

Octopus: Monitoring, Visualization and Control of Sensor Networks

Raja Jurdak*, Antonio G. Ruzzelli†, Alessio Barbirato†, and Samuel Boivineau‡

* CSIRO ICT Centre QCAT Technology Court Pullenvale QLD 4069 Australia

† CLARITY: The Centre for Sensor Web Technologies

School of Computer Science and Informatics

University College Dublin Bellied D4 Ireland

‡ Ecole Polytechnique de L'Universite de Nantes, France

Abstract—Sensor network monitoring and control are currently addressed separately through specialized tools. However, the high degree of coupling of network state to the physical environment in which the network is deployed demands that users can monitor the network and respond to network state changes continuously. This paper presents the open-source Octopus visualization and control tool. Octopus is a protocol-independent tool that provides live information about the network topology and sensor data in order to enable live debugging of deployed sensor networks. It enables operators to reconfigure the network behavior, such as switching between time-driven, event-driven, and query-driven modes or between awake and sleep modes of one, many, or all nodes through its graphical interface. Octopus also supports changing duty cycles of nodes, data reporting period, or sensing thresholds in event-driven networks. Reconfiguration of nodes is achieved through short request messages that support typical reconfiguration options without the overhead of epidemically sending new program images over the air. Our empirical tests showcase Octopus’s capacity to debug application behavior and to characterize heterogeneous network performance under multiple settings, as a step towards establishing a rules database that relates data delivery to network-level parameters, and towards enabling autonomous network reconfiguration.

I. INTRODUCTION

Wireless sensor and actuator networks (WSN’s) are a potentially revolutionary network paradigm, where tiny wireless modules equipped with sensors monitor a physical space and enable operators to react to specific events. These networks are highly coupled with their physical environment: events in the physical environment affect the network traffic patterns, and the network is a means for remotely initiating actions in the physical environment. The tight coupling of WSN’s with their physical environments distinguishes them from conventional wireless networks, and demands a fresh approach to monitoring and control.

Conventional network monitoring involves tracking of network health and diagnostic data, while conventional network control involves reconfiguration according to operator requirements. In contrast, monitoring WSN’s involves monitoring of *both* the network state and the sensor physical parameters. Controlling WSN’s involves controlling *both* the network configuration and the behavior of the sensing application.

Another feature of conventional monitoring and control tools is that they are completely separate from the running

application. For instance, a WiFi network sniffer is completely separated from any data transfer application over the WiFi network. In contrast, a sensor network is primarily a monitoring network. Its typical application objective *is*, in fact, to monitor and control the physical environment through the network. These features present new opportunities for integrating monitoring and control of the network and the application in a meaningful way for effective sensor network research and development.

A. Research Challenges

Two challenges that face sensor network research and development efforts are: (1) debugging; and (2) reconfiguration. Debugging sensor network experimental applications is difficult, as the nodes typically only have LED outputs, and in some instances JTAG interfaces for debugging [1]. Thus, it is nearly impossible for a developer to trace the code that executes within a node or to have a granular view of the interaction between nodes in a reasonably large network. Existing tools like Surge [3] or MViz [4] within TinyOS [10] provide users with topology-related information from the network. However, Surge is only compatible with TinyOS version 1, but not the current version 2, and it is highly dependent on the underlying routing protocol, whereas MViz provides users with only passive monitoring capability, without and reconfiguration features.

Another issue that arises when deploying sensor networks is the remote reconfiguration of the nodes. Currently, most users or developers either manually reprogram the nodes individually, which is quite tedious and not scalable, or remotely reprogram them in batches over the air, such as with Deluge [2]. While over-the-air programming provides a high level of flexibility in terms of node reconfiguration, it also involves a substantial delay and consumes significant energy resources for epidemically diffusing the new code image to all the nodes in the network. The inefficiency of over-the-air programming is emphasized in convergecast networks, where changing even simple network parameters would require fully reprogramming the nodes.

A final challenge facing widespread sensor network adoption is the need for dynamically reconfiguring the network

according to changes in user requirements or in ambient conditions. A plethora of self-organizing sensor network methodologies have recently been proposed [14, 15]. Most of these methodologies use greedy local processes [11, 12] at the sensor nodes to make reconfiguration decisions. Greedy processes are scalable as they only rely on local information, but they do not incorporate knowledge on network nodes beyond the local neighborhood. The main obstacle to using global state information is the communication and energy overhead involved in collecting the global network state. Other techniques rely on distributed collection of more comprehensive state information of nodes beyond the local neighborhood, by piggybacking the state information of a source node in data packets and snooping on this information at intermediate forwarding nodes [13].

An alternative method for state collection is to rely, at least partially, on the base station, which already has a reasonably comprehensive view of the network state resulting from the convergence of all data packets to the base station. Leveraging network state information at the base station would therefore enable more informed network configuration decisions, but the centralized nature of this technique limits its scalability. A hybrid approach appears to be the best solution, where sensor nodes manage their local configuration in a greedy distributed manner, and the base station manages network level parameters centrally. The objective of such a strategy is to allow the network to adapt to state changes autonomously based on a combination of distributed and centralized information. To reach this objective demands development of a database of rules for configuring network behavior on the basis of changes in the observed state.

B. Integrated Monitoring, Visualization and Control

To address the above challenges, this paper presents the Octopus sensor network monitoring, visualization and control tool. Octopus is a protocol-independent open-source tool developed at University College Dublin in both nesC [16], for the embedded application, and Java, for the user application, specifically for TinyOS Version 2. Octopus provides sensor network practitioners with live information about the network topology and sensor data for intuitive and user-friendly monitoring of the network and sensor state. It also enables reconfiguration of the network and of the application by sending short commands to the sensor nodes over the air. Octopus distinguishes between 2 groups of sensor network practitioners: (1) sensor network developers; and (2) sensor network users. Developers may wish to tailor Octopus for specific applications, add physical maps of the deployment area, include new reconfiguration commands, or port Octopus for new hardware or software platforms. To address the needs of developers, Octopus is independent of the underlying routing protocol and its reconfiguration commands are easily extendible, which enables straightforward customization of the code. In contrast, users may only require quick plug-and-play monitoring of a physical environment, with the ability to support in-situ network reconfiguration. The Octopus GUI

provides users with intuitive visualization of the network and simple graphical features to reconfigure the network.

Taking a step towards building a rules database through which the base station can autonomously reconfigure the network, we have leveraged Octopus to collect empirical performance data under different network conditions. The empirical results relate the data delivery rate to network-level parameters, including duty cycle, network size, and sampling period, which guides the settings of these parameters at the base station for a given set of performance guarantees.

C. Overview of the Paper

This paper provides a comprehensive presentation of the design and features of Octopus, including:

- 1) Design choices and specifications: The design of Octopus follows a protocol-independent and extendible approach that facilitates customization by developers. It also advocates integration of monitoring and control of the network and the application in sensor networks. We present Octopus's component structure, message types and formats, and the application layering scheduling feature.
- 2) High level features: Octopus provides users with several features to customize their visualization of network topology and collection of sensed data. It also enables users to control the configuration of nodes through simple requests. All user features are available in the graphical user interface, that represents the front-end of Octopus.
- 3) Empirical testing: We leverage Octopus to monitor, debug and configure the network in empirical tests. These tests showcase the versatility of Octopus in tracing network behavior in response to physical stimuli, uncovering any erroneous behavior, and easily changing the network settings by issuing request messages through the GUI.

The remainder of the paper is structured as follows. Section II discusses previous work on network visualization and control tools. Section III presents Octopus's design. Section IV presents our empirical testing of Octopus as tool for monitoring and controlling different sized networks and to demonstrate its use in building up a database of dependencies among network parameters. Section V discusses the results and future directions, and concludes the paper.

II. RELATED WORK

Most tools for sensor network management have addressed network state monitoring and network control separately. From the early days of TinyOS version 1 development, the need for monitoring and visualizing node and network state became evident, both for debugging and for a deep understanding of inter-node interactions. This led to the development of the Surge application [3] and its inclusion in the standard release of TinyOS version 1. Surge exemplifies a typical convergecast network application, where all nodes sample their sensors and send the data periodically towards the base station in a

multihop fashion. Surge came in several flavors that supported different sensing modalities. In one version, the node would sample each of several attached sensors in sequence, place all the data into a packet, and send it to its routing parent. In another version, the packet contained only one sensor value, which the user could select or replace with a dummy sensor to debug communication functionality. Surge also provides users with the option of putting nodes into Sleep or Focused modes, or to release them from these modes. In Sleep mode, a node turns off all its application layer timers and simply waits for a wakeup message from the base station. Focused mode enables a user to target a single node and change its behavior individually. Surge also supports data logging into a file for later analysis. While Octopus resembles Surge in its network visualization and data logging aspects, Octopus supports a wider range of user requests to serve various sensor network applications, such as modifying the sampling period, setting the duty cycle, changing between query-, event-, or timer-driven modes, and setting the sensing thresholds. Octopus is also independent of the underlying routing protocol (see Section III-C), whereas Surge is dependent upon the Mint Route routing protocol in TinyOS 1, which uses the ETX routing metric. Finally, Octopus specifically targets the current version 2 of TinyOS while Surge in its current form is non-compatible with this TinyOS version.

Currently, MViz [4] is the only available network visualization tool for TinyOS version 2 that provides users with passive network monitoring but no facilities for any remote control of the network. Octopus differs from MViz as it enables users to control and to monitor the network.

Another approach to sensor network management and monitoring revolves around testbeds. Motelab [5] provides a hybrid hardware and software system for scheduling jobs on a sensor network testbed and obtaining results. It allows users to upload their source code through a Web interface, to schedule the time and duration of their jobs, and to obtain both real time or historical data from their jobs. Motelab does not provide a topology visualization facility, so Octopus can in fact be synergistic with Motelab by using the Java client to connect to the Motelab's SerialForwarder. In addition, Motelab's purpose is to make testbeds at large research centers available to internal and external developers to test the correctness and scalability of their application code. Octopus targets developers and users intending to deploy, monitor, visualize, and control their own sensor network and application. Twist [6] is a similar testbed system that adds power control features.

Sensor network control tools have also been developed for remote configuration of the nodes. A major class of these tools is network reprogramming tools. For example, Multihop Over-the-Air Programming (MOAP) [19] is a code distribution mechanism that specifically targets Mica-2 Motes. Multihop Network Reprogramming (MNP) [20] builds on MOAP and addresses the problem of sender selection to avoid redundant dissemination of the same code updates. It also proposes a phase-by-phase approach and rate-based flow control to address sender-receiver synchronization and

the issue of sending bulk data over low bandwidth wireless links. One of the key sensor network reprogramming tools is Deluge [2], which was originally designed for TinyOS version 1, while efforts to port it to TinyOS version 2 are still underway. Network reprogramming tools enable users to upgrade existing programs running on the nodes by sending the entire code image of the new version, and then rebooting the nodes with the new image. Similarly, users can upload completely new programs to the sensors. Every node that receives a new code image re-advertises the new image to its neighbors and sends it to any neighbors that have yet to receive this image. This results in epidemic dissemination of the new image until it reaches all nodes. SLUICE [21] complements network reprogramming tools by adding security to prevent the malicious injection of packets into the network to gain control over it.

Over the air network programming is most useful for updating the network with new program functionality. As a result, it maximizes flexibility in controlling the sensor nodes remotely at the expense of high communication and control overhead for disseminating the new code image in the network for any required change in configuration. In contrast, the design of Octopus has tried to identify key reconfiguration commands that are common to most sensor network applications, and to provide support for these commands. This design choice significantly reduces the overhead of reconfiguring the nodes, albeit with reduced flexibility in the range of available commands.

Another approach for sensor network control that fits between Octopus and network reprogramming on the spectrum of flexibility in reconfiguring the network is the use of virtual machines, as in Mate [22]. Mate is a byte-code interpreter that can run on the motes. It relies on capsules of 24 instructions per packet that can forward themselves through a network. It composes complex programs by sending the code in several capsules for a combined size in the order of hundreds of bytes, but its energy overhead is less than Deluge as it significantly reduces the size of instructions sent to the nodes. However, the overhead of Mate is still an order of magnitude higher than Octopus, which uses short request messages of up to only 6 bytes.

MANNA [7] explores sensor network management architectures. In particular, it considers the possibilities of having centralized, hierarchical or distributed network managers, as well as the software agents that interact with the manager. Following MANNA's classification, Octopus provides centralized management of *network level* parameters, such as sampling period and duty cycle, from the GUI at the base station. Agents in Octopus handle all local decisions so they are distributed at all the nodes. The distributed agents can also respond to requests from the centralized network manager.

Marwis [8] provides a monitoring, configuration, and code propagation architecture for sensor networks that runs over Contiki. The Marwis architecture positions a wireless mesh network as an overlay for large heterogeneous sensor networks and provides web-based access to the data. Octopus

resembles Marwis in its support of web-based access and intermediate network planes. However, the main difference between Octopus and Marwis is that Octopus provides built-in visualization functionality, whereas this does not exist for Marwis. The SUMAC project [9] also addresses hybrid Wi-Fi mesh and sensor networks. It provides web-based monitoring of sensor networks through a Google Maps interface, complex queries, group selection, and historical data plotting. It also enables users to set sampling periods and aggregation levels in the network from the web-based interface. While SUMAC shares many of the foundations of Octopus, it is a proprietary system that is geared towards large scale monitoring of several deployments, whereas Octopus is an open-source tool that targets the management of a single sensor cluster.

A key consideration for network reconfiguration is to address the needs of a large pool of sensor network applications. Most current applications adopt a convergecast topology, where sensor nodes sample their sensors and transmit their sensor data periodically to a single base station. Such applications typically require reconfiguration of network parameters while maintaining the core functionality of the original application. They also require user visibility into the real-time state of the network. Because virtual machines and network reprogramming involve high transmission overhead, they are more suited for less frequent reloading of new entire applications. In contrast, the low overhead of network reconfiguration with Octopus renders it suitable for frequent tuning of network parameters, especially for convergecast network applications. In addition, Octopus combines both aspects of control and visualization, whereas both virtual machines and network reprogramming focus on the control aspects and leave visualization for other applications, such as Surge or MViz.

III. OCTOPUS DESIGN

Octopus is a monitoring, visualization and control tool for wireless sensor networks implemented in nesC for TinyOS version 2. It provides an integrated platform for both observing and reconfiguring sensor networks over the air, with a focus towards convergecast networks that require frequent updates to network configuration. The Octopus tool enables the viewing of the live network topology and the customization of this view according to specific application needs. It enables users to reconfigure the behavior of one, many, or all nodes through a graphical interface, shown in Figure 1. The interface provides support for switching between time-driven, event-driven, and query-driven modes or between awake and sleep modes. In addition, Octopus also supports changing duty cycles of nodes, sampling period, or sensing thresholds in the case of event-driven networks. Octopus reconfigures nodes through short request messages that support typical reconfiguration options (see Section III-C) without the overhead of epidemically sending new program images over the air.

The remainder of this section is structured as follows. Section III-A provides an overview of the design principles of Octopus. Section III-B presents the software component structure of Octopus, while Section III-C discusses Octopus's

message formats that maintain its protocol independence. Finally, Section III-D introduces the high-level features that the Octopus tools provides to sensor network users, with a focus on the graphical user interface.

A. Design Overview

The main design concepts of Octopus are the following:

- **Embedded network monitoring:** sensor network applications monitor physical spaces. The goal of network monitoring is to monitor the network that is monitoring the physical space. Octopus combines the monitoring of the network *and* the application into one tool that monitors both the network state and the physical space.
- **Protocol independence:** Octopus is independent of all underlying protocols, as it includes all visualization and control data in application layer packets.
- **Centralized control of network-wide parameters:** decisions on network-level parameters, such as duty cycle, sampling period, and the like are taken at the base station, which has a comprehensive view of the network state.
- **Live debugging of physically coupled networks:** live network topology and status information in Octopus enables operators to constantly monitor the state of both the network and the physical space

Octopus plays the dual role of application and monitoring tool. On one hand, Octopus serves as a configurable convergecast application, where all nodes sample sensors and send data on a time-driven, event-driven, or query-driven basis to a base station node. During the operation of this convergecast application, Octopus enables a network operator to monitor the status of the network and to reconfigure it when needed. Alternatively, Octopus could be adapted to support more complex functionality, as discussed in Section V.

The Octopus design strategy also advocates a hybrid network control topology, as a compromise between centralized and distributed control. Centralized control focuses all of the network configuration decisions at the base station. Its advantage is that the base station has a more comprehensive view of network state than individual nodes, which allows it to make better network configuration decisions. In contrast, distributed control spreads the decisions over all the nodes. This enables each node to promptly react to changes in its own and its neighborhood states, which improves the responsiveness of the network to stimuli. Octopus's hybrid strategy adopts user-driven centralized control for network-level parameters, and automatic distributed control for node-level parameters. The reasoning behind this strategy is that decisions on network level parameters, such as sampling period and duty cycle, should only be made on the basis of a comprehensive view of the network state, as they affect the network as a whole. The base station is already the convergence point of all data packets in the network. The base station can thus extract relevant information about the state of all the nodes, and in turn form a reasonably representative view of the current state of the network. Individual nodes cannot make network-level decisions on the basis of their locally available information, as

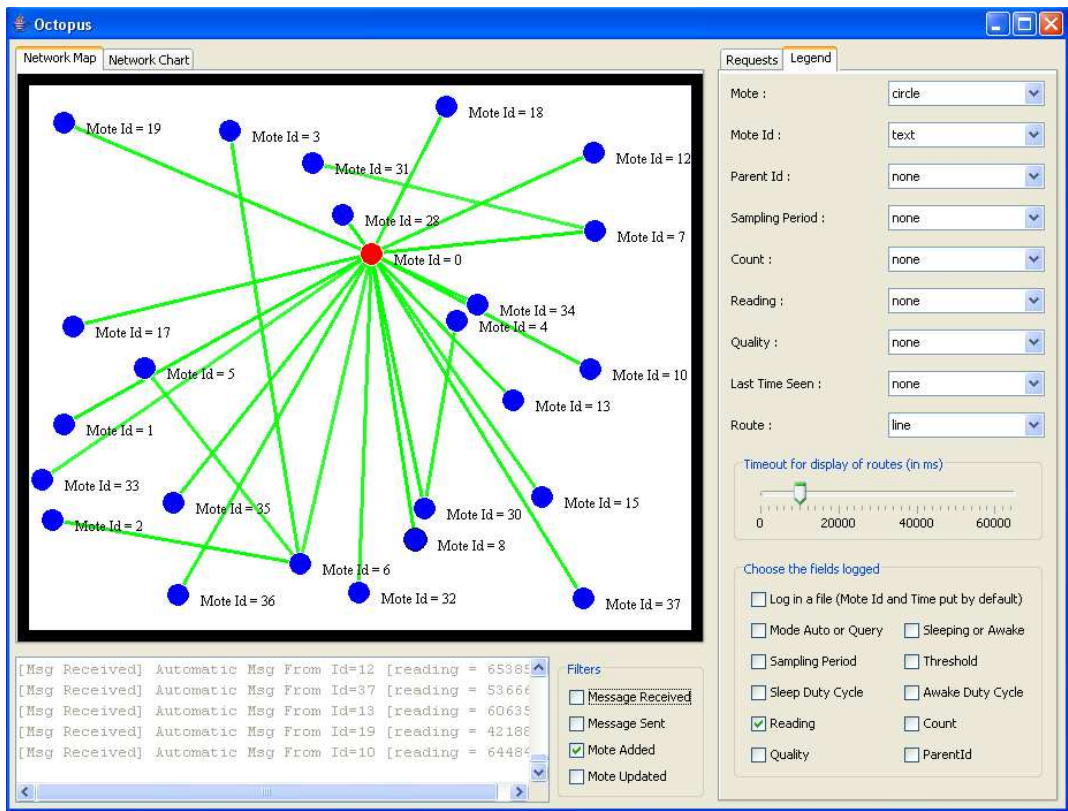


Fig. 1. Octopus graphical user interface

this amounts to a greedy approach that may lead to unfairness in the network. It is also not feasible to collect comprehensive state information at individual nodes due to the overhead involved in flooding state information all over the network. As a result, the user can control all network-level parameters centrally according to the network state that is available at the GUI. Configuration decisions on all other parameters that relate to individual node state (node-level parameters) occur locally at each node, to ensure that nodes are responsive to changes in their local environment.

The live network topology map in Octopus, which is further discussed in section III-D, is a key feature for real-time network debugging. The need for this feature arises from the paradigm shift of sensor networks towards physically coupled networks, as opposed to conventional networks, whose performance and state are mostly detached from random events in the physical environment. For instance, the traffic patterns and routing paths in wired or mobile phone networks are almost independent of the movement of people or cars in a particular sector of the network. In contrast, the movement of a person or an animal within the deployment region of a sensor network often causes changes in link qualities and possible rerouting of packets. Using conventional debugging techniques, which are geared mostly for computer simulation environments with full user control and access to the program flow, network operators can not capture such topological changes due to fluctuations in the physical state. The Octopus

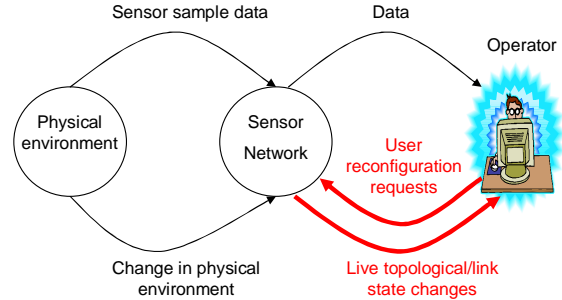


Fig. 2. Octopus enables live topology/link states monitoring for more informed reconfiguration by the operator

live topology map bridges this gap, as shown in Figure 2, by allowing operators to observe instantaneous changes in the network topology and link states as they happen. This empowers operators to correlate the changes with any observed variations in the physical deployment range, on the basis of which operators can reconfigure the network accordingly.

This section has provided an overview of Octopus's design choices, which include embedded monitoring, protocol independence, hybrid control, and live debugging. The next section discusses Octopus's software component structure.

B. Octopus Structure

The main entities that characterize the Octopus structure are shown in Figure 3, and they include: (1) the Graphical

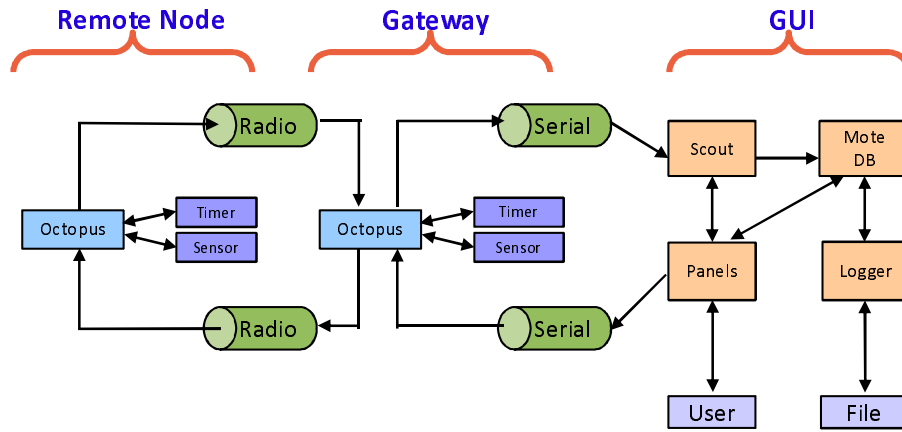


Fig. 3. The component structure of Octopus

User Interface (GUI), (2) the gateway node, and (3) the remote nodes. The GUI is a Java application that serves as the frontend of Octopus, providing a flexible interface for the user to visualize the network topology, to log the incoming data, and to send directives to the nodes. The GUI connects to the gateway node through the serial port on the host computer. Octopus uses the radio and serial connections to communicate with the nodes and the GUI respectively, and it uses the timer/sensor to process and send data.

The embedded portion of Octopus relies on three main components: Octopus core, timers, and sensors. The timers are responsible for maintaining the required sensor and data reporting frequency. In time-driven mode, all nodes set a periodic timer. When that timer fires, the nodes sample their sensors and send the sampled sensor data in packets over the air. In event-driven mode, a timer firing causes nodes to check whether any of the sensor values exceeds its threshold before deciding whether to send the sampled data in a packet. Query-driven mode requires nodes to wait for a sampling request from the gateway to sample their sensors and send the sampled value back to the gateway. The second component of the embedded portion of Octopus is the sensors. The embedded application is wired, through TinyOS programming interfaces, to all the relevant sensors. The Octopus core component simply uses the software interfaces to issue commands to sample the sensors and receive the data samples. Finally, the Octopus core provides the required functionality that enables visualizing network state data and controlling node behavior remotely. It mainly embeds diagnostic and state information in Octopus messages for consumption at the back-end, and it contains the logic for processing incoming requests for reconfiguring node parameters. In addition to these main components, the Octopus embeds application interfaces and leverages all the TinyOS sending and receiving components, such as collection and dissemination, for transporting both upstream and downstream messages. Nodes always receive and forward upstream message through the collection tree protocol (CTP). Downstream messages use the underlying dissemination protocol, which in the case of TinyOS is based on Trickle [27].

Both the gateway node and the remote nodes run the same embedded code. Nodes determine at run-time whether they are the gateway or simply a reporting node, by checking whether their TOS_NODE_ID is zero. If so, then the node sets a flag 'root', indicating that it is the gateway node. While both the gateway and regular nodes comprise the main three components of the Octopus embedded application and use the same underlying collection and dissemination protocols, their send and receive interfaces are wired differently. The gateway node receives collection messages through its radio and forwards these message to the serial port. Configuration request messages, which flow downstream and rely on the dissemination protocol, are received at the gateway node through the serial port, after which the gateway node sends them downstream through the radio. In contrast, regular nodes both receive and send messages through their radios for both upstream and downstream messages.

The Octopus embedded code for the gateway and remote nodes is structured as a typical TinyOS application. It includes a configuration file, where all the component wiring is done, and an implementation file, containing all the tools functionality. It also implements functionality for sampling sensors, sending the data in packets, and receiving data over the serial port or the radio. The default sampling period is specified in Octopus_Config.h, but this can be changed during deployment by the network user.

To support network reconfiguration, the embedded code running at the nodes provides support for incoming request messages. The main function that is responsible for processing user requests in the Octopus embedded application is the processRequest() function, a portion of which is shown in table 4. This function is called whenever a node receives a new request from the user, which is indicated by an event from lower layers. processRequest() first verifies that the incoming request is meant for this node, by comparing the request's targetId parameter with the node's own ID, and by checking whether the request is a broadcast to all nodes. If the request is intended for this node, processRequest() checks the request type through a switch statement. For each request

```

void processRequest(octopus_sent_msg_t
*newRequest) {
    if (newRequest->targetId==TOS_NODE_ID
|| newRequest->targetId==0xFFFF){
        switch (newRequest->request) {
            case SET_MODE_AUTO_REQUEST:
                modeAuto = TRUE;
                call Timer.stop();
                waitT = (1+TOS_NODE_ID % MAX_NUM_NODES)* alpha;
                call WaitTimer.startOneShot(waitT);
                break;
                ...

            case SET_PERIOD_REQUEST:
                samplingPeriod = newRequest->parameters;
                call Timer.stop();
                alpha = samplingPeriod/(max_num_nodes+1);
                waitT = (1+TOS_NODE_ID)* alpha;
                if (sleeping == FALSE)
                    call WaitTimer.startOneShot(waitT);
                break;
                ...
        }
    }
}

```

Fig. 4. Process request function

type, processRequest() implements the desired node response to the request. The code sample shows the implementation of 2 request types in the processRequest() function: setting auto (time-driven) mode; and adjusting the sampling period. For both requests, the sensor sampling timer is stopped and a wait timer is called to avoid implicit synchronization of nodes that receive the request at the same time. Section IV-A elaborates more on this mechanism. In addition to the currently supported requests in Octopus, developers can add new request types by simply including a new 'case' in the switch statement of the new request types, along with the desired sensor response.

The Octopus GUI itself is composed of four components: (1) Scout; (2) Mote Database; (3) Panels; and (4) Logger. Scout is a thread that simply listens to the serial port and processes the received messages. Every received packet causes the Mote Database to update the status of the corresponding nodes. Any component within the GUI accesses the database to learn the status of a particular node. The Logger component refers to the database before storing the collected data in a file for future analysis. Finally, the Panels component handles output to the user, for displaying data on the screen, and user input, mainly in responding to keyboard or mouse events.

The next subsection introduces the messages which enable interaction among Octopus components, and between Octopus and the user.

C. Message Formats and Protocol Independence

The protocol independent feature of Octopus stems from its use of solely application layer information for visualization and control of sensor network, unlike tools like Surge, which use routing layer information from MintRoute to provide a graphical map of the network topology. This section introduces the Octopus application message structure that maintain its protocol independence.

The embedded code supports TinyOS's default collection

```

// Specifying a dissemination protocol
#if defined (DISSEMINATION_PROTOCOL)
// for the root
interface DisseminationUpdate<octopus_sent_msg_t>
    as RequestUpdate;
// for the motes
interface DisseminationValue<octopus_sent_msg_t>
    as RequestValue;
#endif defined (DUMMY_BROADCAST_PROTOCOL)
interface DummyBroadcast;
#else
#error 'A broadcast protocol needs to be selected'
#endif

// Specifying a collection protocol
#if defined (COLLECTION_PROTOCOL)
// specific to the ctp protocol
interface CtpInfo as CollectInfo;
#endif defined (DUMMY_COLLECT_PROTOCOL)
// specific to dummy protocol, used to get some data
interface DummyInfo as CollectInfo;
#else
#error 'A collection protocol needs to be selected'
#endif

```

Fig. 5. Specification of collection and dissemination protocols

tree protocol and dissemination protocol, as well as other collection and dissemination protocols, using the configuration code in figure 5. These preprocessor commands provide developers with a choice to use the TinyOS default protocols by simply adding the statements:

```

#define DISSEMINATION_PROTOCOL
#define COLLECTION_PROTOCOL

```

in the Octopus_Cofig.h header file. Developers can similarly specify other protocols by using the #define preprocessing command to use other protocols that may become available, which are nominally referred to a DUMMY_BROADCAST_PROTOCOL and DUMMY_COLLECT_PROTOCOL.

The Octopus application employs two message types for communication: (1) downstream messages; and (2) upstream messages. Because the messaging layer in TinyOS 2.x is independent of the physical layer, Octopus uses the same message format over both the serial port and the radio connection.

Downstream messages target communication from the GUI to the gateway, and from the gateway to the regular motes. These messages carry user requests from the GUI towards the remote nodes. Figure 6 shows the structure of Octopus downstream messages. The message includes the *targetId* field (16 bits) to identify the node or set of nodes for which the request is meant, the *request* identifier (8 bits), and a set of *parameters* (16 bits). Table I shows all the currently supported downstream requests within Octopus, bearing in mind that the 8-bit request identifier can support up to 256 request types. The current Octopus design supports a handful of typical configuration message types. However, extending the supported configuration message simply requires developers to create a new condition in the switch statement and to add the message ID in the Octopus.h file.

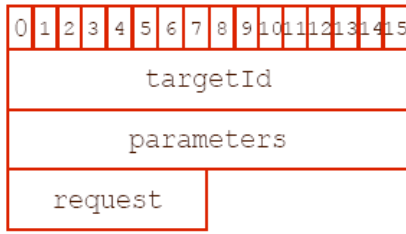


Fig. 6. Format of Octopus downstream messages

Upstream messages are sent from remote nodes to the gateway, and from the gateway to the GUI. These messages carry sensor readings, and replies to user requests. An upstream message, shown in Figure 7, contains the following fields in its header: *parentID*, *link quality*, *count*, *reading*, *moteID*, and *reply*.

The Octopus GUI visualizes paths by aggregating local topology information embedded in upstream messages at the base station. The local topology information has a vector format, as shown in Figure 8(a). The first time a message arrives from a node A at the base station, the Java application creates a new node and attaches ID "A" to the node, by extracting this field from the message header *moteID* field. Note that the *moteID* variable refers to the source node and remains unchanged in a message even when the message is forwarded along multiple hops. This variable is only included in the application layer packet.¹ This provides the source endpoint of the vector in the GUI. To determine the direction and destination endpoint of the vector, the Java application checks the *parentID* field in the incoming message. The weight of the vector, which can be viewed using the link quality option in the Legends Panel, is contained in the *link quality* field. This value is available at the base station and can be logged into a file for later analysis of the evolution of link qualities during the deployment. By aggregating the vectors of all nodes in the network, the Octopus GUI can form the network tree topology, as shown in Fig 6(b).

The *reading* field of the incoming message includes the sensor reading. The Octopus Java application at the base station can display it in the Sensor Reading field in the lower part of the Requests Panel, log it in a file, or plot it in real-time in the Network Chart screen when the node is selected. Finally, the *reply* field indicates if the packet is a reply to a user request, and if so it includes the type of reply. Otherwise, the message is simply an unsolicited periodic message carrying the sensor reading.

Because Octopus messages include all the information necessary for visualizing and controlling the sensor nodes, the tool is independent of the underlying routing protocols. The vector format that combines (*moteID*, *parentID*), which are both included in the Octopus message, ensures that the network topology is visualized correctly in the GUI. Similarly,

¹Packet forwarding still occurs at the routing layer, where source and destination addresses are modified at every hop to reflect the current source and destination of a packet in transit.

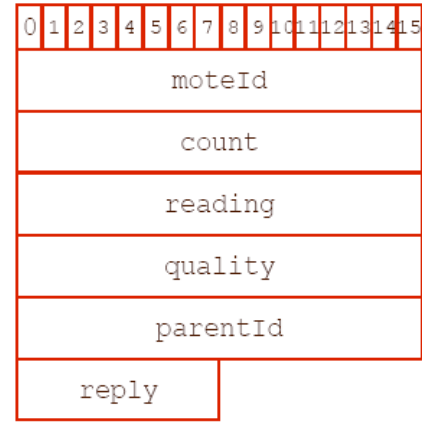


Fig. 7. Format of Octopus upstream messages

all Octopus requests (downstream messages) can be carried as the data payload of any underlying dissemination protocol. Only the *link quality* field assumes that an underlying protocol provides a measure of link quality. If that is not the case, then this field can be simply set to a nominal value for all nodes. Octopus only assumes that an underlying routing protocol determines the routing parent at each node, and that this underlying protocol can make the routing parent identity available to the Octopus application.

The protocol independence of Octopus enables developers to readily reuse the Octopus nesC and Java code with any other underlying routing or MAC layer protocols. This plug-and-play feature of Octopus provides developers and researchers with the freedom to debug and evaluate new protocols using Octopus, and to readily adapt Octopus for new applications with TinyOS 2. Furthermore, this feature enables the development of an Octopus widget, for example within popular web browsers, to check a remote network's status from any browser independently of TinyOS, as we discuss in Section V.

The discussion has so far focused on Octopus's design, component structure, and message formats, which define Octopus's backend functionality. The Octopus GUI, which is described in the next subsection, builds on these backend features to provide an interface for network monitoring and control to the user.

D. Octopus Features

Figure 1 shows the Octopus graphical user interface, which represents the front-end of the Octopus tool. The central part of the GUI visualizes the live network topology, where the nodes are represented by blue circles, the base station with ID 0 by a red circle, and the lines represent wireless links between nodes. Currently, Octopus depicts the logical topology of the network, i.e. the hierarchical relationship between the nodes, while the physical locations of the nodes in the GUI are random.

The right side of the interface provides two tabs to alternate between the Requests and Legend Panels. The Requests Panel, shown in Figure 9(a), enables the user to issue requests to all, many, or one node in order to change their configuration. The

Type (8 bits)	Parameters (16 bits)	Reply (8 bits)	Reading (16 bits)
SET_MODE_AUTO_REQUEST	none	none	none
SET_MODE_QUERY_REQUEST	none	none	none
SET_PERIOD_REQUEST	period (16 bits)	none	none
SET_THRESHOLD_REQUEST	threshold (16 bits)	none	none
GET_STATUS_REQUEST	none	mode (1 bit)	battery (16 bits)
GET_PERIOD_REQUEST	none	none	period
GET_THRESHOLD_REQUEST	none	none	threshold
GET_READING_REQUEST	none	none	reading
SLEEP_REQUEST	none	none	none
WAKE_UP_REQUEST	none	none	none
GET_SLEEP_DUTY_CYCLE_REQUEST	none	none	sleepDutyCycle
GET_AWAKE_DUTY_CYCLE_REQUEST	none	none	awakeDutyCycle
SET_SLEEP_DUTY_CYCLE_REQUEST	sleepDutyCycle	none	none
SET_AWAKE_DUTY_CYCLE_REQUEST	awakeDutyCycle	none	none
MAX_ID	max_num_nodes	none	none

TABLE I
MESSAGE TYPES AND FORMATS IN OCTOPUS

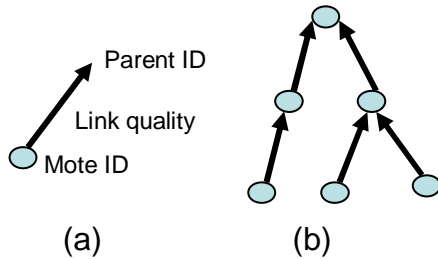


Fig. 8. (a) Octopus message vector format. (b) Aggregated vectors for forming topology.

check box at the top of the Requests Panel enables the user to select broadcast or unicast mode. The user can also select a group of nodes from the network topology by clicking on multiple nodes with the CTRL key pressed, in order to send requests to a group of nodes. The Requests Panel supports the reconfiguration of the reporting mode of nodes, among query-driven, event-driven, or time-driven. Time-driven mode, which can be activated with the "Auto" button, enables periodic monitoring applications, where nodes sample their sensors and send their data to the base station at regular intervals. In event-driven mode, the user can set thresholds for the variation of the sensed value through the "Threshold %" slider in the lower part of the Requests Panel. A nodes only transmits a packet when its sensed value varies by the indicated threshold. Setting the threshold value to zero reduces to placing the nodes in "Auto" mode. Finally, in query-driven mode, which can be activated with the "Query" button, nodes only sample their sensors and transmit the data upon reception of a user query, which can target all or some of the nodes. The user can send a query to a sensor by first selecting the node in the network topology screen, and then clicking the "Read the Sensor" button at the bottom of the Requests Panel. Shortly

afterwards, the sensor data value appears as the "Reading" field next to the "Read the Sensor" button. The user can also use the "Reading" field in "Auto" (time-driven) mode, by clicking on a particular node. The "Reading" field then shows the current sensor reading of the selected node, which changes periodically upon every new packet received from that node.

The Requests Panel also provides for Awake and Sleep modes through the "Wake Up" and "To Sleep" buttons respectively. In Awake mode, the nodes operate normally, subject to the reporting mode set by the user. In Sleep mode, nodes turn off all of their application layer timers, suppress packet generation, and simply wait for a "Wake Up" message from the base station. A node in Sleep mode still forwards packets of other nodes and continues to declare its routing state in periodic beacons. As such, Sleep mode in Octopus refers to the functions of the application layer only. The user can also set the radio duty cycle of each node remotely. In fact, Octopus distinguishes between the sleep duty cycle, which is the duty cycle of a node in Sleep mode, and the awake duty cycle, which is the duty cycle of a node in Awake mode. In general, nodes in Awake mode are more active so their duty cycle should be set higher than nodes in Sleep mode. Both the sleeping and awake duty can be set through their corresponding sliders in the middle portion of the Requests Panel. The user can drag the slider to the desired duty cycle value, and upon releasing the mouse button, the base station sends a request to the selected nodes to set their duty cycles accordingly. The user can similarly set the Sampling Period through the corresponding slider in the Requests Panel. The three buttons at the bottom of the Requests Panel enable the user to select nodes of interest for plotting their recent sensor values in the Charts Panel (See Figure 11).

The Legend Panel, shown in Figure 9(b), provides options for customizing the information that appears in the topology portion of the Network Map Panel. The user can choose to

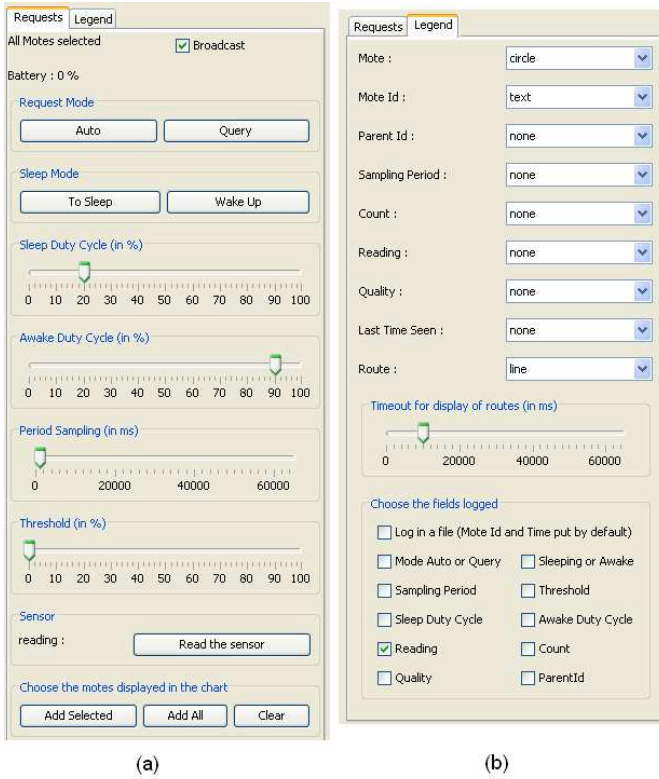


Fig. 9. The Requests and Legends Panels

view tags such as node ID, sensor reading, parent ID, or packet sequence number next to each node. The user can also set the timeout for display of routes for the network links. For a sampling period S , the user may determine that a node that has not sent any packet to the network for $3S$ has lost connectivity from the network. As a result, the user can use the "Timeout for display" slider to set its value to $3S$. Links will then have a fadeaway latency of $3S$, which means that the visualized link will start to fade away progressively from the time the last packet is received at the base station until it disappears entirely after $3S$ time units. If the base station receives another packet from the same node before $3S$ time units have passed, then the GUI brings the corresponding link to full brightness again and resets the fadeaway timer.

The Legend Panel also enables users to log received data into a file, by checking the "Log in a file" checkbox, and to choose exactly which fields are logged into the file, by selecting the corresponding checkboxes below. The user can log the following fields in the current Octopus configuration: reporting mode, awake or asleep mode, parent, link quality, threshold, sleep and awake duty cycles, sensor reading, and packet sequence number.

The bottom part of the Octopus GUI includes the Console Panel (see Figure 10), which provides a textual output of messages sent or received, and updates to node states. The user can choose the textual output of interest, such as received messages, sent messages, mote arrivals or mote updates, through the checkboxes on the right.

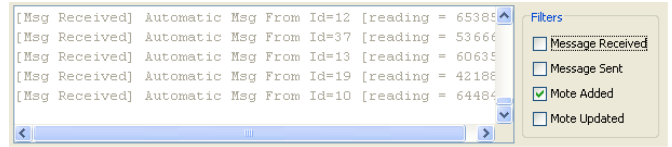


Fig. 10. The Console Panel

Finally, the main screen in Octopus also provides the Network Chart tab which provides a live plotting facility of recent sensor data. Figure 11 illustrates the network chart feature in Octopus that plots the recent readings of multiple nodes using a simulated sinusoidal sensor. The user can select one or several nodes in the Requests Panel of the Network Map screen, and then proceed to the Network Chart screen to view a plot of the sensor data in real-time.

IV. PERFORMANCE EVALUATION

This section presents our empirical experiments to leverage Octopus as a debugging, monitoring, control and visualization tool with different sized networks empirically, under multiple duty cycle and sampling period conditions. The experiments involve networks of heterogenous nodes, all of which use IEEE 802.15.4 as the PHY/MAC communication standard, including MicaZ [17] and Telosb [18] motes. We have performed three distinct sets of experiments that explore the network dynamics through Octopus. The first set of experiments examines the effect of sampling period on the delivery rate. A byproduct of this experiment is the discovery of a bug in Octopus's original implementation that led to implicit synchronization of nodes and adversely affected delivery rates. This led us to introduce application-layer backoff as a standard mechanism in Octopus. The second set of experiments investigates the impact of varying the duty cycle and network size on the delivery rate, while the final set of experiments explores the effect of duty cycle on data delivery rates and delay in networks of different sizes. In addition to unveiling a limitation of the low power listening implementation in the current release of TinyOS 2, the results from these experiments begin to provide trends that can drive a rules engine for automatic reconfiguration of the network.

A. Debugging

The first set of experiments demonstrates Octopus's effectiveness as a network debugging tool. We initially deployed a network of 20 TelosB nodes in our lab in order to use Octopus log the network delivery rates under varying sampling period. All the nodes were placed in a single large office to simulate a dense topology. Observed delivery rates were relatively low, in the range of 80 to 85%, with little dependence on the sampling period. This led us to believe that there was a fundamental bug in the application, as previously reported delivery rates from similar testbeds exceed 95%. Using various Octopus requests, we frequently varied the network configuration, which quickly revealed the implementation bug.

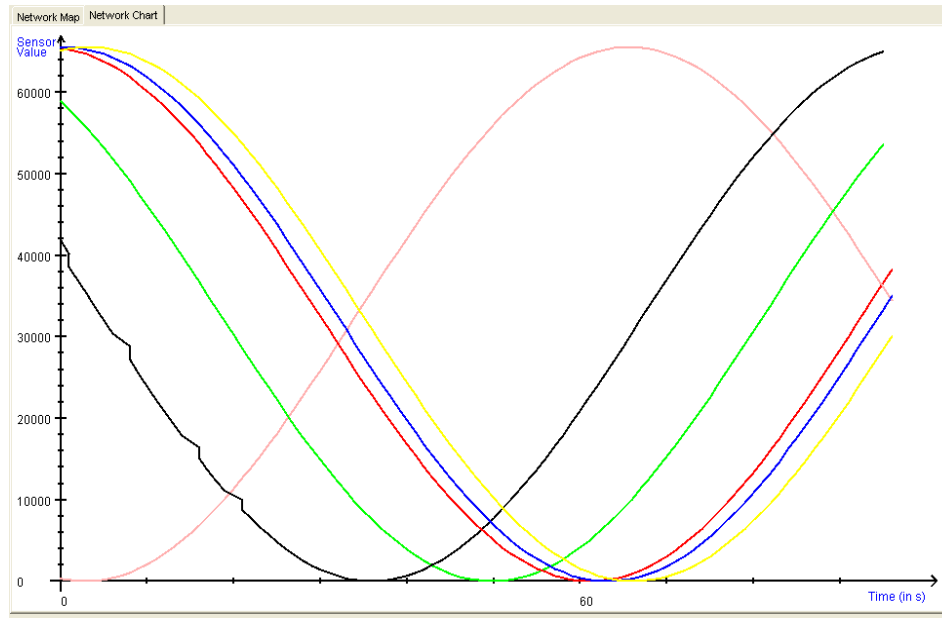


Fig. 11. Octopus Network Chart feature

We observed that delivery rates start dropping after the user issues a request to change the sampling period through the GUI. A request to change the sampling period causes the Octopus GUI to notify the Octopus embedded application running on the mote connected to the PC, which in turn broadcasts a message to all other nodes indicating the new sampling period. Nodes that are one-hop away from the base station first receive this message, process it, and forward it to any downstream neighbors. Downstream forwarding repeats until all nodes have received the instruction to change sampling period.

Upon reception of the change sampling period message by direct neighbors of the base station, each of these nodes resets its sampling period timer and starts a new timer with the new sampling period. Because all of these nodes are direct neighbors of the base station, they receive the sampling period request message, and therefore they set the new timer, at nearly the same time. Under normal conditions, a node sends its data packet when its sampling period timer fires. Because of the implicit synchronization of timers at nodes that have received a request to change their sampling period simultaneously from the base station, these nodes will attempt to send their data packets at virtually the same point in time, causing high contention for the channel and leading to more collisions.

In a periodic monitoring application, the implicit synchronization of nodes not only affects the first data packet transmitted after the change sampling period request, but also all other subsequent data packets until a future change sampling period request is received. At the beginning of every sampling period, all the implicitly synchronized nodes will attempt to transmit their data packets at nearly the same time, causing a recurrence of the initial congestion scenario. This effect can severely harm long-term delivery rates in the network. This implicit synchronization can also occur among nodes two hops

away from the base station, and three hops away, and so on, due to the nearly constant delay per hop of disseminating the change sampling period request.

To avoid this situation, the updated version of Octopus uses a simple dynamic scheduling strategy. This strategy uses information on the maximum number of nodes in the network to schedule transmissions of nodes as far apart as possible within a sampling period. Because all traffic converges to the base station, it monitors all the traffic in the network, and keeps track of the number of nodes in the network. Periodically, the system checks if new nodes have joined the network, and if so, the base station announces the number of nodes in the network. For simplicity, and without loss of generality, the application assumes that all node ID's are set consecutively starting from ID 1, so the highest ID in the network also indicates the number of nodes in the network. As a result, all nodes receive updates on this maximum ID in the network through the MAX_ID downstream message (see Table I), which they can use to locally determine their transmission schedule. Note that upon network startup, the base station waits until the network membership stabilizes before determining and broadcasting the MAX_ID message. Similarly, when a new node joins the network, the base station waits for several sampling periods (5 sampling periods by default) in order to ensure that groups of nodes entering the network at the same time do not cause the transmission of several consecutive MAX_ID messages. In case certain nodes do not receive the most up-to-date MAX_ID message, these nodes simply adhere to their current schedule based on the last MAX_ID they received. Although this may result in a small number of collisions due to mismatched schedules, the underlying IEEE 802.15.4 MAC protocol [24] can handle such collisions through its Binary exponential backoff function.

Upon receiving a request to change its sampling period, a node first stops its current sampling period timer. Then, the node computes a parameter α as follows:

$$\alpha = S / (MAXID + 1); \quad (1)$$

where S is the sampling period and $MAXID$ is the maximum ID in the network. The parameter α essentially determines the maximum guard time between two node transmissions given that the current sampling period is S and that the current number of nodes in the network is $MAXID$. Thus, a large network will have a small guard time between two nodes' transmissions, and a small network will have a larger guard time between two nodes' transmissions. The idea is to spread all the nodes' transmissions as evenly and as far apart as possible within each sampling period.

The nodes then proceed to determine their own application layer backoff time, through the following expression:

$$WaitTimer = (1 + moteID) \times \alpha \quad (2)$$

As a result, node 0 can transmit during the first α time units, while all other nodes refrain from transmitting. At time 2α , node 1 will transmit its data packet, followed by node 2, and so on. Because all the nodes use the same expressions to locally compute their application layer backoff duration, all nodes' transmissions will complete during a single sampling period with minimized risk of collisions. This significantly improves the data packet delivery rate, as our empirical results in section IV confirm. Note that for networks in which the node ID range starts at a number higher than 1, we can simply modify equation 2 to use $moteID$ modulus $MAXID$ instead of $moteID$. Although a random backoff is also possible, the purpose of the application layer backoff here is to optimize the guard times between node transmissions to the current size and sampling period in the network.

To evaluate the impact of this enhancement, we perform systematic empirical experiments in the 20-node dense topology, with and without application layer backoff. For both versions of Octopus, a request to change the sampling period is sent to all nodes in order to expose the implicit synchronization issue. Subsequently, we track the packet delivery rate of each node for 100 sampling periods. Figure 12 illustrates the resulting average delivery rates for sampling periods ranging from 1 to 10 seconds. Comparing the results in the two networks, we note an improvement of 12% to 20% in delivery for the network that uses application layer backoff. This improvement is most pronounced for higher sampling periods, where the backoff schedules nodes' transmissions evenly during the longer sampling period, which provides more time for nodes to successfully complete their transmissions.

When the nodes do not use application layer backoff, the delivery rate is almost constant between 80% and 83%, with no apparent dependence on the sampling period. Since the nodes are implicitly synchronized, they all attempt to transmit during the first 50-100 ms of each new sampling period. As such, most packet transmission and retransmission attempts

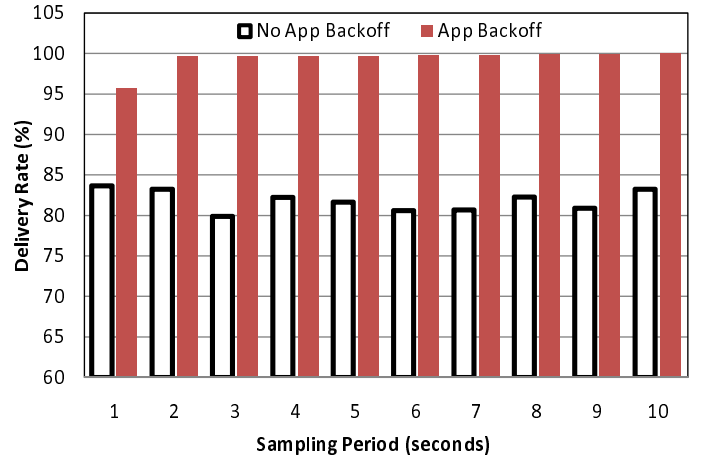


Fig. 12. Impact of application layer backoff for a 20-node network.

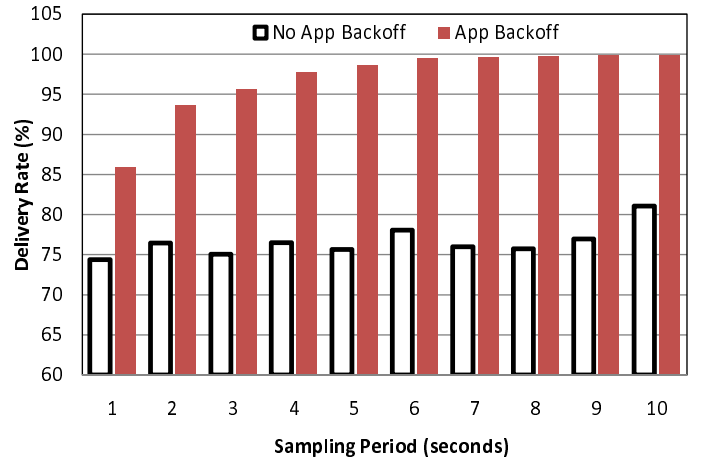


Fig. 13. Impact of application layer backoff for a 40-node network.

are concentrated in the initial portion of the sampling period, so the length of the sampling has no effect on the channel contention and thus the delivery rates.

For the network that uses the application layer backoff, the delivery rate is about 96% for a sampling period of 1 second, and it quickly increases towards 100% delivery at sampling periods of 2 seconds or more. Because application layer backoff schedules the nodes' transmissions evenly across the sampling period, increasing the sampling period provides a higher guard time between consecutive nodes' transmissions, i.e. it provides more time for each node to complete its transmission, which reflects positively on delivery rate.

To explore the performance benefits of application layer backoff further, we have performed the same testbed experiments with a network of 40 nodes. The results of the 40 node network experiments are shown in Figure 13. The delivery rate for the network that does not use the backoff is again nearly constant around 75%. For the backoff-enabled network, the delivery rate is about 85% for a sampling period of 1 second. This value is significantly lower than for the 20-node network, as the network must now support 40 packets every sampling

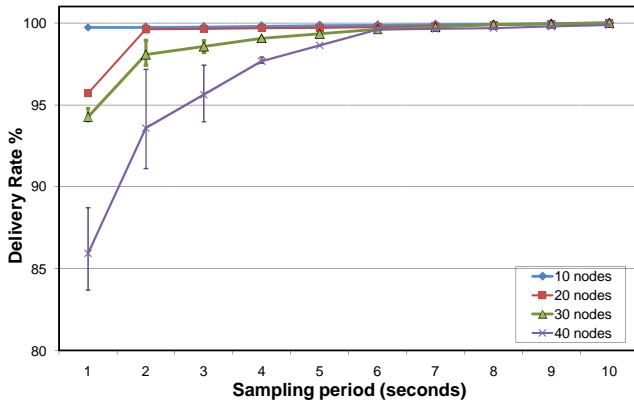


Fig. 14. Delivery rate versus sampling rate

period rather than 20. As a result, attempting to schedule the packets evenly during the 1 second period cannot reduce contention as effectively as for 20 nodes. However, the delivery rate improvement remains about 12% with the backoff for a 1 second sampling period. As the sampling period increases, the delivery rate improvement of having application layer backoffs also increases, as longer sampling periods give more time for scheduling all the nodes' packets. The highest delivery rate improvement appears for sampling period of 8 seconds, where the application layer backoff has a 99.69% delivery rate compared to 75.72% without the backoff, a 24% improvement.

Having confirmed the effectiveness of application layer backoff for improving data delivery with Octopus, we now focus on the impact of varying network level parameters, such as sampling period and duty cycle, on the Octopus network's delivery rate.

B. Network Reconfiguration and Characterization

The objective of the second set of experiments is to leverage Octopus to characterize the relationships between network-level parameters, such as sampling period, network size, and duty cycle, through iterative empirical experiments. While the outcome of this process will be the development of a database of rules that maps combinations of these parameters to network delivery rates, these experiments also demonstrate the versatility of Octopus for changing network parameters through a single click of a button at the GUI. This feature enabled us to gather data from empirical experiments, which are crucial for sensor networks because of the tight coupling with the physical space, with the same ease of conducting computer simulations.

The empirical experiments here first investigate the effect of varying the sampling rate and the network size, while keeping the radio duty cycle constant at 100%. All the experiments were conducted three times with the application layer backoff enabled.

Figure 14 plots the packet delivery rate as a function of data sampling rate, as the sampling rate varies between 1 and 10 seconds, and the network size varies from 10 nodes up to 40 nodes. We chose to perform experiments only up to a sampling

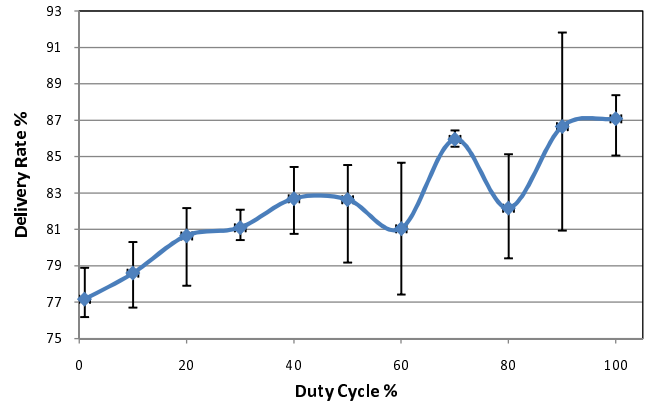


Fig. 15. Delivery rate versus duty cycle

period of 10 seconds because increasing the sampling period further yields delivery rates very close to 100% for longer sampling periods. The intuitive trend we expect to see here is a decrease in delivery rate for lower sampling period, which tend to cause congestion and collisions in the network.

For a small network of 10 nodes, the packet delivery rate is near-optimal for all sampling periods ranging from 1 to 10 seconds, with a value of 99.72% for a sampling period of 1 second that steadily increases to 100% for a sampling period of 10 seconds. Doubling the network size to 20 nodes results in a delivery rate of 95.71% for a 1 second sampling period, highlighting the effect of increased contention for the wireless channel between nodes that are within the same neighborhood and at the same hop count from the base station. For a sampling period of 2 seconds, the 20-node network exhibits a delivery rate of 99.61%. For sampling periods between 2 and 10 seconds, the delivery rate for a network of 20 nodes steadily increases towards a 100% for every step increase in the sampling period.

Increasing the network size further to 30 nodes causes an additional drop in delivery rate at sampling periods between 1 and 6 seconds relative to the 20-node network. The increase in delivery rate for the 30-node network between 1 and 6 seconds appears logarithmic in nature, with a sharp improvement in delivery rate between 1 and 2 seconds, and a more gradual improvement between 2 and 6 seconds. For sampling periods above 6 seconds, the delivery rate is near perfect for the 30-node network, increasing gradually towards 100% packet delivery.

Finally, the larger 40-node network exhibits more noticeable packet losses for smaller sampling periods. The data delivery rate for a sampling period of 1 second is about 86%, and as in the 30-node case, it increases in a logarithmic fashion towards near-perfect delivery for a sampling period of 6 seconds, and remains high for higher sampling periods. These experiments have enabled the characterization of the network performance under several combinations of network size and sampling period, using Octopus's simple reconfiguration interface.

The next set of empirical experiments investigates the effect of varying the duty cycle on the packet delivery rate. For this

purpose, these experiments consider a constant sampling period of 1 second in a 40-node network, in order to demonstrate the impact of changing the duty cycle in a busy and reasonably large network. The experiments consider the following duty cycle values: 1%, 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, 90%, and 100%. Experiments for each duty cycle were repeated at least 3 times with the application layer backoff enabled.

Figure 15 plots the results of these experiments. For duty cycles where we observed high variance, we performed additional experiments to verify the results further. The data points in Figure 15 indicate the average delivery rate for each duty cycle, while the error bars indicate the maximum and minimum recorded values in any of the experiments.

Note that, intuitively, the effective duty cycle is not always the same as the nominal duty cycle [23]. In busy networks, nodes may wake up and sense a busy channel, in which case they may go back to sleep sooner than originally scheduled. This can reduce the effective duty cycle of nodes relative to the nominal value in busy networks.

Returning to the results, the general trend of Figure 15 is that reducing the duty cycle causes a drop in the packet delivery rate. This effect stems from the long sleep periods of nodes, so they are not awake for sufficient time to receive and forward all the packets, causing some of the packets to be discarded at intermediate nodes along the path from the source to the base station. For instance, a duty cycle of 1% results in a nominal packet delivery rate of about 77%, because all of the nodes sleep for 99% of the time, and when they turn on their radios, many neighboring senders may try to reach them for forwarding a packet. As this experiment uses a 40-node network with a 1-second sampling period, the high data rate results in loss of some packets along the way. As we increase the duty cycle to 10% and then 20%, we note a near-linear improvement of delivery rate to 78.59% and 80.65% respectively. The rate of improvement of packet delivery rate slows down when we increase the duty cycle further to up to 50%, because increasing lower duty cycles is more critical for the packet delivery behavior. The experiment results for duty cycles between 60% and 90% are less regular, as the plot exhibits local troughs at 60% and 80%, and a local peak at 70% with high variance among the experiment results for 60% and 80% duty cycles, and extremely low variance for 70%. The highest variance occurs at a duty cycle of 90%, where the delivery rate of some experiments varies by $\pm 6\%$ around the average delivery rate of 87%.

The high variance of packet delivery rate at certain duty cycles reveals some sensitivity of MAC and routing mechanisms in TinyOS 2.x for these duty cycles, particularly 60%, 80% and 90%. The ability to toggle duty cycle online and to observe resulting delivery rates has been central for revealing this behavior in TinyOS. The exact nature of the fix for this issue remains an open issue for future investigation. Overall, the packet delivery rate improves from about 77% to 87% as we increase the duty cycle from 1% to 100%. Note that the result for the 100% duty cycle is in line with the range

of results in Figure 14, where the average duty cycle for the 40-node network at a duty cycle of 100% is close to 86%.

Extending the duty cycle results, we investigate the effect of varying duty cycle on packet delivery delay in Octopus. This experiment involves data aggregation at a specific hop level in the tree, based on the adaptive fidelity aggregation proposed in [9]. Upon receiving a packet for forwarding, a node simply compares its current hop count with the set aggregation level. If the node's hop count is higher than the aggregation level, it buffers the incoming packets until its next timer fire event. When its timer fires, it combines the incoming packet with its own and forwards the combined packet. The buffering of packets for the purpose of aggregation incurs additional delay for aggregated packets, which provides a higher delay scenario for our experiments. To measure delivery delay in our asynchronous network, the gateway monitors the packet arrival time of all packets. For a particular node, the inter-packet arrival time is a strong indicator of packet delay. All of our experiments use a data reporting period of 1 second. If the packet inter-arrival time for a particular experiment is 1.7 seconds, then the network is introducing 0.7 seconds of delay for delivery. Our delay experiments include 20 telosb nodes in a 5-hop network topology. The experiments vary the aggregation level from 1 to 5. An aggregation level of 5 is equivalent to Octopus operating with no aggregation. An aggregation level of 1 means that all packets in the network are aggregated at every level. The experiments were done for three radio duty cycles of: 10%, 30%, and 90%.

Figure 17 illustrates the inter-packet arrival time for these experiments. Each point in the figure represent the average inter-packet arrival time for all 20 nodes for the experiment. As expected, the delay is higher for lower radio duty cycles. The results for an aggregation hop of 5 represent the delay of Octopus upstream packets with no aggregation. These results indicate an upstream packet delay between 0.28 and 0.44 seconds, depending on the radio duty cycle. Given that packet transmission times and channel backoffs for the CC2420 radio are in the order of tens of milliseconds at each hop, Figure 17 suggests that Octopus introduces virtually no delay to upstream packets. As the aggregation hop moves closer to the gateway, the delivery delay increases for all duty cycles, as more packets are buffered at intermediate hops awaiting aggregation with local packets. This buffering delay for networks that rely heavily on aggregation dominates the packet delivery delay, rendering Octopus's minor delivery delay even smaller for high aggregation networks.

V. DISCUSSION AND FUTURE WORK

Our empirical experiments have shown the network-level diagnostic and debugging functionality of Octopus through easy manipulation, visualization, and logging of network state. This functionality has enabled us to uncover limitations in the design of original design of Octopus, namely the implicit synchronization after setting a new sampling period.

Our empirical experiments have also demonstrated the use of Octopus as network characterization tool. By varying the

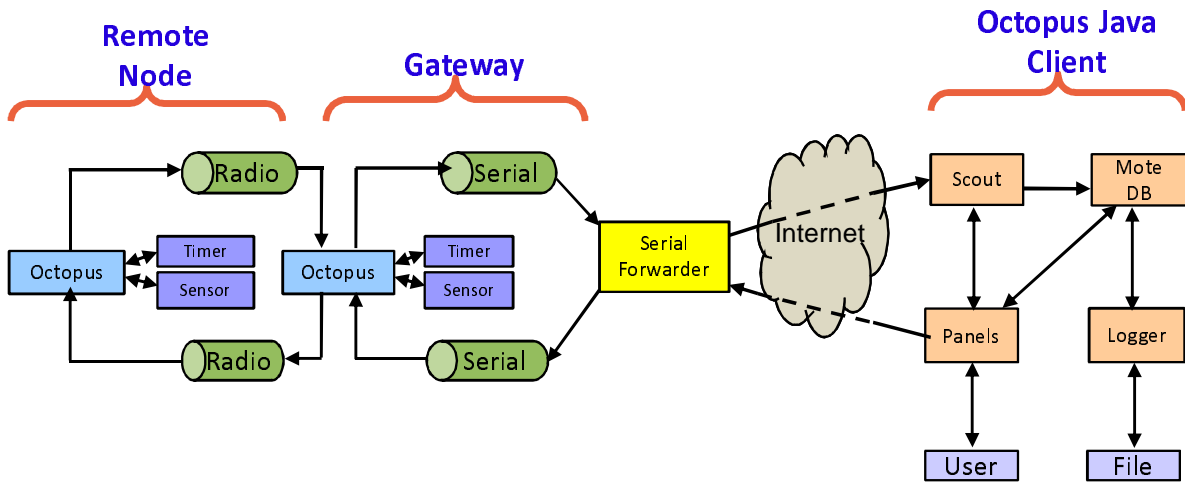


Fig. 16. Web-enabled Octopus client/server structure.

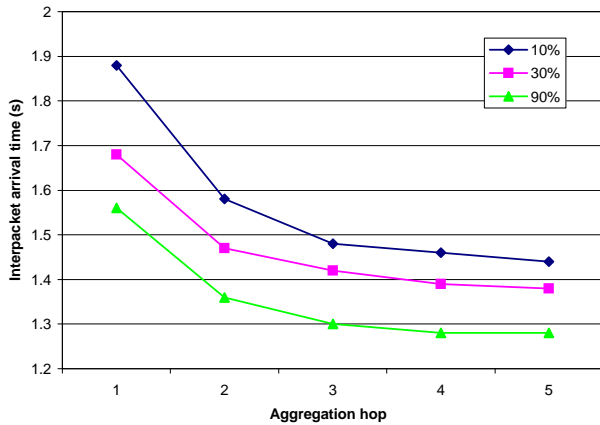


Fig. 17. Interpacket delivery delays

sampling rate through Octopus as one would in multiple simulation runs, we collected delivery rate results through iterative experiments. These results confirm the importance of the network size to sampling period ratio in convergecast networks, as suggested previously in [23]. This ratio provides a strong indication of the effective data rate and the expected delivery rate. As the network size increases, the likely neighborhood size also increases if the network is covering the same physical area. Obviously, covering the same area with more nodes results in a more dense network. The increased neighborhood size (in terms of nodes) causes more contention on the channel and thus the network can guarantee near-perfect delivery for only longer sampling periods. The other effect is the congestion at the critical nodes, where a larger network size can cause convergence of a large number of packets at these nodes, causing a drop in delivery rate.

The results above begin to establish network-level constraints for particular applications. For example, consider an application that requires deployment of 40 nodes with a guaranteed delivery rate of 95%. Through the results in Figure 14, the lowest sampling period that can support the

required delivery rate is 3 seconds. If 10 nodes leave the network, the base station can then reduce the network sampling period to 2 seconds while maintaining a delivery rate guarantee of 95%. Alternatively, users can provide their required network size and sampling period to obtain an estimate on the delivery rate guarantee.

Our final set of experiments that varies the duty cycle add a new dimension to the rules database. These experiments have actually led us to uncover erroneous behavior in the low power listening implementation of TinyOS version 2. During these experiments, we have found Octopus to exhibit stable and predictable behavior, except when varying duty cycles between 60% and 90%. Exploring this issue further, we determined that the discrete formulas in the TinyOS LPL implementation truncate duty cycle values, which is a possible cause for the unstable behavior for duty cycles between 60% and 90%. However, this issue requires further investigation. In addition, the delay experiments reveal that Octopus introduces no delay for networks with no aggregation.

The high visibility into the live network topology and the ability of developers and users to modify the network, whether manually or through remote configuration requests, has proven to be useful for uncovering logical bugs in the implementation of the TinyOS and in the original implementation of Octopus itself. Uncovering these bugs through conventional tools that provide separate control or monitoring would have been significantly more time consuming and tedious. By combining network monitoring and control into a single platform, Octopus enables sensor network users and developers to monitor their network and to configure it through an intuitive interface. Unlike existing tools, Octopus's independence of the underlying communication protocols allows developers to readily reuse its code in testing or deploying different protocols depending on their functional requirements.

The results from the empirical experiments also lay the groundwork for enabling autonomous network reconfiguration in Octopus. For example, based on their application, users can

set a range for the acceptable sampling period for the nodes and the desired level of delivery rate guarantee. The Octopus base station first determines and sets the sampling period to a value within the acceptable user-defined range and that satisfies delivery rate guarantee. Once the nodes are deployed, the base station can continuously monitor the network size based on the incoming data packets from the network. The base station can use the results from the empirical experiments to determine the optimal duty cycle for the current network size, sampling period and target delivery rate. Obviously, the results in Figure 15 only present results for a sampling period of 1 second and a network of 40 nodes. However, carrying out the same type of experiments for different network sizes and sampling period can build up the knowledge base for adapting to a wider variety of situations. Setting network-level parameters on the basis of a comprehensive rules database can be incorporated with existing distributed adaptation approaches, so that the nodes can adapt to high frequency state changes locally, whereas the base station can set network level parameters centrally, as changes in these parameters are required less often.

Octopus currently supports 16 request message types. However, the current message format reserves 8 bits for the message type, which supports up to 255 message types, thus enabling developers to add custom request message types for their given applications. To do so, developers would simply define the new message types and the corresponding parameters to be sent within these messages. Developers can then add the desired node response in the Octopus code by simply adding a new condition to the existing *switch* statement in the `processRequest()` function within the Octopus application code.

Currently, the Octopus Java application runs mainly as a stand-alone Java application. However, the current implementation also supports running Octopus within a client/server architecture. In fact, we have verified this functionality of Octopus already with the support of the `SerialForwarder` Java application in TinyOS, as shown in Figure 16. At the base station, `SerialForwarder` connects to the incoming raw data from the attached sensor node. `SerialForwarder` acts as a server for incoming local and remote client connections over TCP/IP networks. The Octopus Java application runs as a client within this architecture. Multiple remote Octopus clients can simply establish a connection to the local `SerialForwarder` server and provide multiple users with access to the network. An interesting direction for future work is to provide an Octopus client plugin into widely used web browsers and to provide easier connectivity to a remote running server.

Another desirable feature for future development in Octopus is the capability to easily plug in to existing back-end monitoring tools that are designed for other sensor network programming environments. An example is the Fleck Operating System (FOS) [26], which uses several Python-based tools for network health checking at the base station. Providing a suitable interface between these Python tools and Octopus's Java client can render Octopus readily compatible with FOS-based

application. Octopus's plug-in capability should also minimize changes to legacy application code for these programming environments. This demands that the message classes for the Octopus Java portion are adapted to accept data packets of the legacy application. Because some applications may not include all standard Octopus message fields, the new message class compilation must provide flexibility and fall-back options. For instance, if an application does not include link quality metrics in upstream packets, the link quality functionality in Octopus should be disabled at compile time.

While we have recently successfully tested the Octopus GUI with machines that do not run TinyOS, the design concepts of Octopus are readily applicable to other sensor network environments, such as Contiki [25] or FOS [26]. These design concepts include the combination of monitoring, visualization and control into one tool, and to embed the network monitoring tool into the monitoring application. In addition, Octopus is independent of underlying routing protocols for full portability across different protocols and platforms. It provides centralized control of network-level parameters and distributed adaptation of node-level parameters, as a compromise between informed network decisions and responsive node behavior.

In sum, this paper has presented the open-source Octopus network control, debugging and monitoring tool. Octopus provides live network topology monitoring, data logging and plotting information for debugging sensor network deployments. It also enables users to send predefined requests to the sensor nodes to adapt their configuration. Octopus was leveraged through extensive experiments under differing network sizes, sampling rates, and duty cycles. In addition to uncovering application and OS bugs, the experiment results also represent a step towards building a knowledge basis of the dependencies between key network-wide parameters, which will enable a hybrid control strategy that relies on both local distributed communication from the nodes and more comprehensive network state information arriving at the base station. Octopus is an enabling tool for driving sensor network research and development forward.

REFERENCES

- [1] T. C. Henderson, J. C. Park, N. Smith, R. Wright. "From Motes to Java Stamps: Smart Sensor Network Testbeds," University of Utah Tech. Report UUCS-03-003, 2003.
- [2] TinyOS Network Programming J. Hui. <http://www.cs.berkeley.edu/~jwhui/deluge/>
- [3] P. Levis, S. Madden, D. Gay, J. Polastre, R. Szewczyk, A. Woo, E. Brewer, and D. Culler. "The Emergence of Networking Abstractions and Techniques in TinyOS," In proc. *1st Symposium on Network Design and Implementation*, 2004.
- [4] Mviz. <http://www.tinyos.net/tinyos-2.x/apps/MViz/>
- [5] G. Werner-Allen, P. Swieskowski, and M. Welsh. "MoteLab: A Wireless Sensor Network Testbed," In proc. IPSN SPOTS'05, 2005.
- [6] V. Handziski, A. Kopke, A. Willig, A. Wolisz. "TWIST: A Scalable and Reconfigurable Testbed for Wireless Indoor Experiments with Sensor Networks," In proc. international workshop on Multi-hop ad hoc networks: from theory to reality, pp. 63 - 70, 2006.
- [7] L. B. Ruiz, J. M. Nogueira A. A. F. Loureiro. "MANNA: A Management Architecture for Wireless Sensor Networks," *IEEE Communications Magazine*, February 2003.

- [8] G. Wagenknecht, M. Anwander, T. Braun, T. Staub, J. Matheka, and S. Morgenthaler. "MARWIS: A Management Architecture for Heterogeneous Wireless Sensor Networks," In proc. *6th International Conference on Wired/Wireless Internet Communications 2008 (WWIC08)*, 2008.
- [9] R. Jurdak, A. Nafaa, and A. Barbirato. "Large Scale Environmental Monitoring through Integration of Sensor and Mesh Networks," *Sensors*, 8(11):7493-7517, 2008.
- [10] Tiny Operating System. UC Berkeley. available: <http://tinys.net>.
- [11] T. ElBatt and A. Ephremides. "Joint scheduling and power control for wireless ad hoc networks". *IEEE Transactions on Wireless Communications*, vol. 1, pp. 74–85, 2004.
- [12] R. Jurdak, P. Baldi, and C.V. Lopes. "Adaptive Low Power Listening for Wireless Sensor Networks," *IEEE Transactions on Mobile Computing*, Volume 6(8):988–1004, August, 2007.
- [13] R. Winter, J. Schiller, N. Nikaiein, and C. Bonnet. "CrossTalk: A Data Dissemination-based Crosslayer Architecture for Mobile Ad-hoc Networks". In Proc. *Applications and Services in Wireless Networks*, 2005.
- [14] K. Sohrabi, J. Gao, V. Ailawadhi, and G. J. Pottie. "Protocols for Self-Organization of a Wireless Sensor Network", *IEEE Personal Communications*, 2000.
- [15] T. C. Collier and C. Taylor. "Self-Organization in Sensor Networks", *Journal of Parallel and Distributed Computing*, 2004.
- [16] D. Gay, P. Levis, R. von Behren, et al. "The nesC language: A holistic approach to networked embedded systems". *ACM SIGPLAN Notices*, 38(5):1–11, 2003.
- [17] Crossbow Technologies Inc. Crossbow datasheet on MicaZ. available: http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/MICAZ_Datasheet.pdf (2008).
- [18] Crossbow Technologies Inc. Crossbow datasheet on TelosB. [xbow.com/products/product_pdf_files/wireless_pdf/telosb_datasheet.pdf](http://www.xbow.com/products/product_pdf_files/wireless_pdf/telosb_datasheet.pdf) (2008)
- [19] T. Stathopoulos, T. McHenry, J. Heidemann, and D. Estrin. "Remote Code Update Mechanism for Wireless Sensor Networks". CENS Technical Report # 30, 2003.
- [20] S. S. Kulkarni and L. Wang. "MNP: Multihop network programming for sensor networks". In Proc. "International Conference on Distributed Computing Systems", pages 716, June 2005.
- [21] PE Lanigan, R Gandhi, P Narasimhan. "Sluice: Secure Dissemination of Code Updates in Sensor Networks". In Proc. *International Conference on Distributed Computer Systems (ICDCS)*, 2006.
- [22] P. Levis and D. Culler. Mate: A Tiny Virtual Machine for Sensor Networks. In proceedings of *ACM SIGOPS Operating Systems Review*,
- [23] J. Polastre, J. Hill, and D. Culler. "Versatile Low Power Media Access for Wireless Sensor Networks". Proceedings of *ACM SenSys*, 2004. 2002.
- [24] IEEE Standard for Information technology *Local and metropolitan area networks: Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (LR-WPANs)*, October 2003
- [25] Dunkels, A., Gronvall, B., and Voigt, T. "Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors," In Proceedings of the *29th Annual IEEE international Conference on Local Computer Networks* (November 16 - 18, 2004). LCN.
- [26] P. Corke and P. Sikka. "Demo abstract: FOS a new operating system for sensor networks," In *Fifth European conference on wireless sensor networks (EWSN 2008)*, Bologna, Italy, Jan, 2008.
- [27] Levis, P., Patel, N., Culler, D., and Shenker, S. "Trickle: a self-regulating algorithm for code propagation and maintenance in wireless sensor networks," In Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation - Volume 1 (San Francisco, California, March 29 - 31, 2004).