possible to integrate components with differing sizes and capabilities within the same system. As such, URB offers a level of flexibility beyond what is provided by existing fieldbus designs, with a programming model that features a transparent interface through which the development of application software remains completely independent of the topological connectivity of components. Through careful design of low-level protocols for supported types of physical connections, URB also preserves desired real-time properties with low predictable latencies and high bandwidths. Finally, we also proposed a synchronization algorithm compatible with the URB framework that successfully synchronizes data acquisition across multiple nodes connected to the system, allowing us to both reduce latency and ensure timing consistency of data acquired on a distributed set of sensory nodes, critical for uniform sampling rate discrete-time control.

We have shown that the proposed architecture can provide both the modularity and real-time performance properties that are critical for autonomous mobile robots and similarly constrained instrumentation applications through a set of systematic experiments. We used the URB architecture for the design and implementation of our six-legged dynamic legged robot design, SensoRHex, and characterized both the latency and bandwidth properties of a practical URB deployment. Our experiments have established that the framework maintains its performance, even under the severe conditions of a fully operational dexterous legged robotic platform. We believe that the modularity, flexibility, and real-time performance offered by the URB framework will be useful for the instrumentation and measurement community by substantially accelerating the design and deployment of new systems with as much component reuse as possible.

In the future, we hope to complete support libraries for additional physical connectivity protocols such as CAN and RS485 for shared bus connections to URB nodes. Moreover, further refinements on the protocol are possible to facilitate node firmware updates, version control, and data correction. Nevertheless, the current URB design is sufficient to provide a consistent, easy-to-use, and very lightweight fieldbus alternative with support for real-time operation that is suitable for use within small-scale instrumentation applications such as mobile robotic platforms.

Acknowledgment

The authors would like to thank H. Komsuoglu for his work on RiSEBus, which constitutes the basis for the design of URB, and G. Gültekin and H. Mıdoğlu for their contributions to the design of SensoRHex's motor control node.

References

[1] D. Prutchi and M. Norris, *Design and Development of Medical Electronic Instrumentation: A Practical Perspective of the Design, Construction, and Test of Medical Devices*. Hoboken, NJ: Wiley-Interscience, 2004.

[2] J. Yick, B. Mukherjee, and D. Ghosal, "Wireless sensor network survey," *Comput. Netw.*, vol. 52, no. 12, pp. 2292–2330, Aug. 2008.

[3] A. Willig, K. Matheus, and A. Wolisz, "Wireless technology in industrial networks," *Proc. IEEE*, vol. 93, no. 6, pp. 1130–1151, Jun. 2005.

[4] M. Chan, D. Estève, C. Escriba, and E. Campo, "A review of smart homes—Present state and future challenges," *Comput. Methods Programs Biomed.*, vol. 91, no. 1, pp. 55–81, Jul. 2008.

[5] R. R. Murphy, "Human–robot interaction in rescue robotics," *IEEE Trans. Syst., Man, Cybern. C, Appl. Rev.*, vol. 34, no. 2, pp. 138–153, May 2004.

[6] E. U. Acar, H. Choset, Y. Zhang, and M. Schervish, "Path planning for robotic demining: Robust sensor-based coverage of unstructured environments and probabilistic methods," *Int. J. Robot. Res.*, vol. 22, no. 7/8, pp. 441–466, Jul. 2003.

[7] P. S. Schenker, T. L. Huntsberger, P. Pirjanian, E. T. Baumgartner, and E. Tunstel, "Planetary rover developments supporting mars exploration, sample return and future human–robotic colonization," *Auton. Robots*, vol. 14, no. 2, pp. 103–126, Mar. 2003.

[8] J.-H. Lee and H. Hashimoto, "Controlling mobile robots in distributed intelligent sensor network," *IEEE Trans. Ind. Electron.*, vol. 50, no. 5, pp. 890–902, Oct. 2003.

[9] A. A. Rizzi, J. Gowdy, and R. L. Hollis, "Distributed coordination in modular precision assembly systems," *Int. J. Robot. Res.*, vol. 20, no. 10, pp. 819–838, Oct. 2001.

[10] M. J. Spenko, G. C. Haynes, J. A. Saunders, M. R. Cutkosky, A. A. Rizzi, R. J. Full, and D. E. Koditschek, "Biologically inspired climbing with a hexapedal robot," *J. Field Robot.*, vol. 25, no. 4/5, pp. 223–242, Apr./May 2008.

[11] U. Nunes, J. A. Fonseca, L. Almeida, R. Araújo, and R. Maia, "Using distributed systems in real-time control of autonomous vehicles," *Robotica*, vol. 21, no. 3, pp. 271–281, Jun. 2003.

[12] J. L. Fernandez, M. J. Souto, D. P. Losada, R. Sanz, and E. Paz, "Communication framework for sensor–actuator data in mobile robots," in *Proc. IEEE Int. Symp. Ind. Electron.*, Jun. 2007, pp. 1502–1507.

[13] I. Verhappen and A. Pereira, *Foundation Fieldbus: A Pocket Guide*. Research Triangle Park, NC: ISA, 2002.

[14] E. Tovar and F. Vasques, "Real-time fieldbus communications using Profibus networks," *IEEE Trans. Ind. Electron.*, vol. 46, no. 6, pp. 1241–1251, Dec. 1999.

[15] Echelon Corp., Palo Alto, CA, 078-0183-01a edition Introduction to the LONWORKS System, 1999.

[16] J. A. Janet, W. J. Wiseman, R. D. Michelli, A. L. Walker, M. D. Wysochanski, and R. Hamlin, "Applications of control networks in distributed robotic systems," in *Proc. IEEE Int. Conf. Syst., Man, Cybern.*, Oct. 11–14, 1998, vol. 4, pp. 3365–3370.

[17] S. T. Bushby, "BACnet: A standard communication infrastructure for intelligent buildings," *Autom. Construction*, vol. 6, no. 5/6, pp. 529–540, 1997.

[18] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Y. Ng, "ROS: An open-source robot operating system," in *Proc. Open-Source Softw. Workshop Int. Conf. Robot. Autom.*, 2009.

[19] H. Bruyninckx, "Open robot control software: The OROCOS project," in *Proc. IEEE Int. Conf. Robot. Autom.*, 2001, vol. 3, pp. 2523–2528.

[20] J. Jackson, "Microsoft robotics studio: A technical introduction," *IEEE Robot. Autom. Mag.*, vol. 14, no. 4, pp. 82–87, Dec. 2007.

[21] Industrial Communication Networks—Fieldbus Specifications, IEC 61158, 2000.

[22] M. Santori and K. Zech, "Fieldbus brings protocol to process control," *IEEE Spectr.*, vol. 33, no. 3, pp. 60–64, Mar. 1996.

[23] R. Patzke, "Fieldbus basics," *Comput. Standards Interfaces*, vol. 19, no. 5, pp. 275–293, Mar. 1998.

[24] S. H. Hong and I.-H. Choi, "Experimental evaluation of a bandwidth allocation scheme for foundation fieldbus," *IEEE Trans. Instrum. Meas.*, vol. 52, no. 6, pp. 1787–1791, Dec. 2003.

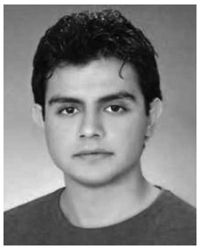
[25] S. H. Hong and S. M. Song, "Transmission of a scheduled message using a foundation fieldbus protocol," *IEEE Trans. Instrum. Meas.*, vol. 57, no. 2, pp. 268–275, Feb. 2008.

[26] Q. Li, D. J. Rankin, and J. Jiang, "Evaluation of delays induced by foundation fieldbus H1 networks," *IEEE Trans. Instrum. Meas.*, vol. 58, no. 10, pp. 3684–3692, Oct. 2009.

[27] K.-S. Hwang, C.-Y. Lo, and W.-L. Liu, “A modular agent architecture for an autonomous robot,” *IEEE Trans. Instrum. Meas.*, vol. 58, no. 8, pp. 2797–2806, Aug. 2009.
[28] A. Z. Alkar and M. A. Karaca, “An Internet-based interactive embedded data-acquisition system for real-time applications,” *IEEE Trans. Instrum. Meas.*, vol. 58, no. 3, pp. 522–529, Mar. 2009.
[29] R. M. Metcalfe and D. R. Boggs, “Ethernet: Distributed packet switching for local computer networks,” *Commun. ACM*, vol. 19, no. 7, pp. 395–404, Jul. 1976.
[30] G. Held, *Ethernet Networks: Design, Implementation, Operation, & Management*. New York: Wiley, 2002.
[31] CAN specification 2.0a, Stuttgart, Germany, 1991.
[32] M. Wargui and A. Rachid, “Application of controller area network to mobile robots,” in *Proc. Electrotech. Conf.*, Bari, Italy, May 1996, vol. 1, pp. 205–207.
[33] Universal Serial Bus Specification, Apr. 27, 2000.
[34] C. Ozturk, “A USB-based real-time communication infrastructure for robotic platforms,” M.Sc. thesis, Dept. Comput. Eng., Bilkent Univ., Ankara, Turkey, 2009.
[35] Philips Semicond., Eindhoven, The Netherlands, The I2C-BUS Specification, Jan. 2000. ver. 2.1.
[36] A. Avci, “The Universal Robot Bus: A local communication infrastructure for small robots,” M.Sc. thesis, Dept. Comput. Eng., Bilkent Univ., Ankara, Turkey, 2008.
[37] M. Gergeleit and H. Streich, “Implementing a distributed high-resolution real-time clock using the CAN-bus,” in *Proc. Int. CAN Conf.*, Mainz, Germany, 1994.
[38] F. B. Schneider, “Understanding protocols for byzantine clock synchronization,” Cornell Univ., Ithaca, NY, Tech. Rep. TR87-859, 1987.
[39] Q. Li and D. Rus, “Global clock synchronization in sensor networks,” *IEEE Trans. Comput.*, vol. 55, no. 2, pp. 214–226, Feb. 2006.
[40] P. Verissimo, L. Rodrigues, and A. Casimiro, “Cesiumspray: A precise and accurate global time service for large-scale systems,” *Real-Time Syst.*, vol. 12, no. 3, pp. 243–294, May 1997.
[41] S. PalChaudhuri, A. K. Saha, and D. B. Johnson, “Adaptive clock synchronization in sensor networks,” in *Proc. 3rd Int. Symp. IPSN*, 2004, pp. 340–348.
[42] J. Encinas, *Phase Locked Loops*. New York: Springer-Verlag, 1993.
[43] U. Saranli, M. Buehler, and D. E. Koditschek, “RHex: A simple and highly mobile robot,” *Int. J. Robot. Res.*, vol. 20, no. 7, pp. 616–631, Jul. 2001.
[44] P.-C. Lin, H. Komsuoglu, and D. E. Koditschek, “A leg configuration measurement system for full-body pose estimates in a hexapod robot,” *IEEE Trans. Robot.*, vol. 21, no. 3, pp. 411–422, Jun. 2005.

Uluç Saranlı (S’94–M’02) received the B.Sc. degree in electrical and electronics engineering from the Middle East Technical University, Ankara, Turkey, in 1996 and the M.Sc. and Ph.D. degrees in computer science from the University of Michigan, Ann Arbor, in 1998 and 2002, respectively.

He was subsequently a Postdoctoral Fellow with the Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, until 2005. He is currently an Assistant Professor with the Department of Computer Engineering, Bilkent University, Ankara. His research interests include the analysis and control of dynamic locomotion with legged robots, nonlinear dynamical systems, embedded systems, software architectures for robot programming and control, and formal methods applied to planning and robotic autonomy.



Akın Avcı (S’04) received the B.Sc. and M.Sc. degrees from Bilkent University, Ankara, Turkey, in 2006 and 2008, respectively. He is currently working toward the Ph.D. degree at the University of Twente, Enschede, The Netherlands.

His research interests include embedded systems, robotics wireless sensor and actuator networks, inertial sensors, and activity recognition.

M. Cihan Öztürk (S’04) received the B.Sc. degree from Hacettepe University, Ankara, Turkey, in 2006 and the M.Sc. degree from Bilkent University, Ankara, in 2009.

He is currently an Embedded Software Engineer with Aselsan, a Turkish defense industry company. His research interests include embedded systems, BSP and device driver development, and robotics.