DESAL$^\alpha$: An Implementation of the Dynamic Embedded Sensor-Actuator Language

Andrew R. Dalton*, William P. McCartney†, Kajari GhoshDastidar‡, Jason O. Hallstrom*, Nigamnath Sridhar*, Ted Herman†, William M. Leal§, Anish Arora‡, Mohamed Gouda‡

*Clemson University, †Cleveland State University, ‡University of Iowa, §Ohio State University, ¶University of Texas

Abstract—We present DESAL$^\alpha$, a realization of the Dynamic Embedded Sensor-Actuator Language for Telos-based devices. The platform provides native support for (i) rule-based programming, (ii) synchronized action scheduling, (iii) neighborhood management, and (iv) distributed state sharing. We describe the design and implementation of DESAL$^\alpha$, present examples that illustrate its use, and summarize the resource requirements of compiled applications. Finally, we present lessons learned based on our use of DESAL$^\alpha$ during the past year.

Index Terms—Sensor network, programming language, runtime system, guarded command, synchronization, dynamic binding

I. INTRODUCTION

There is a mismatch between the target user audience for wireless sensor networks, and the tools used to construct them. Agricultural scientists, civil engineers, hydrologists, and other data consumers benefit from the instrumentation fabric these networks provide — computing professionals do not. Yet standard programming platforms cater to the latter group. Developers must master low-level programming primitives, operating system components, and network protocols. It should be easier. We present a sensor network programming platform engineered to achieve this goal.

DESAL$^\alpha$ is the first realization of the Dynamic Embedded Sensor-Actuator Language. The realization targets Telos-based devices and supports the core features of the language and runtime vision proposed in [1]. The platform is characterized by four key features:

Rule-Based Programming. DESAL$^\alpha$ programs are rule-based. Program behavior is specified using guarded actions defined in terms of program state. There is no hidden control context (e.g., pending tasks, events, interrupts).

Synchronized Action Scheduling. DESAL$^\alpha$ programs coordinate through synchronized actions. Components are activated for action selection at a specified periodicity. Activation periods are automatically synchronized across the network.

Neighborhood Management. DESAL$^\alpha$ programs have built-in access to the network neighborhood of the hosting device. Neighborhood management services, including node discovery and health monitoring, are performed automatically.

Distributed State Sharing. DESAL$^\alpha$ programs communicate via state sharing. There is no network programming required. Instead, programs declare bindings to the shared variables exposed within the neighborhood of their host.

When combined, these features enable a new programming paradigm for wireless sensor networks — one that simplifies program construction and offers potential reliability improvements. The goal is to improve the accessibility of wireless sensor networks for those communities most likely to benefit from their deployment.

Paper Organization. Sections II and III present the DESAL$^\alpha$ language and runtime system, respectively. Section IV presents application examples. Section V characterizes the resource requirements of compiled applications. Section VI considers elements of related work. Finally, Section VII concludes with lessons learned while using DESAL$^\alpha$ during the past year.

II. LANGUAGE DESIGN

An abbreviated DESAL$^\alpha$ grammar appears in Listing 1. Trivial productions have been omitted. Non-terminals such as (subcmpnts) and (bindings) represent repititions of non-terminals (subcmpnt) and (binding), respectively. Non-terminals such as (cid) and (vid) represent component and variable identifiers, respectively. The non-terminal (num) represents a constant expression evaluating to a natural number.

Each DESAL$^\alpha$ application is implemented by a single component, (cid), which contains one or more subcomponents (line 1). A subcomponent is a unit of scope consisting of variable declarations, binding declarations, and a body containing guarded actions (line 2). Variable declarations follow

Listing 1: Abbreviated DESAL$^\alpha$ Grammar

C-style syntax (line 3); each consists of a type specifier\(^2\) and an identifier (line 4). In DESAL\(^a\), however, a declaration may be prefixed with the unshared keyword to indicate that the variable is node private, or the shared keyword to indicate that the variable may be read by neighboring nodes. Access to shared variables is achieved through a corresponding binding.

Two types of bindings are supported (line 5). The first is the singleton binding, used to establish a local name for an instance of a remote variable (line 6). The \(\langle\text{dec}\rangle\) portion of the binding declaration specifies the type of the end-point variable and assigns the local identifier used to access its value. The right-hand side of the declaration (i.e., after the \(\langle\text{dec}\rangle\)) specifies the end-point. This includes the identifier of the hosting node, \(\langle\text{nid}\rangle\), the identifier of the declaring component, \(\langle\text{vid}\rangle\), and the identifier of the desired variable, \(\langle\text{vid}\rangle\). The binding is used as a proxy for the variable when implementing guards and actions. Its value always reflects, with some delay, the current value of its end-point. If an end-point fails due to a node or network fault, the binding value remains fixed at the last known end-point value. We will consider the syntax used for detecting binding failures shortly.

In some cases, it is useful to change the end-point of a binding. In a routing component, for instance, an action may need to change the end-point when an improved route to the basestation is identified — or, alternatively, in the event of an end-point failure. To support rebinding, a singleton declaration may specify a uint16 variable in place of the node identifier \(\langle\text{nid}\rangle\). When a program action assigns to this variable, the end-point of the binding is automatically changed.

The second type of binding is the multi-binding (line 7), used to access a set of end-points. The declaration syntax is similar to that used in declaring a singleton binding, but introduces a "*" in place of the node identifier. This indicates that the binding accepts multiple end-points — namely, all instances of the variable shared within the host’s neighborhood. When a multi-binding is used in an action, it is treated as a set containing the current values of its end-points. If an end-point fails, the corresponding value is automatically removed. We will consider the syntax for accessing this set shortly.

To prevent resource exhaustion, each multi-binding specifies an upper-bound, \(\langle\text{num}\rangle\), on the number of end-points that it can accept. In some scenarios, the number of candidates can exceed capacity. In such a case, the binding accepts a subset of the candidates, chosen non-deterministically. The end-points remain fixed unless there is a binding failure, in which case the failed end-points are replaced with other candidates.

The body of a DESAL\(^a\) subcomponent is executed at a specified periodicity, \(\langle\text{num}\rangle\), after some initial delay, \(\langle\text{num}\rangle\) (lines 8–9). The \(\langle\text{unit}\rangle\) specifier indicates the unit of time being used in the periodicity and delay specifications: milliseconds, seconds, minutes, or hours (line 10). For instance, a body specification that begins with every 500ms after 5s indicates that the actions within the body are eligible for execution every half second after a delay of five minutes. When a subcomponent is activated for execution, all of its guards are evaluated. The enabled actions (i.e., those with guards evaluating to true) are then executed sequentially. After all enabled actions are executed, the subcomponent enters an idle state until its next activation.

The delay specification requires further explanation: The value does not refer to a delay that begins at node startup. Instead, it refers to the time at which periodic behavior should begin, as measured by a global, synchronized clock. Network time synchronization is provided automatically. Hence, subcomponents with the same periodicity and delay specifications are activated at the same instant across the network, enabling coordination of distributed actions.

An action consists of a boolean guard and a list of statements (lines 11–12). We omit standard productions for expressions, but note that common arithmetic, relational, and bitwise operators are supported. The statement list of an action may include assignment statements, conditional statements, binding iteration statements, and calls to system functions. We again omit standard productions and focus on the last two syntactic elements.

The foreach statement is an iteration construct used to access the end-point values within a multi-binding (line 13). It is analogous to the foreach construct provided by Java and C# for enumerating elements within a collection. In each iteration, the variable \(\langle\text{vid}\rangle\), scoped to the iteration body, takes on the next end-point value within \(\langle\text{bid}\rangle\). The order of the iteration is non-deterministic. We again note that failed end-points are pruned automatically; they are not enumerated by foreach.

Binding functions are used to access information about a binding end-point (line 14). Each may be applied to either a singleton binding or a value accessed during a foreach iteration. Hence, in the \(\langle\text{bindfunc}\rangle\) production, \(\langle\text{bid}\rangle\) refers to a singleton identifier or the identifier of a binding iteration variable. The latter application enables access to information about each of the end-points contained in a multi-binding.

The first function, bound(), returns a boolean value indicating whether the binding end-point is healthy. In the case of a multi-binding, the function always returns true, as dead end-points are automatically pruned. In the case of a singleton binding, a return value of false indicates binding failure. The second function, src(), returns the node identifier of a binding end-point. The function is most often used within a foreach block to identify the node end-points within a multi-binding. Finally, age() is used to determine the freshness of a binding value. The return value indicates the number of milliseconds that have elapsed since the binding value was last updated from the end-point.

Seven sensor functions are provided to retrieve sensor data (line 15–16): The first four functions correspond to the standard sensors included on the Telos platform. Specifically, the functions correspond to temperature (\(\text{\$temp()}\)), humidity (\(\text{\$humidity()}\)), total solar radiation (\(\text{\$totalSolar()}\)), and photosynthetically active radiation (\(\text{\$par()}\)) sensors. The next two functions provide generic support for analog sensors connected to ADC

\(^2\)The DESAL\(^a\) compiler currently supports a limited set of types: bool, uint8, uint16, and uint32.
Fig. 1: DESAL$^\alpha$ Application Architecture

The DESAL$^\alpha$ compiler uses TinyOS [2] as the compilation target. The choice simplifies code generation, leverages existing drivers, and takes advantage of the optimizations performed by the nesC compiler [3] chain. The architecture of a translated application is illustrated in Figure 1. Blue modules are common to all applications, yellow modules are compiler-generated, and the edges between them represent dependencies. The Binding Manager is a conceptual module introduced for the sake of exposition; its functionality is implemented by the Soft State Store. The orange module represents a Java application running on a serial-attached basestation.

DESALMain implements the state elements and guarded actions specified by the input component. Unshared DESAL$^\alpha$ variables are translated to nesC module variables of the same type. Shared variables and bindings are translated to handle variables used to interact with the Soft State Store. (We will consider this module shortly.) Each subcomponent body is translated to a single function that contains a list of if-then blocks that implement the corresponding guards and actions. Each is executed as a separate thread using the TinyThread library for TinyOS [4].

The use of TinyThread simplifies code generation by eliminating the need to generate split-phase execution logic. This is achieved by developing threaded wrappers that provide blocking semantics over standard split-phase calls. These wrappers are particularly useful in generating code involving sensor and communication functions. Consider, for example, the sensing functions $\text{Stemp}()$ and $\text{Shumid}()$. In TinyOS, the corresponding driver modules provide call-back semantics, complicating code generation — especially in the case of nested or iterative calls. Using the TinyThread approach, each function is translated to a single blocking call on a wrapped driver.

Threads are activated based on the periodicity and delay parameters specified by their corresponding subcomponent body. These activations are handled by the Activation Scheduler. The module provides an interface for requesting activation events at a specified periodicity, after some initial delay. At boot time, an activation schedule is requested for each thread. Internally, the module uses one-shot timers tuned to the system clock. The timer delay for a given thread is set to $(\text{time} + \text{delay}) \text{mod} \text{period}$, where $\text{time}$ denotes the current value of the clock, $\text{delay}$ represents the requested delay, and $\text{period}$ represents the requested period. When the timer fires, the activation event is signaled on the body thread, and the one-shot timer is again set according to the same formula.

The $\text{time}$ value used by the Scheduler is retrieved from the host’s clock. To synchronize activations across the network, the Time Sync module implements a time synchronization protocol. The current implementation uses a beaconing approach that converges to the lowest clock value in the network. We estimate the precision to be on the order of tens of milliseconds across nodes within the same neighborhood. This level of precision has been suitable to our applications. If more stringent guarantees are required, any of the numerous time synchronization services could be used, with only minor adjustments to its interface.

The DESAL$^\alpha$ syntax for statements and expressions parallels the nesC syntax, simplifying the translation of guards and actions. One exception is in translating references to shared variables and bindings. These elements are allocated within the Soft State Store and accessed through corresponding handles. In the case of a shared variable or singleton binding, the handle encodes the allocation address within the Store. R-value references are translated to read calls on the Store, passing the appropriate handle as argument. L-value references are similarly translated to write calls on the Store. Multi-bindings require additional state: The handle encodes the address of a slot list containing the end-point values and a reference to the current slot in a binding iteration. This state is used by the iteration commands provided by the Store to access the slots in a multi-binding (i.e., reset(), hasNext(), next(), current()). These commands are used in translating foreach statements.

Shared variables are accessed through the Store to capture updates that must be propagated through the network. When a variable is altered, the Soft State Transport is notified. The module is responsible for broadcasting the update to neighboring nodes. In addition to write-triggered broadcasts, the module periodically rebroadcasts all shared variables allocated within the store. This strategy provides neighbors with timely variable updates and also protects against message loss and data corruption.

When an update message is received by the Transport, it signals the Soft State Store. In turn, the Store searches its binding table for an element bound to the identifier recorded in the message. Compiler-assigned numeric identifiers are used for storage efficiency, and the binding table is sorted by variable identifier to support efficient search. In the case of a singleton binding, the value is updated, and an update timestamp is recorded. For a multi-binding, the Store must determine whether the source of the update is already repre-
The periodicity and delay specification instructs next, the component declares its first body Blink application distributed with TinyOS. When installed on a host, the application causes the device to turn its red LED for a duration of 1 second, followed by an off period of 1 second (lines 4–5). Hence, the red LED will be turned on at $t = 0, t = 3, t = 6$, and turned off at $t = 1, t = 4, t = 7$, yielding the desired behavior.

Note that the program listing includes the complete application source; no code elements have been omitted. There is nothing in the program text concerning distributed coordination. And yet, when Blink is installed on a network, all nodes within the network blink in a coordinated manner. Indeed, all DESAL$^\alpha$ applications are endowed with this coordination support automatically.

### B. MaxTemp

The next example, MaxTemp, involves a network computation. The program identifies the maximum temperature reading in a network and propagates the maximum to every node. The application source is shown in Listing 3.

Unlike Blink, MaxTemp maintains state: The component defines the shared variable, max, used to store the current network-wide maximum known to the host (line 2). Additionally, it defines a multi-binding used to access the $max$ value stored at each of the host’s neighbors (line 3). The upper-bound indicates that the binding can store (up to) 20 end-points.

The component defines two bodies. The first body includes two actions, executed once every second (lines 4–9). The first action includes a guard that compares the local temperature to the current maximum. If the local temperature exceeds the known maximum, the action is enabled and the maximum is updated. The second action is similar, but considers the values stored by neighboring nodes (lines 7–9).

The component defines two bodies with no shared variables or bindings. The program begins with the component identifier, Blink (line 1). Next, the component declares its first body (lines 2–3). The periodicity and delay specification instructs the DESAL$^\alpha$ scheduler to activate the body every 3 seconds, with no initial delay. Its single action has a trivial guard of true, indicating that the action will be executed during each activation. The statement turns on the host’s red LED. The second body turns off the host’s red LED every 3 seconds, but after a delay of 1 second (lines 4–5). Hence, the red LED will be turned on at $t = 0, t = 3, t = 6$, and turned off at $t = 1, t = 4, t = 7$, yielding the desired behavior.

### IV. Application Examples

We now consider using DESAL$^\alpha$ to construct a sensor network application. Three demonstrative examples are presented.

#### A. Blink

The first application is a DESAL$^\alpha$ variant of the standard Blink application distributed with TinyOS. When installed on a host, the application causes the device to turn its red LED on for a duration of 1 second, followed by an off period of 2 seconds. This process repeats indefinitely. The application source, shown in Listing 2, illustrates DESAL$^\alpha$’s periodic execution model.

The component defines two bodies with no shared variables or bindings. The program begins with the component identifier, Blink (line 1). Next, the component declares its first body (lines 2–3). The periodicity and delay specification instructs
being a network-centric application, the program does not include any communication primitives. All communication is handled by the DESAL α runtime. Indeed, this is true of all DESAL α applications; there is no concept of messaging — all communication occurs through shared state.

C. PlantMonitor

The final example is a system designed to monitor, from a remote location, the soil moisture of household plants. The moisture sensors [5] are attached to the Telos platform via an ADC pinout, and the adc0() primitive is used to access the corresponding data. Each host periodically samples a plant’s soil moisture. A basestation-attached host collects these samples and transmits the results to the basestation. Moisture values are then published to the Internet via Microsoft’s SenseWeb³. The application source is shown in Listing 4.

The component declares two shared variables and two multi-bindings. The shared variables, adc0Val and voltVal, are used to store the host’s current sensor reading and internal voltage, respectively (lines 2–3). (The latter value is required to interpret the sensor data.) The multi-bindings are used to access these values on (up to 5) neighboring nodes (lines 4–5).

The program consists of a single body with a single action, executed each second (lines 6–12). First, the action toggles the mote’s red LED as a visual heartbeat. Next, soil moisture is sampled using adc0(). In the following statement, the host’s identifier, ID, is stored in the high-order bytes of adc0Val. This packaging approach tags each value with its corresponding source. While unnecessary at the node level —$src() provides this information automatically— this enables the basestation to associate values with devices. The last two statements perform an analogous function, sampling the node’s internal voltage and tagging the value with the host identifier. We note that the packaging approach is an artifact of the compiler’s implementation status; struct types are unsupported. We are currently completing support for structs.

Note that the two bindings, adc0ValL and voltValL, are never used in the DESAL α program. Instead, these values are inspected through the Soft State Store Bridge⁴. The basestation periodically activates the Bridge to inspect the values within

\( \alpha \)

Listing 4: DESAL α PlantMonitor Source

<table>
<thead>
<tr>
<th>Body Count</th>
<th>ROM</th>
<th>RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>14144</td>
<td>1207</td>
</tr>
<tr>
<td>2</td>
<td>14210</td>
<td>1397</td>
</tr>
<tr>
<td>3</td>
<td>14574</td>
<td>1587</td>
</tr>
<tr>
<td>4</td>
<td>14650</td>
<td>1777</td>
</tr>
<tr>
<td>5</td>
<td>14730</td>
<td>1967</td>
</tr>
<tr>
<td>6</td>
<td>14806</td>
<td>2157</td>
</tr>
<tr>
<td>7</td>
<td>14892</td>
<td>2347</td>
</tr>
<tr>
<td>8</td>
<td>14950</td>
<td>2537</td>
</tr>
<tr>
<td>9</td>
<td>15022</td>
<td>2727</td>
</tr>
<tr>
<td>10</td>
<td>15096</td>
<td>2917</td>
</tr>
</tbody>
</table>

(a) Subcomponent Bodies

(b) Sensor/Actuator Drivers

TABLE I: Basic DESAL α Requirements

the bindings and uses Microsoft’s SenseWeb libraries [6] to post the results to SensorMap [7].

V. RESOURCE REQUIREMENTS

We now consider the resource requirements of DESAL α applications. All results are based on tests performed using the TelosB platform.

The DESAL α compiler removes runtime components unused by an application. If, for instance, an application does not declare any shared variables or bindings, the Soft State Store is not included in the generated program image. Hence, we begin by considering the baseline requirements of the DESAL α runtime as a function of subcomponent declarations. Recall that each subcomponent body requires a separate thread. Each thread in turn requires a separate stack, which impacts RAM usage. Further, each body introduces initialization calls (e.g., to initialize threads, to schedule activations), which impact ROM usage. Table Ia summarizes the memory requirements associated with a DESAL α application as a function of the empty subcomponent bodies it contains. (An empty body declares a single action with a guard of true⁵: this represents baseline in the remaining figures.) There is a constant cost of 90 bytes of RAM for each body; the size of the program image also grows linearly with the number of bodies.

Table Ib summarizes the resource requirements of the DESAL α system drivers. The values do not reflect the requirements of the system calls, but rather, the (TinyThread-wrapped) drivers that must be introduced to satisfy the calls. The overhead listed for volt(), for example, is incurred for the first call only; subsequent calls incur negligible overhead. Beginning from baseline, each system call was introduced, exclusive of the others, to calculate the remaining values. The values for $temp(), for instance, represent the RAM and ROM requirements for an application that includes a single action that invokes $temp(), with a trivial guard of true. The others are defined analogously.

All non-trivial DESAL α applications declare shared variables or bindings, and consequently incur the expense of the Soft State Store. This module introduces fixed overhead, as well as incremental overhead for each additional shared variable or binding. Table Ia and Figure 2a summarize the expense of shared variables. The Telos platform uses Texas

³A live demonstration of this system is available at http://hdlu.com/xmisc/sensors/DP/.

⁴The basestation system uses a port of the Java Bridge library for C#.

⁵The integer literals used in the periodicity and delay specification are 1 and 0, respectively. The values chosen have a negligible impact on ROM size.
Several groups have considered new programming platforms for wireless sensor networks. Here we summarize some of the most significant.

Some have considered *macroprogramming* approaches. The basic idea is to provide programmers with the view that the sensor network is a single entity, rather than a collection of distributed nodes. The *Kairos* language [8], [9], for example, allows developers to program networks using a *shared memory* abstraction. By way of this abstraction, programmers are able to ignore low-level communication details and focus instead on the collective behavior of the network. This is similar to DESAL’s use of shared variables. *EnviroSuite* [10] is another macroprogramming platform that provides *object-based* constructs that model entities in the physical environment.

Another macroprogramming approach is to treat the network as a single data store and to query its contents. *TinyDB* [11] and *TAG* [12] are platforms that enable users to pose *SQL*-like queries against a sensor network. A key challenge addressed by these platforms is the ability to perform efficient in-network data aggregation in response to user queries. *Semantic Streams* [13] provide an additional level of abstraction; sensor network applications are modeled as compositions of *inference units* corresponding to semantic interpretations of sensor data.

The programming systems that share goals most similar to ours are the *Tenet* [14] and *Regiment* [15] platforms. *Tenet* proposes a tiered network architecture that includes low-capability nodes, as well as resource-rich masters. Network behaviors are largely implemented by the programs running on the masters, while the low-capability nodes are delegated smaller units of work, deemed *tasklets*. This basestation-centric view is tailored to heterogeneous devices, a small number of which will be feature-rich, while the remaining majority will be resource-poor. Our view is that while some networks naturally fit the basestation-centric model, there are classes of applications that do not. Consequently, DESAL supports a network-centric view as well as the base-station-centric view.

The *Regiment* macroprogramming system offers a functional reactive interface that enables *spatiotemporal* macroprogramming. The platform enables programmers to address regions of a larger network and provides abstractions for addressing...
time-varying sensor data and application state. A program can, for instance, refer to the 2-neighborhood of a given node and define filters to query attributes across a region. This is similar in spirit, though not in syntax, to DESAL\textsuperscript{α}’s use of multi-bindings.

VII. CONCLUSION

DESAL\textsuperscript{α} has been in use at our institutions for just over one year. The implementation has been used to support protocol experimentation and graduate courses in wireless sensor networks. We have also begun to consider real deployments. These include the soil moisture system already discussed and a large-scale environmental monitoring network that is currently in the design stage. Based on these initial experiences, we have identified three opportunities for improvement.

Composite Bindings. In some implementations, it is useful to correlate end-point values across multi-bindings. A component might, for instance, declare two multi-bindings, \( \text{temp} \) and \( \text{hum} \), used to access the temperature and humidity values of neighboring nodes. Recall, however, that the end-points within \( \text{temp} \) and \( \text{hum} \) are chosen non-deterministically by the DESAL\textsuperscript{α} runtime. The end-point ordering, and indeed, the end-points themselves, are likely to vary between the two multi-bindings. Hence, correlating temperature and humidity values is unnecessarily complex: The task requires nested foreach loops that inspect the \( \text{src}() \) of each end-point. This is both inefficient and developer-unfriendly. The concept of a composite binding addresses this limitation. This type of binding would enable developers to specify multi-bindings that capture multiple variables at each end-point. In effect, the correlation work would be deferred to the DESAL\textsuperscript{α} runtime.

DE Sal\textsuperscript{α} Extensions. The DESAL\textsuperscript{α} development model is not well-suited to low-level programming tasks or tasks involving hard real-time processing constraints. It is difficult, for example, to express standard signal processing algorithms using DESAL\textsuperscript{α}’s guarded action notation. Similarly, the timing guarantees provided by the runtime make DESAL\textsuperscript{α} unsuitable to the development of device drivers and other time-critical components. In our work, when we have encountered the need for a low-level service, we have implemented the service using nesC and introduced corresponding language constructs to access the service. For DESAL\textsuperscript{α} to be useful to a broad audience, this ad hoc approach must be replaced with a structured mechanism. Specifically, developers must be able to introduce new services using low-level primitives (e.g., those provided by nesC or C) and be able to access those services from DESAL\textsuperscript{α} guards and actions. This would allow developers to leverage DESAL\textsuperscript{α}’s high-level programming model without sacrificing functionality or performance.

Multi-Hop Bindings. While DESAL\textsuperscript{α} has been used to implement simple routing protocols, exposing this behavior at the component level may be viewed as inconsistent with the DESAL vision. A guiding objective is to enable developers to focus on high-level application programming without regard to network communication. Hence, while it is possible to implement a spanning tree protocol to route data to a basestation, it would be more natural for the basestation to establish a multi-binding to each cluster-head in its network. The basestation would then iterate through the end-points to access their data. A multi-hop binding implementation would enable this functionality. Or more generally, DESAL\textsuperscript{α} should support a pluggable binding implementation to enable developers to supply binding semantics appropriate to their application.

Addressing these limitations and integrating the remaining features described in our DESAL vision [1] forms the basis for future work.

Acknowledgments. This is work resulting from National Science Foundation award CNS-0520222.

REFERENCES