SNBENCH: A Development and Run-Time Platform for Rapid Deployment of Sensor Network Applications [†]

AZER BESTAVROS best@cs.bu.edu ADAM D. BRADLEY artdodge@cs.bu.edu ASSAF J. KFOURY kfoury@cs.bu.edu MICHAEL J. OCEAN mocean@cs.bu.edu

Computer Science Department
Boston University

***Abstract*— We envision the emergence of general-purpose, well-provisioned sensor networks—which we call "Sensoria"—that are embedded in (or overlayed atop) physical spaces, and whose use is *shared* amongst autonomous users of that space for independent and possibly conflicting missions. Our conception of a Sensorium stands in sharp contrast to the commonly adopted view of an embedded sensor network as a special-purpose infrastructure that serves a well-defined, fixed mission. The usefulness of a Sensorium will not be measured by how highly optimized its various protocols are, or by how efficiently its limited resources are being used, but rather by how *flexible* and *extensible* it is in supporting a wide range of applications. To that end, in this paper, we overview and present a first-generation implementation of SNBENCH: a programming environment and associated run-time system that support the entire life-cycle of programming sensing-oriented applications. The components of SNBENCH are analogous to those commonly found in traditional, stand-alone general-purpose computing environments. SNAFU (SensorNet Applications as Functional Units) is a high-level strongly-typed functional language that supports stateful, temporal, and persistent computation. SNAFU is *compiled* into an intermediate, abstract representation of the processing graph, called a STEP (Sensorium Task Execution Plan). The STEP graph is then *linked* to available Sensorium eXecution Environments (SXEs). A Sensorium Service Dispatcher (SSD) decomposes the STEP graph into a linked execution plan, *loading* STEP sub-graphs to appropriate individual SXEs and binding those loaded sub-graphs together with appropriate network protocols. The SSD may load many such programs onto a Sensorium simultaneously, taking advantage of programs' shared computation and dependencies to make more efficient use of sensing, computation, network, and storage resources.**

I. INTRODUCTION

To date, the emphasis of most research efforts targeting Sensor Networks (SNs) has been on protocol refinements and algorithmic optimizations of how underlying resources in SNs are utilized to achieve a single static task or mission, subject to stringent constraints imposed by the typically impoverished resources of these SNs (*e.g.*, battery lifetime, noisy radio communication, limited memory, *etc*.) In many ways, an inherent assumption of most SN research is that *both* the SN infrastructure as well as the applications deployed on this infrastructure are owned and controlled by a *single* entity. This "closed system" mindset allows the designers of such SNs to think of a SN as a "special-purpose" standalone system.

Motivation and Challenge: With the increased commoditization of sensing, computing, networking, and storage devices, a different model of SN infrastructures is emerging, whereby (1) the owner of the SN and its users may be different entities, and (2) the users of the SN may autonomously deploy independent applications that share the SN infrastructure.

To motivate this alternative "open system" view of a SN, consider a public space such as an airport, shopping mall, museum, parking garage, subway transit system, among many other such examples. Clearly, there are many different constituents who may have an interest in monitoring these public spaces, using a variety of sensing modalities (*e.g.*, video cameras, motion sensors, temperature sensors, smoke detectors, *etc*). In the case of a shopping mall, these constituents may include mall tenants, customers, mall security, local police and fire departments, *etc*. One alternative is for all these different constituents to overlay the public space they share with independent SN infrastructures – comprising separate sensors, actuators, processing and storage server nodes, and networks – each of which catering to the specific mission or application of interest to each respective constituent. Clearly, this is neither efficient nor practical, and for many constituents may not be even feasible. Rather, it is reasonable to expect that such a shared SN infrastructure will be an integral part of the public space in which it is embedded, operated and managed by an entity different from the entities interested in "programming" it for their own use.

Harnessing the power of such shared SN infrastructures will hinge on our ability to streamline the process whereby relatively unsophisticated "third parties" are able to rapidly develop and deploy their applications without having to understand or worry about the underlying, possibly complex "plumbing" in the SN infrastructure supporting their applications.

Today, programming SNs suffers from the same lack of organizing principles as did programming of stand-alone computers some fourty years ago. Primeval programming languages were expressive but unwieldy; software engineering technology improved with the development of new high-level programming languages that support more expressive and useful abstraction mechanisms for controlling computational processes, as well as through the wide adoption of common software development platforms that leverage these abstractions. For SNs, we believe that the same evolutionary path need not be (painfully) retraced, if proper abstractions, expressive languages, and software engineering platforms are developed in tandem with advances in lower-level SN technologies.

The SN workBench: In this paper, we overview and present such a platform, which we call SNBENCH (SN workBench). SNBENCH embodies a programming environment and associated distributed run-time system that support the entire life-

[†] This research was supported in part by NSF awards ITR-0205294, EIA-0202067, and ANI-9986397.

cycle of programming SN applications. Our primary focus is *not* on optimizing the various algorithms or protocols supporting an *a-priori* specified application or mission, but rather on providing *flexibility* and *extensibility* for the rapid development and deployement a of wide range of applications.

While snBench is conceived (by design) to be oblivious to the sensing modalities in a SN, in our current implementation we are naturally focusing on the particular SN infrastructure in our laboratories, which we call the "Sensorium", consisting of a web of wired and wireless networked video cameras and motes [1] spanning several rooms, processing units and a terabyte database, all of which are managed together to be able to execute ad-hoc programs (or queries) specified over the Sensorium's monitored spaces. In many ways, we view our Sensorium as prototypical of emerging SN infrastructures whose use is *shared* amongst autonomous users with independent and possibly conflicting missions.

Programmatic access to the Sensorium is provided via a high-level task-centric programming language that is compiled, distributed, scheduled, and monitored by snBench. A snBench program is specified in the SNAFU (SN Applications as Functional Units) language. SNAFU is a strongly-typed functional-style programming language which serves as an accesible, high-level language for developers to glue together the functionalities of sensors, actuators, processing, storage, and networking units to create stateful, temporal, and persistent programs. In effect, snBench presents programmers with an abstract, "single-system" view of a SN, in the sense that SNAFU code is written for the Sensorium as a whole, and not separately for each of its various subsystems.

A SNAFU program is compiled into a Sensorium Task Execution Plan (STEP), which takes the form of a directed acyclic graph (DAG), in which the nodes are the sampling and computation operations required to execute the program. An execution plan may consist of nodes that are either bound (*i.e.* must be deployed on a specific resource) or unbound (*i.e.* free to be placed wherever sufficient resources may be found). A STEP is analogous to a program that has not been linked, and has a straightforward serialized XML representation.

The Sensorium Service Dispatcher (SSD) is responsible for the linking and scheduling of a STEP onto the SN infrastructure. In general, the SSD solves a restricted form of a graph embedding problem, finding resources capable of supporting sub-graphs of the STEP graph and allocating them as appropriate. The SSD optimizes the use of resources and identifies common subexpressions across already deployed execution plans such that computation resources may be shared and/or reused. The SSD relies heavily on the Sensorium Resource Manager (SRM), a registrar of computing and sensing resources available in the system at present. The SSD decomposes a single "unbound" STEP graph into several smaller "bound" STEP graphs and dispatches those graphs onto available Sensorium functional units.

Each Sensorium functional unit features a Sensorium eXecution Environment (SXE), which is a run-time system that realizes the abstract functionalities presented to SNAFU programmers as basic building blocks. Such realizations may rely on native code (*e.g.*, device drivers, or local libraries) or may entail the retrieval of programmer-supplied code from remote repositories. An SXE interprets and executes partial STEP graphs that have been delegated to it by the SSD. Such a partial STEP graph may involve computing, sensing, storage, or communication with other SXEs.

Paper Overview: In this paper, we discuss our first-generation implementation of snBench, highlighting the salient features of its SNAFU programming paradigm and its SXE and SSD run-time support. Section II offers a general overview of SNAFU. Section III briefly illustrates the composition of a STEP graph. Section IV describes how the SSD schedules and deploys STEPs onto Sensorium resources, and Section V describes the run-time systems available on those resources. Finally, Sections VI and VII conclude the paper with a brief exposition of related work as well as research work enabled by snBench.

II. The SNAFU Programming Language

SNAFU is a strongly-typed programming language intended primarily for "gluing together" the sensors, computational resources, and persistent state that make up a Sensorium application. SNAFU programs are written in a simple functional style. As a functional language, SNAFU functions do not allow for side effects, however a subset of the standard library functions are stateful (*i.e.*, they do have side effects). All SNAFU program terms are expressions (*i.e.*, their evaluation produces values) as exemplified by the simple SNAFU program below which inspects a single video frame, and returns either the name of a person recognized in the frame, or NIL.

```
identify(facefind(snapshot(cam1)))
```

We impose specific restrictions on the composition of SNAFU programs to ensure that a program can be represented as an acyclic graph. Explicit recursion is forbidden in SNAFU and is detected by the compiler/type engine. A user is not allowed to assign functions to variables, and we statically check for recursion (including transitive cases) by annotating each function with the transitive closure of all functions called by it; if a function calls such a function with its own name in the annotation list, the type engine throws a type error.

Triggers and Repetition: In SNAFU, iterative executions are carried out through the trigger construct, within which limited forms of recursion are possible (as described towards the end of this section). A trigger is a "cycle-safe" way to repeat and predicate the evaluation of expressions in time. SNAFU supports three general trigger types:

The "do-once" trigger construct `trigger(x,y)` will continually evaluate `x` until it becomes true, at which point it will evaluate `y`, returning `y` as the value of `trigger(x,y)`. The following SNAFU program will shutdown "MyComputer" when its CPU temperature exceeds 65 degrees centigrade.

```
trigger(
  greaterthan(cputemp(MyComputer),temp("65C")),
  shutdown(MyComputer))
```

The "level-trigger" construct `level_trigger(x,y)` will continually evaluate `x` and every time `x` evaluates to true, `y` is re-evaluated. Note that this expression will continually re-evaluate in perpetuity. Intuitively, once such a term has been evaluated, its evaluation tree can be "reset" and the evaluation begun again, causing the value of the expression to change regularly based upon new input values. The SNAFU example below would run indefinitely, constantly sending a notification to the owner of "MyComputer" as long as the predicate evaluates to true.

```
level-trigger(
  greaterthan(cputemp(MyComputer),temp("60C")),
  notify(owner(MyComputer),
    "Your CPU is very hot."))
```

The "edge-trigger" construct `edge_trigger(x,y)` will continually evaluate `x` and when `x` *first* becomes true (or if it initially evaluates to true), `y` is evaluated once. When `x` becomes false and subsequently becomes true again, `y` will be re-evaluated, and so on. The SNAFU example below would run indefinitely, sending a notification to the owner of "MyComputer" only when the predicate evaluates to "true" after having evaluated to "false" in the prior evaluation.

```
edge-trigger(
  greaterthan(cputemp(MyComputer),temp("60C")),
  notify(owner(MyComputer),
    "Your CPU has become very hot."))
```

Both the level and edge triggers could be viewed as forms of persistent queries which can be queried by other expressions using trigger value retrieval functions (discussed in the next section). In general, a persistent query will live until it goes un-used for some configuration-specific period of time (*e.g.*, 60 minutes). This can be overridden using a flow type which states a particular persistence policy, duration, or predication.

Trigger Value Retrieval: The transient values of edge and level triggers can be accessed using reads that are either blocking, non-blocking, or fresh.

A non-blocking read (denoted by `nonblock` and invoked upon a single trigger argument) is only willing to wait if the expression has never completed an evaluation; in all other cases, it immediately returns the result of the last completed evaluation of the trigger's target expression. A blocking read (denoted by `block` and invoked upon a single trigger argument) waits until the next (or currently ongoing) evaluation of the target expression completes, then produces that value. A fresh read (denoted by `fresh` and invoked upon a single trigger argument) waits for a complete re-evaluation of the trigger's predicate and target and produces that value.

An edge or level trigger cannot be used in an expression without wrapping it in one of these primitives; failure to do so produces a type error. A one-time trigger does not need to be accessed using these primitives, as it has an implicit blocking semantic.

A trigger expression may wish to make use of its own prior evaluations (*e.g.*, to maintain a list of the last 10 results). This is supported through the use of the `LAST_TRIGGER_EVAL` token, which (syntactically) acts as a variable and (semantically) acts as a non-blocking read of the trigger which is its closest enclosing parent, returning NIL if the trigger has not yet evaluated.[1]

[1]A variation on this, which can refer to outermore containing triggers is under consideration, but we expect that the same behavior can easily be achieved using the simple form described here in combination with letonce binding.

Let-Bindings: Most useful programming languages include some notion of binding a single recurring symbol to either some persistent value (state) or some repeated instantiation of a more verbose program term (macro). SNAFU supports this through "let-bindings." For example, let-bindings allow a programmer to take a single sample from a sensor or a persistent expression and use that single value in several places within an expression.

SNAFU offers four kinds of let-binding – functions, let-each, let-once, and let-const – to support different useful semantics. The particular (and peculiar to many functional languages) semantics of triggers demand that we define three different forms of variable let-binding.[2]

In some programs, it is desirable to compute the value of an expression once and to re-use that result for the life of the program. The `letconst` binding offers precisely that behavior: the first time the execution envrionment encounters a letconst-bound variable, it will evaluate the bound-in expression; all occurrences of the variable thereafter evaluate as having that same value. Consider the assignment that allows all further instances of the expression `cam1` to resolve to the single, eager evaluation of camera("Grad Lab South") which will be only evaluated once.

```
letconst cam1 = camera("Grad Lab South") in ...
```

In other cases, programmers may wish to use symbols as macros: to represent commonly-occurring subexpressions, each instance of which is to be independently evaluated. A `leteach` binding of the form "`leteach x = y in z`", replaces each occurrence of `x` in `z` with an independent copy of `y`. Notice that it is possible that `y` will have a different value in each such occurrence.

```
leteach prefContactMethod =
  get_contact_mode(user("Michael J. Ocean"))
  in ...
```

Sometimes it is useful to have a common subexpression evaluated only once within an iteration of a trigger's control loop. For this purpose, we use letonce-bindings, of the form "`letonce x = y in z`". This allows the expression `y` to be evaluated once per iteration of the containing trigger as in the following program fragment, the intent of which is for the trigger to repeatedly take samples from the camera, and whenever a sample is found which contains a face (the predicate), we would want to pass that same frame to the facial recognizer.

```
letonce x = snapshot(cam5) in
  level_trigger(
    is_face_visible(x),
    identify_face(x))
```

In general, this means that the children of a persistent query are evaluated using an on-demand strategy; all children which are needed to compute the predicate are enabled and evaluate, and once a true predicate is detected, the remaining children (which were not activated in that predicate computa-

[2]Notice that we do not support higher-order terms, in the sense of assigning a function to a variable.

tion) are also activated to compute the trigger's value.[3]

The behavior of a SNAFU program is undefined if a letonce-bound variable is separated from the letonce statement by more than a single persistent trigger (nested triggers) or if the letonce-bound variable appears within several disjoint persistent triggers (parallel triggers), as either of these cases allows the letonce-bound expression to be re-evaluated under the control of several simultaneously-executing triggers.

Type-checking and Flow-types: SNAFU is a strongly-typed language; SNAFU programs are statically type checked to identify errors. SNAFU disallows type casts and no explicit data types are used in the program syntax. Instead, permissible type promotions and coercions are handled automatically in the type inference phase of compilation. We use standard rules from the lambda-Calculus with subtyping [2] to type-check SNAFU programs. The compiler also maintains a map from type pairs (τ, τ') to "shim function" names; whenever a promotion from τ to τ' takes place, if the function name so identified is non-null then the promoted expression is passed as an argument to the named shim function, which is then treated as the promoted expression.

SNAFU allows program terms to be annotated with "flow types". Flow types provide programmers with a mechanism via which they may constrain the manner in which their programs are deployed and/or executed in the Sensorium. As examples, the programmer may be interested in ensuring some particular Quality of Service (QoS), say a minimal rate of program evaluation, or may be interested in protecting the results of some portion of the computation/commnunication from using/traversing untrusted computers/networks, or may be interested in ensuring that certain compute nodes are used (even if others are available). Obvious examples of flow types include labeling a term's value as "private" (such that it may only transit "trusted" network links unencrypted), labeling a trigger as persisting in the network for no less than some minimum lifespan, or labeling a trigger as evaluating its predicate at no less than some (real-time) rate.[4]

```
public(facecount(private(snapshot(cam1))))
```

```
period(100ms,level_trigger(f(cam1),g(cam1)))
```

SNAFU Program Compilation, Linking, Loading, and Execution: Owing to its functional style, SNAFU expressions are easily mapped to a tree-like directed acyclic graph (DAG) with a single root. The tree evaluates by having values percolate up from the leaves through intermediary function nodes toward the root. The value of any node in the tree depends upon its children (leaf nodes), and a node is invoked once all of its children have produced values.

[3]Systems programmers will readily recognize the relationship between this property and "race conditions"; intuitively, if some action and its predicate are both dependent upon a single value, it is usually undesirable for that value to change between the evaluation of the predicate and the action.

[4]An exhaustive account of the nature (and semantic) of these flow types is beyond the scope of this paper, and is indeed premature, in the sense that much of these "types" will be closely matched to constraints that are possible to enforce or guarantee through the SSD's "linking" and "loading" of compiled SNAFU programs. For example, supporting QoS flow types is predicated on the ability of the SSD to reject a program whose flow types are "incompatible" with the current state (capacity, available guarantee-able resources, *etc*) of the resources under its control.

The SNAFU compiler transforms the Abstract Syntax Tree (AST) of a SNAFU program into an interpretable/executable target format, which we call a "Sensorium Task Execution Plan" (STEP). A STEP is a DAG whose nodes specify values, computations, sensing operations, or communication tasks, and whose edges specify evaluation dependencies. Simple programs will usually be transformed directly into such STEP evaluation trees; the single-evaluation let construct (see "Let-Bindings" above) can be used by a developer to explicitly link a single subtree onto several parents.

Generally, SNAFU compilation results in the creation of an "unbound" STEP program – a STEP graph containing one or more "unbound" nodes. An "unbound" STEP program cannot be run until its constituent nodes are linked (bound) to Sensorium resources. An "unbound" STEP program is posted to the SSD, the entity responsible for linking and dispatching STEP programs. Given the state of the available system resources and the resources required by the nodes comprising this graph, the SSD linking process attempts to decompose the "unbound" STEP graph into one or more "bound" sub-graphs (a "bound" graph is comprised entirely of "bound" nodes). These generated sub-graphs are "bound" STEP programs which are deployed by the SSD to available devices in the Sensorium. Such devices are able to execute STEP programs using a run-time interpreter.

Alternatively, in some cases, it may make more sense to compile SNAFU ASTs directly into a more suitable target language (*e.g.* C99, Intel asm, *etc*) or to translate them into STEPs which are then translated by an intermediary for the target.[5]

The dissection and dispatch of STEP graphs are explained in IV, while the discussion of how STEP graphs are interpreted and executed is provided in Section V.

III. Sensorium Task Execution Plan (STEP)

A STEP graph could be construed as the "assembly language" of snBench.[6] As we hinted earlier, a STEP graph is comprised of STEP nodes, which must be "bound" by the SSD before execution is possible. We note that there is not a one-to-one mapping between the "unbound" STEP graph (resulting from the compilation of a SNAFU program) and a running "bound" STEP DAG. For example, a running STEP DAG may be a sub-graph of a single STEP DAG, or it may be the confluence of several STEP DAGs. To this end, we support the encoding of general DAGs, through two techniques. First, a STEP program file is an XML object with a non-program root element, alowing us to encode several independent STEP DAG roots as its "children" (for serialization purposes only). Second, the DAG is encoded as a set of trees embedded in the DAG which span all of its nodes (reaching each exactly once). STEP nodes that require communication with remotely deployed STEPS utilize special "socket" child nodes,

[5]In our current implementation, our primary consideration is on maintainability and simplicity of the components, with secondary consideration given to computational costs.

[6]The STEP "assembly language" is admittedly somewhat high-level and although STEP programs are human-readable (they are serialized as XML documents), it is a cumbersome language for direct program composition.

while STEPS that may be merged together for computational reuse utilize "splice" nodes to correspond edges to reusable components.

In the remainder of this section, we use the example STEP program shown in Figure 1 as a backdrop against which we briefly discus the various types of meaningful STEP nodes that comprise a STEP graph. This STEP program is the result of compiling the following SNAFU snippet, which returns the maximum number of faces detected/observed from any one of two cameras mounted on `s05(.sensorium.bu.edu)`.

```
max(facecount(snapshot(sensor("s05","cam1"))),
    facecount(snapshot(sensor("s05","cam2"))))
```

`exp` nodes represent calls to primitive functions analogous to the fundamental operations supported by the SXEs (*e.g.*, addition, string concatenation, image manipulation, *etc*). Each `exp` node identifies a function/operation via its `opcode` attribute. Actuator nodes are a sub-class of `exp` nodes that specify "push-style" interactions with physical devices. Expressions involving the use of actuators are used to accomplish tasks such as switching lights on and off, controlling HVAC components, turning a camera on or off, and so on. The children of `exp` nodes are the parameters/operands (if any) of the operation.

`sensor` nodes are abstractions of physical hardware input devices. When present, these nodes will always be leaves in a STEP graph. Operationally, a `sensor` node stands as a reference to a specific piece of sensor hardware in the SN. For example, an `exp` node with opcode "`snapshot`" may have a child node that is a `sensor` node specifying the particular camera the frame should be captured from). A sensor node has a `type` attribute indicating the general type of input device that this node abstracts. Types currently supported in SNBENCH include `image`, `video`, `audio`, and `temperature`. The value of a `sensor` STEP node is an HTTP URI to the sensor device, generally via some SXE host (*e.g.*, http://s05.sensorium.bu.edu:8080/snbench/sxe/sensor/image/2).

`value` nodes pass a value (string, integer, image, *etc*.) from one point to another. These nodes are generally used as parameters feeding `exp` nodes.

`trigger`, `edge_trigger`, and `level_trigger` nodes specify persistent query evaluations that parallel the corresponding *trigger*, *edge_trigger*, and *level_trigger* SNAFU constructs. Trigger nodes must have at least two children: the predicate and the body, and may have zero or more `flowtype` nodes. The `opcode` attribute determines whether the trigger will be read via blocking, non-blocking, or fresh reads.

`read` nodes indicate access to the values produced by `trigger` nodes. They have at least one child: a single `trigger` node.

`socket` nodes signify communication between SXEs over a loosely-coupled network. These nodes are almost never present in the STEP files produced by compilers; they are injected into STEP DAGs by the SSD to allow distribution of a program's evaluation across SXEs.

A socket node has a `role` attribute which is set to either `sender` or `receiver`. Recall that values perculate up from the leaves to the root of a STEP graph. As such, a socket node is a *sender* if it has no parents and a *receiver* if it has no children. A `socket` cannot be both a sender and a receiver, because the semantics of its attributes are dependent upon this distinction. Such a node can be simulated with a two-node graph connecting one receiver with one sender; we call this construct a "rendezvous node".

The `protocol` attribute determines which protocol model this socket uses to communicate. Currently supported models are GET (HTTP/1.1 pull) and POST (HTTP/1.1 push), we also anticipate adding UDP, serial port, and SN radio communication protocols. The `peeruri` attribute is used to bind the communications to a particular (logical or physical) address.

`const` nodes are used to prevent re-evaluation of their children (*e.g.*, a letonce binding). Each has exactly one child, namely the subgraph whose evaluation we wish to block.

`flowtype` nodes encode run-time security, performance, and persistance contraints. If they have any children, they will be defined and validated using an independent XML namespace. These nodes appear as children of the nodes that they constrain.

`splice` nodes are used to describe how one partial STEP graph should be spliced/grafted onto another. Splice nodes allow independently compiled STEP graphs to share common components. Specifically, after a STEP graph has been deployed to a given SXE, another program may be admitted by the SSD to that SXE so as it would share part of the already-deployed STEP graph. Put differently, a newly admitted STEP graph may have a subgraph in common with an existing, running STEP graph and we may wish to re-use those computations as appropriate.[7]

The `splice` node has a `target` attribute whose value must be the id of another existing node. Structurally, a splice resembles a `socket` insofar as it has either no children or no parents. Splice nodes are only represented in "bound" STEP files posted by the SSD to an SXE that already has active STEP graphs. The splice nodes exist transiently, while an SXE's active STEP graph is being augmented to graft on the newly added STEP graph at the splice node specified.

IV. THE SENSORIUM SERVICE DISPATCHER (SSD)

The SSD is responsible for scheduling the execution of a STEP program on Sensorium resources, based on the availability/state of these resources as maintained and reported by the SRM. The SSD's role is analogous to both the register allocation and dynamic linking phases of a conventional compiler. In this section, by "scheduling" we mean assigning a graph of STEP nodes to one or more appropriate SXEs for execution.

At present, the SSD and SRM are implemented in the Java programming language. Java provides a flexible and rapid development platform for our first generation implementation. Although a port to another language would certainly be

[7]The STEP graph posted to an SXE containing `splice` nodes are created by the SSD through its knowledge of the "entire state" of the SN infrastructure.

```
<step id="202219@s00.sensorium.bu.edu">
   <exp opcode="max" id="abcd" bindto="http://s05.sensorium.bu.edu:8080">
      <flowtype name="persist" value="Mon Jul 25 23:59:59 EDT 2005" />
      <exp opcode="facecount" id="bcde" bindto="http://s05.sensorium.bu.edu:8080">
         <exp opcode="snapshot" id="cdef"><value id="defg">
            <sensor type="snbench/image">
               http://s05.sensorium.bu.edu:8080/snbench/sxe/sensor/image/1
            </sensor></value></exp></exp>
      <exp opcode="facecount" id="efgh" bindto="http://s05.sensorium.bu.edu:8080">
         <exp opcode="snapshot" id="fghi"><value id="ghij">
            <sensor type="snbench/image">
               http://s05.sensorium.bu.edu:8080/snbench/sxe/sensor/image/2
            </sensor></value></exp></exp>
   </exp>
</step>
```

Fig. 1. STEP program for computing the maximum number of faces detected/observed from any one of two cameras mounted on s05(.sensorium.bu.edu).

feasible, at present we see little advantage to leaving the Java platform. For signaling and communication between the SSD/SRM and SXEs we use the HTTP/1.1 protocol. In using HTTP, we have the advantage of using ubiquitous and mature clients for testing (*i.e.*, off-the-shelf web browsers). More importantly, HTTP URIs provide a natural hierarchical namespace for describing SN resources, and there is a natural relationship between the HTTP protocol header methods (*e.g.*, `POST, PUT, DELETE, GET,` *etc*), response codes (*e.g.*, `OK, CREATED, ACCEPTED, MOVED,` *etc*) and the general signaling communication we must support.[8]

A. Operational Overview

The SSD maintains a master STEP DAG that reprsents the composition of all SXE-deployed STEP DAGs under its control. The SSD does not execute this graph or store values for any nodes; the graph is maintained for resource allocation and scheduling purposes only. All STEP nodes within this master graph indicate onto which physical SXE the STEP node has been deployed. Conceptually, the nodes of the graph are colored with each node's color representing the SXE on which it resides. Nodes are also labeled with load and performance information from the Sensorium Resource Manager (SRM).

Each SSD is tightly coupled with an SRM, which tracks the physical resources available within its managed Sensorium (*e.g.*, SXEs, Sensors, controlled SSDs *etc*). The individual SXEs communicate with the SRM to indicate any Sensorium state changes, and likewise an absence in communication from a managed SXE represents a change in state.

B. SSD-Driven Events

A master STEP graph could be altered at an SSD by a number of events, which we overview below.

Program Insertion: When a new STEP program DAG is posted to the SSD, it must be inserted into the master STEP DAG; all new "unbound" (uncolored) nodes within the program must eventually be "bound" (colored) and transmitted to particular available SXEs. The trivial approach is to insert a new STEP program DAG as a disconnected subgraph, though more interesting approaches try to identify re-usable common subgraphs and avoid re-instantiating such common subgraphs. That is, the SSD should identify a common subgraph between two STEP DAGs, which can be treated as a single subgraph in the master DAG.

In its general form, this problem is an NP-hard graph embedding problem. Fortunately, our problem is further constrained such that, at least in practice, there is probably a much tighter bound. Our master STEP graph is directional and acyclic, our nodes have labels (where the labels, *i.e.*, node classes and opcodes, must match), we only consider "leaf" subgraphs (*i.e.*, no node in the common subgraph has a child which is not in the subgraph), and finally we only look for subgraph matches within persistent (trigger) expressions.

Our initial implementation of the SSD only tries to match identical "trigger-rooted" subgraphs of a new STEP program to subgraphs of the master STEP graph; the reuse of a previously deployed non-persistent sub-expression would return data that is stale to the newly added program. To improve searching performance, we maintain a hash table of all active trigger nodes in the master STEP graph to avoid searching the master STEP graph on every insertion. Our graph reuse algorithm compares all trigger rooted subgraphs of the newly added STEP with the entries of the hash table; If the two graphs are "identical" then reuse is possible.

In practice there may be structural differences between two graphs that are functionally equivalent (*e.g.* socket communication nodes may be inserted to distribute the computation across two SXEs). Our matching algorithm "flattens" STEP graphs into functional representations by considering the node classes and opcodes and attempts to ignore such differences.

It may be beneficial to widen the graph reuse to recently deployed subgraphs within some time threshhold. The more general case of the code-resuse problem is made all the more interesting because we may be splicing around and onto partially-evaluated terms, and our own resetting of child nodes may introduce scheduling (and, in extreme cases, starvation) issues. If graph reuse is possible, the master STEP graph will be updated to contain a graft of the new nodes onto the existing master STEP graph.

The SSD transforms the newly posted STEP graph into a splice STEP graph in which the new matching subgraph is replaced at its root with a single `splice` node with the same ID as the already deployed match. The `splice` node is a directive to an SXE that asserts the dependency between

[8]For example, a new STEP program DAG is uploaded to the SSD by an HTTP post of an XML object to the URI "http://host:port/snbench/ssd" the response to which will be either "201 Created" or one of several 400-level error codes.

the new nodes and the existing node at the specified point in the graph. Section V-A has details about splicing from the perspective of the SXE.

The remaining new nodes must be colored (assigned to an SXE) and, in the case that no code reuse is possible, the new STEP program is added as a disconnected subgraph.

In many cases a new STEP graph may need to be split across SXEs for any number of reasons (*e.g.*, insufficient resource availability on a single SXE). Even in cases of graph reuse, the SXE running the reusable graph component may not have resources available for the new STEP nodes (computations). In such cases, the STEP graph will need to be split into smaller subgraphs, and those subgraphs deployed onto SXEs with available resources.

The STEP node coloring algorithm attempts to assign each unbound STEP node to an SXE with sufficient resources. We group all unbound nodes and attempt to find a single SXE with sufficient resources to host all of these nodes in the STEP graph. If no SXE has enough resources available, the graph is split into two or more connected subgraphs and we recurse on each such subgraph.[9]

The SSD will insert `socket` nodes to join subgraphs that span multiple SXEs across the same logical STEP program graph. The algorithm to perform this is a straightforward graph traversal; when a node's color differs from that of its child, `socket` nodes are inserted.

(Sub-)Program Deletion: If a program is deleted, reference counts to the effected nodes must be reduced and nodes that are no longer referenced must be removed. Hosting SXEs are instructed to take action if nodes are to be deleted. Section V-D has details about deletion from the perspective of the SXE.

C. SXE/SRM-driven Events

The SRM is responsible for direct communication with the SXEs to determine the run-time state and general health of Sensorium resources. As such, any SXE state changes observed by the SRM will be forwarded to the SSD. We describe these events next.

Evaluation Completion: When a single evaluation has completed its execution or when a persistent expression has passed its lifespan without being renewed, its constituent nodes (if not referenced by any other programs) should be deleted from the master STEP DAG. Evaluation completion is reported to the SRM via the SXE, and the SXEs require no additional instruction from the SSD when natural evaluation completion occurs.

SXE Graceful Shutdown: If an SXE notifies the SRM of a shutdown, the SSD must re-color all of that SXE's nodes (and therein transmit them, along with the requisite "wiring updates", to other SXEs). If an evaluation cannot continue without the shutting-down SXE (*e.g.*, a node depends upon an operation particular to that SXE), the evaluation is terminated with an error condition.

[9]The graph splitting algorithm walks the graph from the head looking for a node with multiple children. When such a node is reached all nodes on the path until this node (and including this node) are the first subgraph, and each child of this node is a separate subgraph. If the end of the graph is reached then the graph is actually a list and any split will achive two connected subgraphs.

SXE Failure Detected: The SXE periodically sends a heartbeat to the SRM indicating health, load, and other useful resource information. Should an SXE miss three successive heartbeats, or indicate that it is otherwise in trouble (*e.g.*, missing deadlines required by the flowtype of a deployed STEP node), the SSD will remove and relocate any affected STEP graph nodes.

Should an SXE fail to report its state as expected, the SRM assumes that the SXE has failed or has otherwise become disconnected and notifies the SSD accordingly. The SSD must reset all STEP nodes present on that SXE, migrate these nodes to another SXE, and the evaluation must also be re-started (since partial evaluation state may have been lost and sub-evaluations may therefore need to re-transmit).

SXE Reset Detected: If an SXE should fail, yet come back up before its failure is detected, the SRM will receive a heartbeat message from the SXE indicating that the SXE has no active program nodes without sending an evaluation completion message. In this case the SSD must reset all STEP nodes present on that SXE, and force all child STEP nodes residing on other SXEs to re-transmit their last evaluation results (if any).

D. Intra-SSD Events

In practice we anticipate that many SSDs may be deployed simultaneously over a Sensorium. In this model, each SSD would still administer its own "local" SN, yet the boundaries of a local SN may shift for any number of reasons, and SSDs may need to exchange events to collectively manage the Sensorium. Our current implementation lacks complete support for the management of multiple SSDs, however much of the practical infrastructure to support this change is already in place (*e.g.*, SSDs can pass "live" SXEs between each other). It remains to be seen whether the management of multiple SSDs is best achieved via a hierarchy featuring a "root" SSD (*a la* DNS) or through a distributed (peer-to-peer) administration model.

V. Sensorium Execution Environments (SXEs)

An SXE interprets and exectutes a partial STEP delegated to it by the SSD. Through the STEP program abstraction, an SXE provides a generic interface to SN resources, including *Sensor Elements* (SEs) and *Computing Elements* (CEs) capable of processing and storage of sensory data, as well as control of actuators.

An SXE is responsible for enabling the concrete tasks of a program's execution, including communication with other participant SXEs (usually via `HTTP GET`, `POST`, and `PUT` operations). Our initial implementation of SXEs uses Java technologies – namely, Java 1.5 for the runtime, Java Media Framework for sensor interaction, and Java based NanoHTTPD for HTTP communications. In addition to the programming benefits of the strongly-typed Java language and the protection benefits of its sandboxed runtime environment, Java provides us with straightforward mechanisms for runtime loading of functionality over the network (via JARs or dynamically compiled source code). The GCJ suite also allows us to compile Java programs into native bytecode when runtime performance is an issue.

Each SXE instance has a UUID, which persists across executions (shutdowns, crashes, *etc*) and is registered with the SSD via the SRM. The SXE's primary external interface is via an HTTP Server under the URI root "http://host:port/snbench/sxe/". The SXE also acts as an HTTP client to interact with the SSD, SRM, and other SXEs, although the SXE also implements other communication protocols to interact with specialized SXEs or non-SXE elements (*e.g.*, motes). Each SXE periodically communicates with the SRM to which it is assigned and reports its sensing and computation capabilities at that instant in time. The SSD uses this information in deciding which (partial) STEP programs to dispatch to a given SXE. Partial STEP graphs are uploaded to an SXE via an HTTP POST of an XML object. The SXE executes a program specified as a STEP graph by continually iterating over all execution-enabled (ready) nodes. Notice that this approach to evaluating a STEP graph is not unique.[10]

The essential operations of the SXE can be divided into four classes of algorithms: STEP program admission (splicing), STEP program interpretation, STEP `exp` node evaluation, and STEP program deletion (pruning).

A. STEP Admission (Graph Splicing)

When a new STEP program graph is posted to an SXE, the new graph may need non-trivial integration (*i.e.*, splicing) into the existing SXE STEP DAG. This problem is made all the more interesting because we may be splicing around and onto partially-evaluated terms, and our own resetting of children nodes may introduce scheduling (and, in extreme cases, starvation) issues.[11] We refer to the two splicing operands as the "base" and the "graft". Every node in each graph has an, globally unique ID. The base is a standard STEP DAG. The graft interacts with the base using the following rules:

1. Node Connection: The graft includes `splice` nodes, each of which is a placeholder to be filled by the node in the base graph whose ID corresponds with the splice node's targetID attribute.

2. Node Replacement: The graft includes normal STEP nodes with the same IDs as nodes in the base graph; we call the graft node the *doppelganger* and the base node the *original*, where the intention is for the doppelganger to replace the original. If the doppelganger has no children, the original's children are re-parented to it; if the doppelganger has children, the original's children are orphaned. The doppleganger keeps its parents from the graft graph as well as inheriting those of the original node in the base graph (although those may also be or have been replaced by other doppelgangers).

B. STEP Interpretation

Each SXE maintains an active STEP graph, consisting of all partial STEP graphs it has received from its SSD. These partial STEP graphs consist of a variety of STEP nodes (described in above in Section III). Recall that a STEP graph is a DAG in which values propegate up toward a single root; Tasks appearing in nodes higher in the STEP graph are not be able to be executed until their children have been evaluated. The STEP interpretation algorithm iterates over all nodes of the STEP looking for actionable nodes (*i.e.*, nodes whose children have evaluated and are currently non-stale values) and executes these actionable nodes accordingly. Trigger evaluation largely involves setting and resetting the state of nodes as stale or ready after evaluation as appropriate.

C. STEP `exp` Node Evaluation

STEP `exp` nodes are analogous to the opcodes of the STEP programming language. The SXE contains a library of basic "opcodes" and their implementations in the Java programming language, known as `sxe.core`. For example, there is a class `/sxe/core/math/add.java` corresponding to the opcode "`sxe.core.math.add`" as there is for each opcode known to the SXE. We implement a custom Java ClassLoader to support dynamic loading of new opcodes from trusted remote sources.

Internally, all opcode methods manipulate snObjects. The snObject is a first-class Java representation of a STEP `value` node within a STEP graph. The snObject itself is a helper classs that provides common methods that allow objects to be serialized for transmission between SXEs (as XML) and for viewing results via a standard web browser (using standard mime-type appropriate content). Similarly, snObjects implement a method to parse an object from its XML representation. Specific snObjects exist including snInteger, snString, snImage, snBoolean, snCommand, *etc*.

All STEP `value` nodes that are children of an `exp` node will be passed to the appropriate opcode implementation as the correct corresponding snObject. Likewise, The opcode implementation is responsible for returning an snObject such that its result may be passed further up the STEP graph. Within the body of the method, any computation may take place and this computation is not limited to Java calls (*e.g.* communication with remote hosts, execution of C++ code via the Java Native Interface, or generation and transmission of machine code to a remote host). Two simple examples – the if-then-else conditional and string concatenation – are given below for illustrative purposes.

```
snObject Call(snObjectArgList argv)
  throws EvaluationFailure
{
  snBoolean cond = argv.popBoolean();
  if(cond.equals(snBoolean.TRUE))
    return argv.popObject();
  else{
    /* throw away the first */
    argv.popObject();
    /* return the second */
    return argv.popObject();
  }
}

snObject Call(snObjectArgList argv)
  throws EvaluationFailure
{
  StringBuffer b = new StringBuffer();
  snString s = null;
  while(argv.hasNext()){
    s = argv.popString();
    b.append(s.getString());
  }
  return new snString(b.toString());
}
```

[10]Indeed, the selection of which nodes to consider next amounts to a scheduling decision which may be constrainted by QoS requirements, or other considerations (*e.g.*, frame rates, *etc*).

[11]For this reason, it may also be interesting to consider admission-control algorithms for determining when a new partial STEP DAG is eligible for splicing onto an existing STEP DAG.

D. STEP Program Deletion (Graph Pruning)

Pruning a STEP graph is required when the SSD requests that an expression be removed from the SXE's STEP. We support removal at the granularity of nodes, where the removal of a node terminates evaluation of any expressions which are parents of that node. The SSD is responsible for reporting the aborted status; the SXE does not immediately delete the nodes from its URI namespace, rather they are marked to be collected as garbage at some time in the future.

As such, pruning is a two-phase operation, consisting of garbage marking and physical deletion. The garbage marking algorithm is a straightforward postfix DAG ascent, while the cleanup algorithm simply iterates over all nodes, removing those which have expired.

VI. Related Work

A. Similar Initiatives

We restrict our coverage of related work to efforts focusing on the development of programming paradigms for the composition of services for a general purpose sensor network, as opposed to efforts focusing on application development frameworks for a particular class of SNs, or for a special-purpose architecture (*e.g.*, motes) [3], [4].

TAG [5] and Cougar [6] are examples of works that allow the composition of a query with an abstract representation of the underlying phyical sensor network. Unfortunately, these solutions are limited to query style programs, whereby the SN is viewed as a distributed data acquisition/storage infrastructure (following the conception of the "SN as a database" [7]), and thus lacking extensibility and arbitrary programmability.

MagnetOS [8] provides greater flexibility with respect to the programs that can be deployed and has a "single system" programming model. In fact, MagnetOS provides runtime components to pool SN resources to form a single Java virtual machine (JVM). While this approach supports extensible dynamic programming, it lacks in its inability to share SN system resources across autonomous applications. One may also argue that a JVM is not the best abstraction for a SN. Impala [9] provides modularity and adaptivity though a variety of runtime agents, though their solution is programmed at the granularity of the individual sensor, and it too lacks support for multitasking an entire SN.

The work in [10] shares our vision of a shared, heterogeneous SN, and in fact also uses XML formatted messages over standardized web-based protocols (WSDL via SOAP) to program nodes. Unfortunately, this work falls short in its limited ability to deploy truly arbitrary computations and its lack of a "single system" programming paradigm.

Perhaps the closest conception of a SN as a general-purpose, programmable infrastructure is the Distributed Token Machines (DTMs) work [11], which is in many ways similar to that of Active Messages [12]. The DTM approach provides participating sensors with a Token Machine runtime system that is able to accept and execute tokens written in Token Machine Language (TML). TML may be generated from higher level languages, so presumably one can generate programs at the network-level.[12] The DTM work is a highly-customized solution aimed at the particular constraints of motes [1]. Indeed, DTMs lack a middleware support infrastructure for application deployment (deployment is part of the TML program).

[12]To date, no high-level language exists that compiles into TML.

B. Catalytic Work

We envision snBench as a catalyzing agent for a number of interesting research directions both intrinsic (*i.e.*, research that aims to improve future generations of snBench) as well as extrinsic (*i.e.*, research that advances the state-of-the-art in other areas). The following are examples, inspired by the projected trajectories of active research projects currently being pursued within our department.

Extrinsically, snBench abstracts out the details of the SN infrastructure allowing researchers to work on the problems they are best suited to deal with. For example, vision researchers don't need to understand communication protocols, real-time schedulers, or network resource reservation to research HCI approaches for assistive environments [13]. Similarly, snBench provides researchers in motion mining [14] and in stream database applications [15] with a unique opportunity to implement and test proposed approaches and algorithms in a real setting. The functional, and strongly-typed nature of snBench programs may inspire the development of snBench (domain-specific) programming languages that are more expressive than SNAFU. In particular, SNAFU maps to STEP in a fairly straightforward way; additional more expressive front-end languages with less intuitive mappings could also be developed.

Intrinsically, the ability of the SSD to guarantee system performance could leverage advances in overlay network QoS management [16], distributed scheduling [17], and on-line measurement, inference and characterization of networked system performance [18]. Moreover, the algorithmic efficiency of the SSD will depend upon finding efficient solutions to labeled graph embedding problems [19], where those labels will have interesting interactions with the scheduling and performance issues already raised. SXEs ought to be high-performance runtime systems, and thus can benefit significantly from operating systems virtualization [20] and optimization techniques [21].

VII. Conclusion and On-Going Research

snBench provides an accessible, flexible, and easily extensible platform for developing distributed sensing applications in open SN networks. In this paper, we have provided more of a bird's eye view – as well as some details regarding initial implementations – of its various elements. With a first-generation, "proof-of-concept" snBench prototype in place, we are now turning our attention to a number of challenging problems, which we briefly highlight below.

Performance Profiling/Benchmarking: For an SSD to make judicious allocation of STEP subgraphs to SXEs requires the availability of performance/benchmarking data about the resource consumption of the various "opcodes" supported within an SXE. Said differently, for an SSD to successfully engage in scheduling and capacity-planning requires accurate characterization of the resource needs of the basic functions that comprise a STEP program. At present, we lack such characterizations. We invision a solution in which SXEs generate simple performance statistics about each opcode as it is run, and these are reported to the local SRM to build opcode performance profiles. Thus, as new opcodes are added their profiles can be dynamically built and probabilistically refined.

Scheduling and RealTime Type Systems: Performance monitoring infrastructure is clearly required to support runtime type constraints, as are realtime modifications to the SXEs. It is not clear what scheduling algorithms are best suited, or to what granularity we expect a program to be able to change the scheduling model on a given SXE to suit its needs (if at all). Related to this issue is the question of identifying appropriate type annotations to describe such scheduling and resource management constraints.

Graph Segmentation (Coloring) and Affinity: Communication between STEP nodes on separate SXEs is orders of magnitude more expensive than between STEP nodes on a single SXE. As such, there is performance pressure to make large, contiguous regions of the STEP graph the same color. At the same time, some programs by their nature draw samples from across an array of network-distributed sensors, and some very heavyweight operations (*e.g.*, face-finding) have computation costs which dwarf communication costs, particularly if they must share scarce resources; both of these (and other concerns) pressure us to have smaller contiguous color regions. What is the right balance, and how can it be stated algorithmically? How close can an online version of this algorithm come to an offline one? Can we also factor in a cost for re-coloring nodes, allowing us to utilize the space between the online color-each-node-once approach and the offline optimal-coloring one?

Expressive Naming and Name Resolution: Naming resources in the Sensorium is a great area of interest to us. At present we support naming of sensors via URI, relative to the physical SXE (host) that the sensor is connected to. This approach requires the Resource Manager to maintain knowledge of all sensors connected to each host and to prioritize some STEP program segments to compute physical sensor resources. Presently we can support a limited form of sensor naming by identity, yet wish to generalize this naming further with more powerful functions to support naming by identity (*e.g.* "The webcam in Azer's Office"), naming by property (*e.g.* "Any two cameras aimed at Michael's chair by 90 degrees apart"), naming by performance or topological characteristics (*e.g.* "Any processing element within 2msec from WebCam1 and WebCam2"), and naming by content (*e.g.* "Any webcam which sees Adam right now"). Such naming conventions will require persistent STEP queries to enable these lookups, and it is unknown as of yet which such persistent queries should be instantiated and run to produce the highest odds of success at the lowest cost.

Security: Security issues in open environments (such as those we target with SNBENCH) are paramount, requiring the incorporation of mechanisms that deal with authentication (*e.g.*, to ensure that unauthorized persons do not upload STEPs to the SSD or to SXEs), support for privacy constraints (*e.g.*, ensuring proper encryption of data streams traversing unsecured links), and trust (*e.g.*, certifiable enforcement of access controls to sensors, actuators, and historical data). Currently, we are considering the implementation of some of the more basic security functionalities – *e.g.*, using digest authentication for SSDs and SXEs, public key authentication for SXEs and SSL authentication for SSD, and using SSL (https) to preserve the privacy of data-plane and control-plane communication.

REFERENCES

[1] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David E. Culler, and Kristofer S. J. Pister, "System Architecture Directions for Networked Sensors," in *Architectural Support for Programming Languages and Operating Systems*, 2000.

[2] Benjamin C. Pierce, *Types and Programming Languages*, MIT Press, Cambridge, MA, USA, 2002.

[3] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler, "The *nesC* Language: A Holistic Approach to Networked Embedded Systems," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PDLI)*, 2003.

[4] P. Levis and D. Culler, "Mate: A Tiny Virtual Machine for Sensor Networks," in *International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA*, 2002.

[5] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong, "TAG: a Tiny AGgregation Service for Ad-Hoc Sensor Networks," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, 2002.

[6] Yong Yao and Johannes Gehrke, "The Cougar Approach to In-Network Query Processing in Sensor Networks," *SIGMOD Rec.*, vol. 31, no. 3, 2002.

[7] R. Govindan, J. Hellerstein, W. Hong, S. Madden, M. Franklin, and S. Shenker, "The Sensor Network as a Database," Tech. Rep. 02-771, CS Department, University of Southern California, 2002.

[8] Rimon Barr, John C. Bicket, Daniel S. Dantas, Bowei Du, T. W. Danny Kim, Bing Zhou, and Emin G&#252;n Sirer, "On the Need for System-Level Support for Ad hoc and Sensor Networks," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. 2, pp. 1–5, 2002.

[9] Ting Liu and Margaret Martonosi, "Impala: A Middleware System for Managing Autonomic, Parallel Sensor Systems," in *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, New York, NY, 2003, ACM Press.

[10] Flávia Coimbra Delicato, Paulo F. Pires, Luci Pirmez, and Luiz F. Rust da Costa Carmo, "A Flexible Web Service Based Architecture for Wireless Sensor Networks," in *ICDCS Workshops*, 2003.

[11] Ryan Newton, Arvind, and Matt Welsh, "Building up to Macroprogramming: An Intermediate Language for Sensor Networks," in *Proceedings of the International Symposium on Information Processing in Sensor Networks (IPSN)*, 2005.

[12] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser, "Active Messages: A Mechanism for Integrated Communication and Computation," in *19th International Symposium on Computer Architecture*, Gold Coast, Australia, 1992.

[13] J. J. Magee, M. R. Scott, B. N. Waber, and M. Betke, "EyeKeys: A Real-time Vision Interface Based on Gaze Detection from a Low-grade Video Camera," in *IEEE Workshop on Real-Time Vision for Human-Computer Interaction (RTV4HCI)*, July 2004.

[14] Tai-Peng Tian, Rui Li, and Stan Sclaroff, "Tracking Human Body Pose on a Learned Smooth Space," in *IEEE Workshop on Learning in Computer Vision and Pattern Recognition*, 2005.

[15] J. Considine, F. Li, G. Kollios, and J. W. Byers, "Approximate Aggregation Techniques for Sensor Databases," in *Proc. of the 20th IEEE Int'l Conference on Data Engineering (ICDE '04)*, Boston, April 2004.

[16] Mina Guirguis, Azer Bestavros, Ibrahim Matta, Niky Riga, Gali Diamant, and Yuting Zhang, "Providing Soft Bandwidth Guarantees Using Elastic TCP-based Tunnels," in *Proceedings of ISCC'04: IEEE Symposium on Computer and Communications*, Alexandria, Egypt, 2004.

[17] Azer Bestavros, "Load Profiling: A Methodology for Scheduling Real-Time Tasks in a Distributed System," in *ICDCS'97: The IEEE International Conference on Distributed Computing Systems, Baltimore, Maryland*, May 1997.

[18] Azer Bestavros, John Byers, and Khaled Harfoush, "Inference and Labeling of Metric-Induced Network Topologies," *IEEE Transactions on Parallel and Distributed Systems*, 2005.

[19] Jeffrey Considine, John W. Byers, and Ketan Mayer-Patel, "A case for testbed embedding services," in *Proceedings of HotNets-II*, November 2003.

[20] Yuting Zhang, Azer Bestavros, Mina Guirguis, Ibrahim Matta, and Richard West, "Friendly Virtual Machines: Leveraging a Feedback-Control Model for Application Adaptation," in *1st ACM/USENIX Conference on Virtual Execution Environments (VEE'05)*, June 2005.

[21] Richard West, Yuting Zhang, Karsten Schwan, and Christian Poellabauer, "Dynamic Window-Constrained Scheduling of Real-Time Streams in Media Servers," *IEEE Transactions on Computers*, vol. 53, no. 6, 2004.