SNBENCH:

Programming and Virtualization Framework for Distributed Multitasking Sensor Networks *

Michael J. Ocean Azer Bestavros Assaf J. Kfoury
Boston University
mocean@cs.bu.edu best@cs.bu.edu kfoury@cs.bu.edu

Abstract

We envision future Sensor Networks (SNs) that will be composed of a hybrid collection of a variety of sensing devices embedded into shared environments. In such environments it follows that the embedded SN infrastructure would also be shared by various users, occupants, or administrators of these shared spaces. As such a clear need emerges to virtualize the SN, sharing the resources of the SN across various tasks executing simultaneously. To achieve this goal, we present the SNBENCH (SN Workbench). The SNBENCH abstracts a collection of dissimilar and disjoint resources into a shared virtual SN. The SNBENCH provides an accessible high-level programming language that enables users to write "macro-level" program for their own virtual SN (*i.e.*, programs are written at the scope of the SN rather than its individual components and specific details of the components or deployment need not be specified by the developer). To this end SNBENCH provides execution environments and a run-time support infrastructure to provide each user a Virtual Sensor Network characterized by efficient automated program deployment, resource management, and a truly extensible architecture. In this paper we present an overview of the SNBENCH detailing its salient functionalities that support the entire life-cycle of a SN application.

Categories and Subject Descriptors C.2.4 [*Computer Communication Networks*]: Distributed Systems—Network Operating Systems

General Terms Management, Design, Reliability, Languages, Verification

Keywords Sensor Networks, Distributed Resource Management, Domain Specific Languages, Programming Environments

1. Introduction

We anticipate the emergence of embedded SNs, comprised of a collection of heterogeneous computers, sensors and actuators that are literally built into both public (or private, managed) physical spaces. These resources must be pooled into a single, dynamically tasked composite resource which may process requests from different users simultaneously. Indeed these users may task the SN with complementary, orthogonal or conflicting goals for simultaneous execution. Each user may approach this shared SN as his or her "private" SN, easily tasked with new programs and handling the scheduling, deployment, and management concerns transparently (*i.e.*, offering each user an abstract, Virtual SN). To achieve this goal we offer the SNBENCH, (SN Workbench) an accessible, flexible, and extensible programming platform and run-time infrastructure for the development of distributed sensing applications to run on the VSNs it creates.

Although SNBENCH is conceived (by design) to be agnostic to the specific sensing resources of a given SN (as its goal is to abstract the SN), we admit that our current implementation has been guided by the support of the particular SN infrastructure deployed in our laboratories, which we call the "Sensorium". Our Sensorium consists of a network of video sensors and motes[1] spanning several rooms, processing units and a terabyte database. We view our Sensorium as prototypical of the emerging VSN infrastructures whose use is shared amongst autonomous users with independent and possibly conflicting missions. These resources are managed together by the SNBENCHinfrastructure to enable the execution ad-hoc programs (*e.g.*, queries) specified over the Sensorium's monitored spaces.

The SNBENCH provides:

I. A high-level, functional-style network programming language (SNAFU) and a compiler to produce a Sensorium Task Execution Plans as output. SNAFU exists to ease development, provide type safety, and allow the developer a clear shift toward programming the network rather than the nodes.

II. The Sensorium Typed Execution Plans (or STEPs), a task- oriented, cycle-safe macroprogramming language used to assign computations to individual execution environments within the Sensorium. STEPS chain together core capabilities supported by the runtime environment of the SN to form a logical path of execution. Formally, a STEP is a directed acyclic graph of the computations requested of a single computing element. Given a program's representation as a DAG, parallelization and other optimizations become tractable. Notice that although a STEP is represented as a DAG we do support a limited iteration construct that can be used to emulate recursion.

III. Various run-time support components that monitor SN resources and schedule newly submitted STEP programs to available SN resources. The Sensorium Resource Manager (SRM) is responsible for maintaining a current snapshot of the available resources in the SN, while the Sensorium Service Dispatcher (SSD)

* This research was supported in part by NSF awards ITR-0205294, EIA-0202067, CyberTrust-0524477, and NeTS-050166.

VEE'06 June 14–16, 2006, Ottawa, Ontario, Canada.

is responsible for accepting new STEP programs for the SN and scheduling the tasks of the STEP to available resources. Optimally, the scheduler must identify tasks within the new STEP that match currently deployed tasks and reuse them as appropriate and find a partitioning of the STEP program that can be deployed to physical resources.

IV. A run-time virtualization environment for heterogeneous computing and sensing resources. The Sensor eXecution Environment (SXE) is a common runtime that provides sensing or computing elements of the SN the ability to interpret and execute dynamically assigned task execution plans. In part, the execution environment hides irrelevant differences between various physical components and exposes a unified virtualized interface to their unique capabilities. The sensor execution environment also allows the resource to be dynamically loaded new functionalities via the network.

In this paper, we provide a bird's eye view – as well as some details regarding initial implementations – of the various SNBENCH components. We frame the components with motivating examples in the domain of networked computer vision applications. Further motivation and high level details may be found in [2].

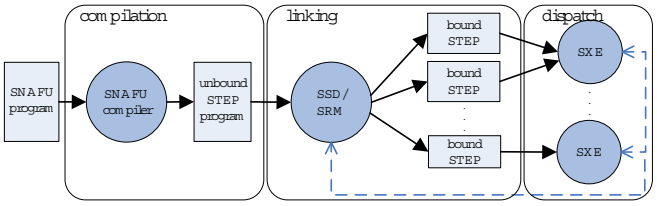

Figure 1. The SN program life-cycle as enabled by the SNBENCHRectangles represent data, circles represent tasks/processes, and the dashed lines represent control communication (*i.e.*, dependency).

2. The SNAFU Programming Language

The current SNBENCH programming interface is a high-level, network-oriented language called SNAFU (Sensor Network Applications as FUnctions). SNAFU is a strongly-typed programming language designed to specify the interdependence of sensors, computational resources, and persistent state that comprise a Sensorium application.

SNAFU programs are written in a simple functional style; all SNAFU program terms are expressions whose evaluation produces values. Consider the following example SNAFU program which inspects a single video frame, and returns the number of faces detected in the frame.

```
facecount(snapshot(sensor("s02","cam0"))))
```

A novice user will specify a SNAFU program written for the SN (not the individual nodes), and the infrastructure will transparently handle translation, resource allocation, and dissemination of the program to involved nodes of the SN. SNAFU is provided as a convenient and accessible interface to the Sensorium Task Execution Plan (STEP) language. In future iterations of STEP, we plan for STEP and SNAFU to diverge to a greater extent, as we wish to use the SNAFU interface as a restriction on the power of STEP [1].

SNAFU's direct benefit lies in its type-checking engine and STEP compiler. SNAFU programs are implicitly typed and dissallow explicit data-type annotation The type engine statically type checks SNAFU programs to identify typing errors and inserts permissible type promotions. SNAFU forbids explicit recursion (including transitive cases) instead providing an iteration construct that enables "simulated" recursion as described in the next section.

Cycle-safe iteration: SNAFU programs do not allow traditional recursion, but instead provide iterative execution through the "trigger" construct. A trigger indicates that an expression should be evaluated more than once, given some predicate expression. The repetition supported in SNAFU can be divided into two larger classes: terminating and persistent.

A terminating `trigger(x,y)` will repeatedly evaluate `x` until it is evaluated to be true, at which point `y` will be evaluated and the value of `y` is returned as the value of the trigger expression. The trigger will not execute `x` again after it has evaluated to be true (*i.e.*, it will terminate).

Alternatively, persistent triggers continue to re-evaluate their predicate "indefinitely". We provide two persistent triggers; the `level_trigger(x,y)` will continually evaluate `x` and every time `x` evaluates to true, `y` is re-evaluated. The "edge-trigger" construct `edge_trigger(x,y)` will continually evaluate `x` and when `x` transitions to become true from false (or if it initially evaluates to true), `y` is evaluated. The trigger's value is initially NIL and is updated to the value of `y` every time `y` is evaluated. The SNAFU trigger example below runs "indefinitely", repeatedly counting the number of faces found at the specified sensor.

```
level_trigger(true,
  facecount(snapshot(sensor("s02","cam0")))
)
```

In fact, persistent triggers typically live for a configuration-specific period of time (*e.g.*, one hour). To terminate a persistent trigger based on some run-time predication, the programmer may wrap the persistent trigger within a terminating trigger. Alternatively a persistent trigger may be wrapped in a "flow type" function allowing the programmer to specify a particular temporal persistence policy.

Simulated recursion: For some tasks, the body of a persistent trigger expression may need to make use of its own prior evaluations (*i.e.*, utilizing prior state, similar to tail recursion). SNAFU supports this via the `LAST_TRIGGER_EVAL` token, which acts as a non-blocking read of the closest enclosing parent trigger's value.

Naturally the results of persistent triggers may be used by other expressions as data sources for some other task. As the values of persistent triggers are transient and the temporal needs of the dependent expression may vary, we provide three different `read` functions that allow the programmer to specify a synchronization rule for the read of the trigger with respect to the trigger's own evaluation.[2]

Specifying a "`non-blocking`" read to a trigger requests the read immediately return the result of the last completed evaluation of the trigger's target expression, blocking if and only if the target expression has never completed an evaluation. A "`blocking`" read waits until the currently ongoing evaluation of the target expression completes, then returns that value. Finally a "`fresh`" read waits for a complete re-evaluation of the trigger's predicate and target expressions before returning a value.

Let assignments: SNAFU provides the ability to bind a value to a recurring symbol to either some persistent value (constant) or commonly occurring sub expression (macro).

[1] We have a forthcomming technical report that shows STEP is Turing complete; hence we intend to use SNAFU to restrict the programmer from the full potential (and pitfalls) of STEP.

[2] The value of a persistent trigger should always be read using one of these primitives. A program term containing an expression that directly accesses the value of a persistent trigger will be rejected by the SNAFU type engine. Terminating triggers, on the other hand, have an implicit blocking semantic and should not be wrapped by read primitives.

The `letconst` directive assigns a constant term or expression to a symbol, such that the value of an expression is evaluated only once (when first encountered). All further occurrences of the symbol are assigned the computed value and will not be evaluated. In the following example, the resolution of the sensor `cam1` will only be performed once, at the first instance of `cam1` in `Z` All other instances of `cam1` in `Z` will refer to this same sensor. resolution request.

```
letconst cam1 = sensor(ANY,IMAGE) in Z
```

Alternatively symbols may also be used as a shorthand to represent longer (sub)expressions, each instance of which is to be independently evaluated (*i.e.*, macros). The `leteach` binding, "`leteach x = y in z`", replaces every occurrence of `x` in `z` with an independently evaluated instance of the expression `y`. Notice in the example below every instance of `cam2` in `Z` may refer to a different sensor.

```
leteach cam2 = sensor(ANY,IMAGE) in Z
```

Finally, SNAFU allows a symbol to be assigned within the scope of a trigger such that the symbol obtains a new value once per each evaluation of the trigger (*i.e.*, once per iteration). The `letonce` bindings, for use with trigger contexts, have the form "`letonce x = y in z`" and allows the expression `y` to be evaluated for the symbol `x` once per iteration of the containing trigger defined in `z`. Consider the use of the `letonce` binding in the following program fragment that continues from the previous example, the intent of which is to take an image sample from each camera once per iteration and then return the image sample that has the most faces in it in a given iteration.

```
letonce x = snapshot(cam1) in
letonce y = snapshot(cam2) in
  level_trigger(true,
    if-then-else(greater(facecount(x),facecount(y)),x,y)
  )
```

Run-time types: SNAFU allows program terms to be wrapped by "flow type" functions. Flow types provide explicit constraints for program deployment and/or execution in the Sensorium, providing type information for the control flow as well as the data flow. As examples, the programmer may require a particular periodicity for a trigger term's evaluation, or may wish to ensure that some computations are only assigned to a trusted set of resources. Other example flow types are given through-out the paper, however an exhaustive account of the nature (and semantic) of these flow types is beyond the scope of this paper. We refer the reader to our work on using strong-typing for the compositional analysis of safety properties of networking applications [3] for some insight as to how flowtypes could be type checked for safety.

SNAFU Compilation: SNAFU has been designed to ensure that the abstract syntax tree (AST) of a SNAFU program maps to a task dependency diagram in the form of a directed acyclic graph (DAG) with a single root.[3] Nodes in the DAG represent values, sensors or tasks while edges represent data communication/dependency between nodes. The graph is evaluated by lower nodes executing (using their children as input) and producing values to their parents such that values percolate up toward the root of the graph from the leaves. The SNAFU compiler transforms the AST of the SNAFU program into such a representation, which we call a "Sensorium Task Execution Plan" or STEP. The terms and expressions of SNAFU have analogous constructs (nodes) in STEP or clear encoding in the structure of the STEP graph. For example, the single-evaluation `letconst` construct is a directive to link a single subtree onto several parents, and the `if-then-else` function refers to the placement of a conditional (`cond`) STEP node.

3. Sensorium Task Execution Plan (STEP)

A Sensorium Task Execution Plan (STEP) is a specification of a Sensorium program in terms of its fundamental sensing, computing and communication requirements. A STEP is serialized as an XML document that encodes the directed acyclic graph (DAG) of the explicit task dependency (evaluation strategy) of a Sensorium program. The STEP language is used to describe (1) whole programs written in the scope of the entire Sensorium (*i.e.*, programs compiled from SNAFU that are either largely or entirely agnostic as to the specific resources on which their constituent operations are hosted) and (2) (sub)programs to be executed by specific individual sensor execution environments to achieve some larger programmatic task (*i.e.*, to task the individual Sensorium resources in support of (1)).

STEP is the preferred target language for the compilation of SNAFU (and other future languages) and as such we refer to STEP as the Sensorium "assembly language" (*i.e.*, STEP is our "lowest-level" Sensorium programming language). That said, STEP is a relatively high-level interpreted language. [4] We note that although STEP is the preferred target language for SNAFU compilation, for some constrained resources, running a STEP interpreter may not be desirable. In such situations we rely on gateway nodes to interpret STEP and relay requests on the nodes behalf. We use this approach within our current Sensorium to integrate Berkley Motes for temperature sensing. In a future generation of the SNBENCH we may consider providing a SNAFU compiler that produces targets more suitable for constrained devices (*e.g.* C99, Intel asm, *etc*).

Individual tasks within a STEP (*i.e.*, nodes within the STEP graph) may be "bound" to a particular SN resource (*e.g.*, some sensor sampling operation that must be performed at a specific location) while others are "unbound" and thus free to be placed anywhere in the SN where requisite resources are available. In general, SNAFU compilation results in the creation of an "unbound" STEP – a STEP graph containing one or more "unbound" nodes.

Unbound STEPs are analogous to unlinked binaries insofar as they can not be executed until required resources are resolved. Unbound STEPs are posted to a Sensorium Service Dispatcher (SSD), the entity responsible for allocating resources and dispatching STEP programs. Given the state of the available system resources and the resources required by the nodes comprising this graph, the SSD fragmenting the unbound STEP graph into several smaller bound STEP subgraphs.

In the remainder of this section we describe the "classes" of tasks (nodes) that are supported by the STEP programming language and convery with broad strokes the runtime semantic they convey. The reader should note a correlation between the node classes presented in this section and the presentation of the SNAFU semantic. Indeed, these constructs are a direct encoding of the functionalities presented in that section.

We frame this discussion within the context of the example STEP program given in Figure 2. 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"))))
```

[3] Although the SNAFU AST is a tree, the execution semantic of SNAFU is actually a graph. Consider the `letconst` binding that allows a single node to have multiple parents.

[4] Although STEP programs are technically human-readable, their lack of type checking and XML representation (including attributes which the user may have no interest or business assigning) make direct program composition in the STEP language inadvisable at best.

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

Figure 2. An unbound STEP program for computing the maximum number of faces detected from one of two cameras mounted on s05.

`step` **node:** The `step` node is the root node of a STEP and contains the entire STEP program. The node has an `id` attribute that is a globally uniquely identifier (GUID) generated by the SNAFU compiler, uniquely identifying this program (and all nodes of this program) from other programs. The immediate child of the `step` node is the true root node of the program.

`exp` **nodes:** An `exp` (*expression*) node conveys a single computing, sensing, storage, or actuator function to be performed by some Sensor eXecution Environment (SXE). An expression node has an `opcode` attribute that identifies which function should be performed, and the immediate children of the expression node are the arguments to the function. Example opcodes include addition, string concatenation, image capture, image manipulation, detecting motion, *etc*. Opcodes are core library' operations distributed with the SXE. If an SXE does not have the opcode required, the `jar` attribute may specify a URL where a Java bytecode implementation of this opcode can be found. Similarly, the `source` attribute may be used to specify the location of the Java source code for this opcode.[5]

`cond` **nodes:** A `cond` (*conditional*) node has three children: an expression that evaluates to a boolean value (*i.e.,* a condition), an expression that will be evaluated if the condition is true, and an expression that will be evaluated if the condition is false. The *conditional* node has an evaluation semantic that ensures the first child (sub-tree) is evaluated first and, depending on the result, only the second or the third child will be evaluated.

`sensor` **nodes:** A `sensor` node conveys a specific physical sensor within the sensor network, and is used to provide the sensor as an argument to some expression node. In the example given (Fig. 2), the two snapshot expression nodes each have a `sensor` node as a child to specify on which particular image sensor they will operate. Sensor nodes may have a `uri` attribute to indicate where the sensor can be found and will have a `type` attribute to indicate the type of input device that this node provides (*e.g.*, `image`, `video`, `audio`, `temperature`, *etc*). Sensor nodes only appear as leaves in a STEP graph. A sensor node requires additional processing on the SSD to resolve and reserve a "wildcard" sensors (*i.e.,* when the `uri` of the sensor is omitted).

`trigger`**,** `edge_trigger`**, and** `level_trigger` **nodes:** All `trigger` nodes specify that their descendants are subject to iteration as indicated by the corresponding *trigger* construct (explained in Section 2). Trigger nodes have two children: the predicate and the body. A trigger may also have zero or more `flowtype` nodes to convey the runtime/deployment QoS constraints of this trigger. Related to trigger functions, `read` **nodes** may appear as the parent of a `trigger` node to explicitly request specific temporal access to the values produced by that trigger. The `read` node's `opcode` attribute determines whether the trigger will be read via a "blocking", "non-blocking", or "fresh" semantic (also described in Section 2).

`flowtype` **nodes:** The `flowtype` nodes are used to encode run-time security, performance, and persistence constraints. These nodes appear as children of the nodes that they constrain.

`socket` **nodes:** `socket` nodes may be inserted into unbound STEP DAGs by the SSD during the binding (scheduling) process to allow distribution of a program's evaluation across SXEs. The `socket` node connects the computation graph from one SXE to another SXE across the network. A socket node has a `role` attribute which is set to either `sender` or `receiver`. A `sender` node takes the value passed up by a child node and sends it across the network to another SXE specified by the node's `peeruri` attribute. Assuming the peer SXE is hosting a corresponding `receiver` node, that `receiver` node sends this value along to its parent node allowing a STEP "edge" to span SXEs. A `protocol` attribute specifices which specific communication protocol should be used for data transfer (*e.g.,* HTTP/1.1 pull, HTTP/1.1 push).

`splice` **nodes:** A `splice` node is used as a pointer to another node, allowing the encoding of graphs within the tree-centric XML. The `splice` node indicates that the parent of the splice node should have an edge that is connected to the "target" of the splice node (the splice node has a `target` attribute specifies an id of another existing node). The splice node only exists when a STEP is serialized as XML, when deserialized, the edge is connected to the target node. Splice nodes may occur within a compiled STEP graph if some node/subgraph has multiple parents (*e.g.,* the "let" binding provided in SNAFU) or a splice may occur as a result of computational reuse allowing one STEP program to be grafted on to another. Splice nodes allow the SSD to reuse components of previously deployed STEP graph within newly deployed STEP graphs by replacing a common sub-graphs in the new STEP program with a splice node.

`const` **nodes:** The `const` node class is use to block the propagation of "clear" events during evaluation, in effect preventing the re-evaluation of its descendants (*e.g.,* to support a letonce binding). A const node will have exactly one child, namely the subgraph that we wish to limit to a single evaluation.

4. The Sensorium Service Dispatcher (SSD)

The Sensorium Service Dispatcher (SSD) is the administrative authority of and single interface to each "local-area" Sensorium. The SSD is responsible for allocating available concrete Sensorium resources to process STEP (sub)programs (*i.e.*, scheduling) and dispatching STEP (sub)programs to those resources. Each SSD is tightly coupled with a Sensorium Resource Manager (SRM) that maintains the state and availability of the resources within the Sensorium.

[5] The dynamic migration of opcode implementations raises clear security/trust concerns. We expect the SXE owner will maintain a white-list of trusted hosts from which opcodes can be safely retrieved.

The current SSD/SRM implementation is Java based and utilizes HTTP as its primary communication model; an HTTP server provides an interface to managed resources and end users alike. Communications are XML formatted and, for end users, responses are transformed into viewable interactive web page by XSLT. The HTTP namespace is leveraged to provide a natural interface to the hierarchal data and functionality offered by the SSD.

The SSD/SRM has two primary directives: Resource Management and STEP Scheduling and Dispatch. Both are described below.

4.1 Resource Management

The Sensorium Resource Manager monitors the state of the resources of its local Sensorium and reports changes to the SSD. Each computing or sensing component in the managed domain that hosts an SXE sends a heartbeat to its SRM, the result of which is used to populate a directory (hashtable) of all known SXEs and their attached sensoring resources. The heartbeat includes the SXE's uptime, sensing capabilities, and a scaled score indicating available computing capacity. Should an SXE miss a configurable number of heartbeats, or the SXE report an unexpected computing capacity change without notification of a "STEP complete" the SRM assumes the SXE has failed or restarted and informs the SSD of the change. The SRM's knowledge of the state of the managed Sensorium is essential to the SSD's correct operation in deploying and maintaining STEP programs.

When an SXE leaves the Sensorium (*e.g.*, SXE shutdown, reboot, or graceful exit) there may be impact to one or more running STEP programs as multiple STEP applications may be dependent on a single STEP node or resource. When the SRM detects that an SXE has left the Sensorium, the SSD will treat all STEP tasks deployed on that SXE resource as a new STEP program submission and try to reassign it (in part or in whole) to other available resources.[6] Updated socket nodes are sent to redirect those SXEs hosting sockets connected to the exitting SXE.[7] If no sufficient resources can be found to consume the STEP nodes that had been hosted by the old resource, we must traverse the dependency graph and remove all impacted STEP nodes (*i.e.*, programs).

4.2 Scheduling and Dispatch of STEP Programs

The SSD maintains a master, non-executable STEP graph consisting of all STEP programs currently being executed by the local Sensorium. Each STEP node in this graph is tagged with the GUID of the SXE on which that STEP node is deployed, such that this master STEP graph indicates what tasks are deployed in the local Sensorium and on to what resources.

When a STEP program is submitted to the SSD, the SSD must locate available resources for the newly submitted STEP graph, fragmenting the newly submitted STEP into several smaller STEPs that can be accommodated by available resources. We approach this task as a series of modules that process the unbound STEP program (figure 3). We present our approach for each of these modules, yet emphasize the benefit of this modular approach is that any module may be replaced with a different or improved algorithm to achieve the same goal.

Code Reuse: In this generation of the SNBENCH we assume that the reuse of computation is paramount. When a new STEP graph is posted, the SSD first tries to ensure that the scheduling of unbound expression nodes does not result in unnecessary instances

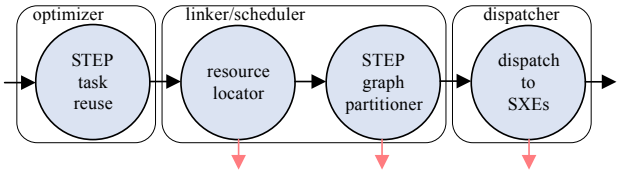

Figure 3. The SSD Scheduling and Dispatch process for STEP submission. Circles represent modules, while downward arrows indicate these modules may reject a STEP program due to insufficient resources.

of new tasks. Unfortunately checking for all instances of "functionally equivalent" tasks in a language as expressive as STEP is NP-hard.

Indeed, as many sensing functionalities will be dependent on the use of fresh data, our current code reuse algorithm is intentionally conservative to avoid the reuse of stale data. Unless explicitly specified with flowtypes, the SSD will only reuse nodes that are "temporally compatible". [8] Thus all new nodes that match already deployed nodes are replaced with splices (*i.e.*, pointers) to the previously dispatched nodes.[9] Regardless of how the code-reuse module is implemented, after it is complete the "new" computation cost of the submitted STEP graph should be reduced.

Admission Control: The SSD must deny admission if any bound STEP node refers an unavailable resource or if the remaining resource cost exceeds total available resources. The SSD first iterates over all bound nodes of the STEP graph to ensure that requested SXEs are known by the SRM and have available computing resources to consume these computations (including available sensors where sensors are prerequisites for a computation). Once complete, the SSD ensures that the total free resources in the Sensorium are sufficient to accommodate the total cost of the remaining unbound nodes.

Graph Partitioning: The SSD must bind all unbound nodes in the STEP graph to specific resources, a task analogous to a graph partitioning where each partition represents deployment on some SXE (*i.e.*, physical resource) the the goal of minimizing the total cost of edges between partitions (*i.e.*, minimize induced communication cost between SXEs). Fortunately, the computations represented at each node have associated datatypes and that type information yields a bound on the "cost" of each edge. For example, if a STEP node returns an Image, the communication cost of spanning this edge across two different physical resources (*i.e.*, adding this edge to the cut) will be greater than cutting and edge of a node that produces an Integer value (figure 5).

Our initial graph partitioning algorithm makes only a nominal attempt to reduce communication cost. The procedure tries to assign the entire unbound region of the graph to any single available resource. Failing that, the unbound region of the graph is split into smaller subgraphs and we recurse, trying to find a resource large enough to consume the "whole" parts.

Our next generation partitioning algorithm uses a relaxed form of spreading metrics [4] to produce partitions. A spreading metric defines a geometric embedding of a graph where a length is assigned to every edge in the graph such that nodes connected via inexpensive edges are mapped to be geometrically close, while

[6] STEP nodes that are submitted to the SSD pre-bound to some specific resource can not be migrated to another SXE and therefore must be terminated

[7] Synchronization issues abound when this occurs so a "reset" is sent to all nodes involved to restart communication in this event

[8] At present, temporal compatibility is ensured by reusing only identical, trigger- rooted subexpressions in the *local* Sensorium (giving us a tractable problem).

[9] There are certainly instances in which such blind bias toward computational reuse will result in a communication penalty that outweighs the benefit of code reuse, however have not accounted for this in our current SSD iteration.

```
<step id="202219@s00.sensorium.bu.edu">
   <exp opcode="max" id="abcd" bindto="http://c02.sensorium.bu.edu:8080">
      <flowtype name="persist" value="Dec 25 23:59:59 EDT 2005" />
      <exp opcode="facecount" id="bcde" bindto="http://c02.sensorium.bu.edu:8080">
          <socket id="face1:in" protocol="POST" role="receiver" peerid="face1:out"
          peeruri="http://s05.sensorium.bu.edu:8080/snbench/sxe/node/face1:out/"/></exp>
      <exp opcode="facecount" id="efgh" bindto="http://c02.sensorium.bu.edu:8080">
          <socket id="face2:in" protocol="POST" role="receiver" peerid="face1:out"
          peeruri="http://s05.sensorium.bu.edu:8080/snbench/sxe/node/face2:out/"/></exp></exp>
</step>

<step id="200507230943:a">
     <socket id="face1:out" protocol="POST" role="sender" peerid="face1:in"
      peeruri="http://c02.sensorium.bu.edu:8080/snbench/sxe/node/face1:in/">
         <exp opcode="snapshot" id="cdef"><value id="defg">
            <sensor type="snbench/image" uri=
            "http://s05.sensorium.bu.edu:8080/snbench/sxe/sensor/image/1"/></value></exp>
     </socket>
</step>

<step id="200507230943:b">
     <socket id="face2:out" protocol="POST" role="sender" peerid="face2:in"
      peeruri="http://c02.sensorium.bu.edu:8080/snbench/sxe/node/face2:in/">
         <exp opcode="snapshot" id="fghi"><value id="ghij">
            <sensor type="snbench/image" uri=
            "http://s05.sensorium.bu.edu:8080/snbench/sxe/sensor/image/2"/></value></exp>\\
     </socket>
</step>
```

Figure 4. Our previous STEP program for computing the maximum number of faces detected from either of two cameras mounted on s05. In this instance, the SSD has split the STEP graph into three STEP subgraphs for deployment on two separate SXEs, c02 and s05.

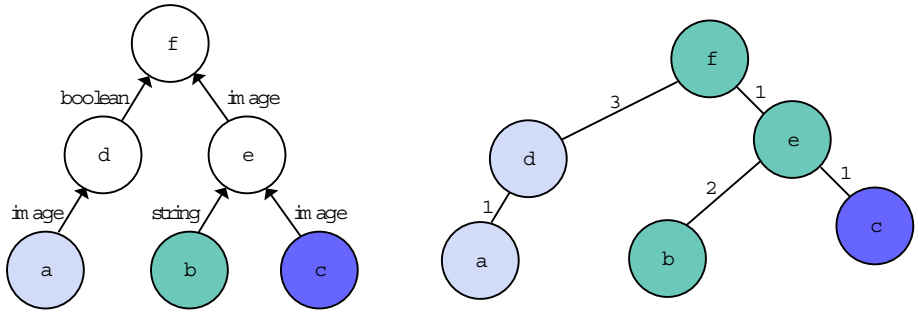

Figure 5. Generating colored partitions in a STEP graph. Coloring nodes is analogous to assigning a task to a particular SXE. Uncolored nodes should be colored to minimize communication between SXEs (colors) – there is no communication cost when adjacent nodes are the same color.

nodes across expensive edges are physically "spread apart" from each other.

The optimization detailed in [4] relies on a linear program to assign lengths to edges. Instead, we will use a "quick-and-dirty" approximation of the spreading metric, in which weights and distances for edges are derived entirely from the type information of the nodes (figure 5). Although this approximation will not yield partitions with the same bounds on minimizing the cut, our approach is favorable in running time and we can compute, off-line, the the minimum cut of the spreading metric to use as benchmark for comparison against our approximation algorithm. Again we point out that any graph partitioning solution may replace our existing partitioning logic, and are investigating some "off the shelf" solutions.

Dispatch: Once all STEP nodes are annotated with bindings the SSD must generate the STEP sub-graphs to dispatch to each individual resource. During this phase the SSD inserts socket nodes to maintain the data flow of the original STEP graph after the partitioning. As each SXE receives only a part of the larger computation (and sockets to SXEs with which it shares an edge) each is unaware of the larger task it helps to achieve.

To dispatch the STEP sub-graphs, the SSD performs an HTTP post of the STEP to the SXE's web server. If all SXEs respond to the dispatch with success, the SSD's dispatch is complete and the STEP program is live. If not, all partial STEPs of the larger STEP that had been posted to SXEs before this failed partial STEP are deleted from those SXEs and the user must resubmit.[10]

5. Sensorium Execution Environments (SXEs)

The Sensor eXecution Environment (SXE) is a runtime execution environment that provides its clients remote access to a partici-

[10] We do not attempt to re-optimize the Sensorium's global STEP graph (*i.e.*, all computations on the current Sensorium) when a new STEP is submitted. It is possible that a better, globally optimal assignment may exist by reassigning nodes across the global STEP graph however we expect the computational cost will far outweigh the benefit. At present, we don't intend to move computations once they have been initially assigned unless absolutely necessary (*e.g.*, in the event of resource failures).

pating host's processing and sensing resources. An SXE receives XML formatted Sensor Typed Execution Plans (STEPs) and the SXE schedules and executes the tasks described. Indeed an SXE is a virtual machine (figure. 6) providing multiple users remote access to virtualized resources including sensing, processing, storage, and actuators via the STEP program abstraction.

The SXE communicates its capabilities and instantaneous resource availability to its local Sensorium Resource Manager (SRM), allowing the Sensorium Service Dispatcher (SSD) to best utilize the each SXE. Each SXE maintains a local STEP graph containing a composite of all the STEP graphs (and subgraphs) tasked to this node by the SSD. In this section we describe the essential functionalities of the SXE and our current implementation of these functionalities. As is the case with the SSD, the SXE is also implemented with extensibility as a chief goal. We describe the SXE in terms of its necessary actions in support of the larger Sensorium via the STEP interface: STEP Program Admission, STEP Program Interpretation, STEP Node Evaluation and STEP Program Removal.

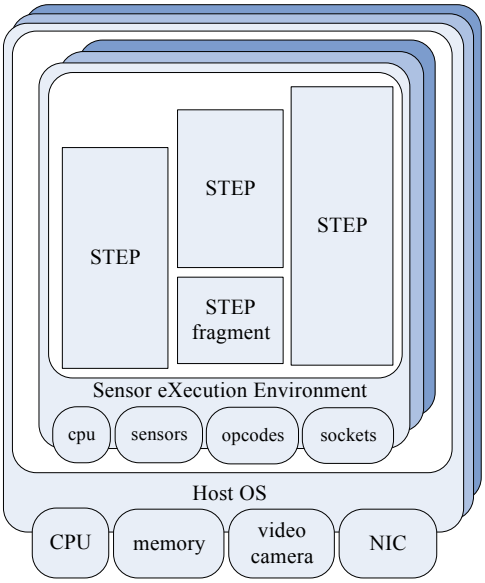

Figure 6. The SXE abstracts the devices provided by the OS, allowing clients to task these devices through the Sensorium Task Execution Plan (STEP) abstraction.

Implementation Overview: Our current implementation of the SXE uses Java technologies, in particular Java 1.5 for the runtime, Java Media Framework for sensor interaction, and Java based NanoHTTPD for HTTP communications. The use of the Java provides natural programming benefits of a strongly-typed language with exception handling. Java provides straightforward mechanisms for runtime loading of dynamic functionality over the network (via jar files or dynamically compiled source code). In addition, Java provides the protection benefits of its sand-boxed runtime environments and a virtual machine profiling API. The GCJ suite allows us to compile Java programs into native byte-code if performance is an issue.

To provide access to the sensing resources of an SXE, all physical sensors are abstracted as generic sensors (data sources) with specific functionalities implemented via classes on top of the generic sensor (e.g., ImageSensor, AudioSensor, etc.).

The size the jar file for the execution environment (SXE) containing all basic functionality including execution plan interpretation, evaluation, web server and client is about 200k uncompressed. We expect the SXE to it to be deployed on low-end desktop machines (Pentium Pro) and have not attempted to port to micro-devices at present. Instead, we have implemented opcodes that act as gateways to communicate with restricted devices. When the participating SXE host also provides video sensing functionality, the system requirements are increased by those of the JMF.

The SXE's primary mode of communication is via HTTP, acting as both a server and client, as appropriate. The SSD communicates with constituent SXEs via their HTTP interfaces and each SXE utilizes an HTTP client to communicate with the SSD, SRM, and other SXEs. Each SXE may also utilize other communication protocols to communicate with non-standard SXEs or non-SXE Sensorium participants (*e.g.*, motes, IP video cameras, *etc*). Data transfer between SNBENCH components is almost exclusively XML formatted, including Base64/MIME encoding of binary data. The SXE sends an XML structured hearbeat to the the SRM via an HTTP post from the SXE to the SSD. STEP graphs are uploaded to an SXE via HTTP POST of an XML object.

Sensorium Task Execution Plan (STEP) Admission: When a new STEP graph is posted to an SXE via the SSD, the new tasks to be executed may be independent of or dependent on previously deployed tasks. Within a newly posted STEP graph, the SSD may embed "splice nodes" in the new STEP graph specifying edges that are to be spliced onto previously deployed STEP nodes (*i.e.*, for task reuse) or nodes that should replace previously deployed STEP nodes with new nodes (task replacement).

1. Task Reuse: A newly posted STEP graph may contain one ore more "splice" nodes with target ids that point to previously deployed STEP nodes, indicating some new computations will reuse the results computed by existing tasks. Although the splice is specified by the SSD through its seemingly omniscient view of the local Sensorium, each SXE maintains local scheduling control to avoid race/starvation issues.[11]

2. Task Replacement: If a new STEP graph includes non-splice STEP nodes with the same (unique) IDs as nodes already deployed, this indicates these new nodes should replace the existing nodes of the same ID. The replacement operation may result in either removal or preservation of children (dependencies) of the original node, while parent nodes are unaffected (although those may modified through iterative replacement)[12].

STEP Interpretation: Recall a STEP is a directed acyclic graph in which data propagates up, through the edges from the leaves toward the root. Tasks appearing higher in the STEP graph are not be able to be executed until their children have been evaluated (*i.e.*, their arguments are available). Likewise, the need for a node to be executed is sent down from a root (parents need their children before they can execute however once executed they don't necessarily need to be executed again).

The SXE's local STEP graph may have several roots, as its graph may be the confluence of several independent STEP subprograms, however there is not necessarily a one-to-one mapping between the number of STEP graphs posted and the number of roots in the local STEP graph.

Each STEP node may be in one of four possible states: ready, running, evaluated, and blocked (figure 5, left). The SXE's role in interpreting a STEP program consists of (1) maintaining and updating the control flow of a STEP graph, advancing them through

[11] 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. At present, the SXE takes a "the customer (SSD) is always right" policy toward admission control.

[12] Notice that such node replacements may cause synchronization difficulties when replacements involve nodes that communicate across multiple SXEs. In general, we limit our use of replacement to redirect communication nodes to a replacement SXE when another SXE has left the Sensorium.

their state transitions (described in this section) and (2) the actual execution of STEP nodes that are in the ”running” state to enable the data flow of a STEP graph (described in the next section).

The SXE interprets its current STEP nodes by continually iterating over all nodes and checking if they are “ready” to be evaluated. A generic node is determined to be ready to be evaluated if it (1) is wanted by a parent node, (2) has fresh input from all immediate children and (3) has not been already executed already (this can be reset by a parent node to enable a node to run more than once as in the case of a node with a parent trigger).

Within our present implementation, all nodes are iterated over in a round-robin fashion to determine if they are “ready”. When non-expression nodes are ready they are evaluated immediately while expression nodes are placed in a separately serviced FIFO run-queue. This approach to evaluating STEP nodes is not unique. Indeed, the selection of which nodes to consider next amounts to a scheduling decision which may be constrained by QoS requirements, or other considerations (*e.g.*, frame rates, *etc*). In fact any scheduling algorithm may be swapped in to service the run queue without adverse effect on graph evaluation or the data flow (figure 5, right).

A ready node is evaluated by the evaluation function for its node class. When no STEP nodes are ready, the iterator sleeps until a new STEP graph is admitted or some other event (*e.g.*, clock, network, shutdown *etc*) wakes the iterator. Once a node has been evaluated it produces a value that is pushed up the graph (possibly enabling parent nodes who are waiting for fresh input from their children). For some node classes, the ready function may be overridden to accommodate a non-standard execution semantic. Persistent triggers, for example, are always wanted if they are orphans, however if a trigger has a parent node this node is only wanted if its parent in turn wants it. We call the reader’s attention to the subtle detail that, although persistent triggers should run indefinitely, they must not run asynchronously from a parent lest nested triggers would easily result in synchronization issues and race conditions.[13]

STEP Node Evaluation: Each STEP node class specifies its own evaluation function. The evaluation function of most node types maintains the runtime semantic of the STEP graph by updating any needed internal (including the execution state flag) and passing up values received from children. The exception to this trivial evaluation model is the evaluation of STEP expression (`exp`) nodes. In all cases, the expectation is that the evaluation function for the node will produce a value to its parents.

The evaluation of `trigger` nodes requires updating the trigger’s internal state, ensuring that first the predicate is evaluated and that the post condition will be evaluated (and re-evaluated) as per the trigger type and result of the predicate. Similarly evaluation of a conditional (`cond`) node maintains state and determines whether the second or third branch should be evaluated and returned depending on the evaluation of the first branch. A `socket` node’s evaluation sends or receives data along the socket, a `value` merely passes a serialized value up the tree and similarly `sensor` nodes are like `value` nodes in that they merely act as an argument to the immediate parent (`exp` node).

The evaluation of an expression `exp` node may take some time and as such the evaluation function for an expression node merely schedules the later execution of the expression node by a separate scheduler and execution thread (`exp` node evaluation should not block the entire resource). Expression nodes are tasks, analogous to the opcodes of the STEP programming language. These nodes are calls to fundamental operations supported by the SXEs (*e.g.*, addition, string concatenation, image manipulation, *etc*), yet the opcode implementations themselves may be dynamically migrated to the SXE at runtime as needed.

The SXE is distributed with a core library of basic “opcodes” implemented 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 the dynamic loading of new opcodes from trusted remote sources (*i.e.*JAR files over HTTP)[14].

Internally, all opcode methods manipulate snObjects, a first-class Java representation of various STEP datatypes. The snObject itself is a helper class that provides common methods that allow data to be easily serialized as XML for transmission between SXEs 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*. Opcode implementations are responsible for accepting snObjects, and returning snObjects such that the result may be passed further up the STEP graph. A sample example opcode is given below for illustrative purposes.

```
/* The addition Opcode */
snObject Call(snObjectArgList argv)
throws CastFailure, InvalidArgumentCount {
 return (snInteger)
          (argv.popInt() + argv.popInt());
}
```

While the implementation of an opcode handler must be in Java, within the body of the opcode, computations are not limited to Java calls (*e.g.* communication with remote hosts, execution of C++ code via the Java Native Interface, generation and transmission of machine code to a remote host, *etc*).

STEP Program/Node Removal: The removal of STEP nodes from the SXE may occur due to local or external events. When the evaluation of a STEP graph completes (either successfully or in error) the SXE reports the completion event with the STEP program ID to the SSD. The SXE may mark the local nodes for deletion if no other programs depend on these nodes (*i.e.*, If any ancestor node attached to these nodes has a different program ID than that of this program the SXE knows other programs are dependent on this computation). Externally requested node removal may be signalled by the SSD (for operational reasons or by request of an end user). Removal may be specified at the granularity of single nodes, however removal of a node signals removal of any parent nodes (dependent tasks) including those from different programs (assuming the SSD knows best).

In either case, the SXE does not immediately delete the nodes from its URI namespace, rather deletion is a two-phase operation, consisting of garbage marking followed by a later 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.

6. Putting it all together

We now illustrate the usage of SNBENCH by following a sample sensing application through-out its lifecycle. We assume that a Sensorium has been deployed, with SSD/SRM located at

[13] We are considering a flowtype synchronization annotation that would allow a persistent trigger to run ”independently” of its parent node as in some cases where the called function may not be able to keep up with the rate at which data is being generated yet a greater sampling rate is desirable (*e.g.*, triggers shared by two different programs running at different rates).

[14] We imagine that applets could be used to allow opcodes from untrusted remote sources, and new instances of VMs created to ensure complete protection from this untrusted code.

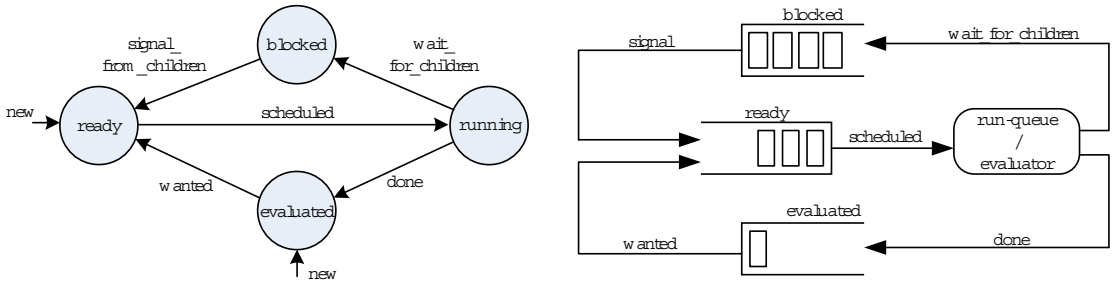

Figure 7. The SXE's STEP node evaluation state transition diagram. During evaluation, STEP expression nodes move between three buffers on the SXE

`ssd(.sensorium.bu.edu)` and with several participant SXEs. In particular, an SXE deployed on `lab-east(.sensorium.bu.edu)` is on-line and with an attached video sensor. `lab-east` advertises its computational and sensing resources via periodic heartbeats to `ssd`.

An end-user, Joe, would like to see the names and faces of people in that lab. Perhaps Joe doesn't know everyone's name yet, such that an image of the people currently in the lab, with the faces of people detected superimposed on the image would be useful to our user. As opcodes that support these functionalities (*e.g.*, grabbing frames, finding faces, *etc*) are available to the SXEs, this program is easily composed in SNAFU.[15] The SNAFU program to accomplish this goal would read:

```
letconst x =
   snapshot(sensor("lab-east","cam0")) in
     imgdraw(x,identify_faces(x))
```

The SNAFU compiler generates an unbound STEP graph, stored as XML. A shorthand of the STEP XML graph is shown below (some attributes have been removed for clarity). Notice the usage of the let binding in the SNAFU program results in the second instance of "x" in the STEP program being stored as a splice onto the same node.

```
<step id="joes-program">
  <exp opcode="imgdraw" id="root1">
    <exp opcode="snapshot" id="snap">
      <sensor uri=
      "http://lab-east/sxe/sensor/image/0"/>
    </exp>
    <exp opcode="identify_faces">
      <splice target="snap"/>
    </exp>
  </exp>
</step>
```

To submit the STEP program to the infrastructure, Joe will launch a web browser and navigate to the SSD that administers the Sensorium deployed in the lab[16] in this case: `http://ssd.sensorium.bu.edu:8081/snbench/ssd/`. An XSLT rendered HTML interface is presented by the SSD, one option of which allows Joe to upload the STEP program (XML file) to the SSD through a standard web-based POST interface.

[15] Recall the SXE is extensible such that if the functionality Joe wishes to accomplish on an SXE is not defined in the SXE core opcode library, Joe may develop his own opcode implementations in Java and make the jar file available via webserver. Such opcodes are accessible within SNAFU using a stub function new_opcode(" *uri*",args), however the usage cannot be typechecked so we warn against general usage of this feature by anyone other than opcode developers during testing.

[16] We plan to implement a DNS-style "root" SSD that will allow users looking to dispatch a STEP program to a particular resource to locate the SSD that administers that SXE (*i.e.*, SXE resolution that returns the location where programs for that node should be submitted).

The SSD then parses the posted STEP graph looking for reusable STEP components (*i.e.*, nodes). Assuming no STEP programs are deployed elsewhere in the Sensorium, the SSD proceeds to try and satisfy pre-bound computations. In our example, the sensor node is "bound" to `lab-east.sensorium.bu.edu` and the SSD's scheduler requires that any opcode immediately dependant on a sensor node should be dispatched to that same resource as the sensor. In practice, this is a reasonable restriction, as it ensures that the SXE hosting the sensing device will be responsible for getting data from the sensor. This will not create a bottleneck as additional STEP programs needing data from this sensor will share the need for the same opcode and reuse will occur attaching new computations to that opcode. In our example, the snapshot opcode will be bound to `lab-east` and the identify_faces and imgdraw opcodes are free to be scheduled to any available SXE resource (potentially including `lab-east`).

To make things more intresting we assume `lab-east` is only able to accommodate the snapshot opcode, so the STEP graph must be split across multiple SXEs. Fortunately, another SXE host on `c02` has available resources for both the identify_faces and imgdraw opcodes. Notice that if we were to split across three SXEs for these computations the Sensorium would pay the communication penalty for transferring the image twice (despite the splice). In this case we only transfer the image once and the socket is reused as a splice target. This is illustrated in the short- hand STEP sub graphs given below:

```
lab-east.sensorium.bu.edu:
<step id="joes-program:a">
  <socket role="sender" id="a" peer="b">
    <exp opcode="snapshot">
      <sensor>0</sensor>
    </exp>
  </socket>
</step>

c02.sensorium.bu.edu:
<step id="joes-program:b">
  <exp opcode="imgdraw" id="root1">
    <socket role="receiver" id="b" peer="a">
      <exp opcode="identify_faces">
        <splice target="b"/>
      </exp>
    </socket>
  </exp>
</step>
```

The SSD dispatches each STEP sub graph to the appropriate SXE (via HTTP/1.1 POST). If the POST at either SXE fails (*e.g.*, the SXE does not respond, fails to accept the STEP, *etc*), the SSD deletes the graph posted at the other SXE by sending a DELETE of the STEP graph's program ID. If both SXEs respond with success codes (200 OK), the SSD and SRM commit their changes and are updated to maintain this new program. The SSD presents Joe with a web page containing a successful POST result and an HTTP link

to the SXE node where he may (eventually) find the result of the computation:
`http://c02.sensorium.bu.edu/snbench/sxe/node/root1`. Optionally, as a security measure, the SSD may be used as a relay to prevent end users from directly connecting to SXEs. Joe may now navigate to that link or other presented links to node in the original STEP program tree and will see the current value or runtime state of each of the STEP sub computations.

As soon as the SXE has accepted the posted STEP program, its own web namespace will be updated to include the posted nodes and their current execution state and values. The SXE on `lab-east` has the lower part of the STEP graph (and no external dependency) such that it can immediately start executing its portion of the STEP graph. When the socket node is encountered, `lab-east` tries to contact `c02` and in doing so, provides `c02` with the data it needs to begin its execution. When `c02` computes a result for the orphan node "root1" it will contact the SSD informing it that the program "joes-program" is complete.

As a single-run (non-trigger) program, the STEP evaluators on each SXE will only run the computation nodes through once and after a configurable amount of time both the nodes and the result are expunged from the SXEs.

7. Related Work

We restrict the focus of our discussion of related work to only those efforts focused 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) [5, 6].

TAG [7] and Cougar [8] are examples of works wherein the SN is abstracted as a distributed data acquisition/storage infrastructure (*i.e.,* "SN as a database" [9]). These solutions are limited to query style programs and thus lacking extensibility and arbitrary programmability.

Regiment [10] provides a Macroprogramming functional language for SNs that is compiled to a Distributed Token Machines (DTMs) model [11] (DTMs are akin to Active Messages [12]). Although Regiment abstracts away many of the low-level management concerns of the mote platform, the DTM work is a highly-customized solution aimed at the particular constraints of motes [1] in which multitasking and sharing resources is not a concern.

MagnetOS [13] provides greater flexibility with respect to the programs that can be deployed, virtualizing the resources of the SN to a single Java virtual machine (JVM). While this approach supports extensible dynamic programming, it lacks the ability to share SN system resources across autonomous applications. One may also argue that a JVM is not the best abstraction for a SN.

The work of [14] most closely resembles our vision and approach toward a shared, multitasking, high powered sensor network. Microsoft's SONGs approach differs from ours insofar as (1) their semantic macroprogramming approach does not lend itself toward provisioning arbitrary computation and (2) reuse of computation is seemingly not on their radar.

The network virtualization of [15] must be mentioned as they face graph embedding challenges for their resource resolution goals. Their goal of network emulation on dedicated hardware is significantly different enough from our goal of a unified sensor network that it should be no surprise that our solution is more lightweight and requires less hardware infrastructure (*i.e.,* we do not require a dedicated system with the transfer of entire system images); We also plan to pursue the potential benefits of simulated annealing for our graph embedding challenges.

8. Conclusion and On-Going Research

SNBENCH provides a foundation for research that occurs both on-top of and within the SNBENCH platform. Users of the SNBENCH framework may develop distributed sensing applications that run on the provided infrastructure. Researchers developing new sensing or distributed computation methodologies (*e.g.,* the development of distributed vision algorithms, distributed hast tables, etc) may take for granted the communication, scheduling, and dispatch services provided by the SNBENCH freeing them to spend their energy investigating their area of interest and expertise. These modules can be provided as opcode implementations and plugged into the architecture with ease. Instead, in this section, we focus on the the research taking place within the components of the SNBENCH itself; that is, the development and research that extends the SNBENCH to improve Sensorium functionalities and meet the unique challenges of this environment.

RunTime Type Specifications: As data-type annotations convey the requirements and safety of a data flow, our notion of flowtypes extends type checking and safety to the control flow. Our work on flow types proceeds in two simultaneous directions (1) the analysis and generation of a palette of useful deployment and run-time constraints/types and (2) the use of the TRAFFIC [3] engine to check and enforce control flow sanity and safety.

Run-Time Support for Flow-Types: To support flowtypes, the SXEs must be modified to accommodate such scheduling parameters, including a monitoring infrastructure that ensures tasks are receiving the performance they have requested advertising an "accurate" real-time resource availability. Work is proceeding on the development of a hierarchical scheduler within the SXE, allowing STEP programs and even individual nodes to specify the scheduling model they require.

Performance Profiling/Benchmarking: Our present performance monitoring uses stub code to represent the free computational resources an SXE has and the computational cost of each opcode. It is clear that an accurate characterization of the computational availability of resources and each opcodes computational requirements will be needed to enable the SSD to *accurately* allocate resources and dispatch programs. We envision a solution in which SXEs generate simple performance statistics about each opcode as it is run, and these statistics are reported to the local SRM to build opcode performance profiles. Such an approach allows new opcodes to be developed with their profiles dynamically built and probabilistically refined.

Expressive Naming and Name Resolution: At present we support naming of sensors via URI, relative to the physical SXE (host) that the sensor is connected to, or the use of wildcards to specify "any" sensor of a given type. The use of URIs requires the Resource Manager to maintain knowledge of all sensors connected to each host and perform some priority computation to resolve resources to compute and reserve physical sensor resources. Assuming sensor resolution processing, we wish to generalize this sensor resolution 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 characteristics (*e.g.* "Any processing element within 2msec from WebCam1 and WebCam2"), and naming by content (*e.g.* "Any webcam which sees Assaf right now"). Such naming conventions will require persistent, prioritized STEP queries to be running as the basis for these results, however it is unknown which such persistent queries should be instantiated, the resource cost of allocating sensors for these tasks, and how we can express these tasks as more commonly used expressions such that we produce the highest odds of success at the lowest exclusive computation cost.

Graph Partitioning and Optimality: From the perspective of communication cost, there is performance pressure to generate STEP schedules (STEP graph partitions) in which contiguous regions of the graph remain in the same partition, to minimize communication between SXEs. Although we may use the data-types as an indication of the communication cost, it may be the case that those expressions that receive large amounts of data as input may have computation costs which dwarf their communication cost (*e.g.*, pattern matching, face-finding, *etc*). The deployment of such resource intensive expressions may generate graphs in which we have many small regions and high communication cost. We anticipate several iterations of algorithms that attempt to achieve the "right" (or configurable) balance between network and computation cost, including heuristics borrowed from spreading metrics [4], simulated annealing [16], and others.

Security and Safety: The emergence of embedded SNs in public spaces produces a clear and urgent need for well-planned, safe and secure infrastructure as security and safety risks are magnified. For example, a hacker gaining access to private emails or crashing a mail server is certainly bad, however it is clearly worse if that same hacker can virtually case an office via stolen video feed, disable the security system, remotely unlock the door, and steal both the physical mail server and the data it contains. The Sensorium is an ideal a testbed for dealing with the inherent security issues in this novel domain, requiring the incorporation of mechanisms that provide authentication support for privacy, constraints, and trust. 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 the SSD, and using SSL (https) to preserve the privacy communication between resources.

Scalability of Networked SSDs: As mentioned in previous sections, the SSD/SRM maintains resources for a "local-area" Sensorium. Although this hierarchal division seems rather natural, the number of resources to be monitored by an SSD must be "within reason". We do not yet have experiments to establish what "reasonable" number of resources our local-area Sensorium can support. Moreover, as more Sensoria come on-line, there will inevitably be demand for computations that involve resources of disjoint Sensoria (*e.g.*, nodes on the periphery between two SSD regions). Our initial approach is a tiered, DNS-like solution in which a root SSD can resolve specific resources beyond the scope of the local SSD and possible (when a local Sensorium is exhausted) out-source computations to another Sensorium. Such algorithms must be implemented, verified, and evaluated for scalability.

Acknowledgments

We would like to acknowledge Adam Bradley for his immense help and contribution to the initial conception of this work. We would also like to acknowledge those members of the larger iBench initiative at BU, of which this work is part. `http://www.cs.bu.edu/groups/ibench/`

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] Azer Bestavros, Adam Bradley, Assaf Kfoury, and Michael Ocean, "SNBENCH: A Development and Run-Time Platform for Rapid Deployment of Sensor Network Applications," in *IEEE International Workshop on Broadband Advanced Sensor Networks (Basenets)*, Boston, October 2005.

[3] Azer Bestavros, Adam Bradley, Assaf Kfoury, and Ibrahim Matta, "Typed Abstraction of Complex Network Compositions," in *ICNP'05: The 13th IEEE International Conference on Network Protocols*, Boston, November 2005.

[4] Guy Even, Joseph Naor, Satish Rao, and Baruch Schieber, "Divide-and-conquer approximation algorithms via spreading metrics (extended abstract)," in *IEEE Symposium on Foundations of Computer Science*, 1995, pp. 62–71.

[5] 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.

[6] 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.

[7] 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.

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

[9] 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.

[10] Ryan Newton and Matt Welsh, "Region streams: functional macroprogramming for sensor networks," in *DMSN '04: Proceeedings of the 1st international workshop on Data management for sensor networks*, New York, NY, USA, 2004, pp. 78–87, ACM Press.

[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] Rimon Barr, John C. Bicket, Daniel S. Dantas, Bowei Du, T. W. Danny Kim, Bing Zhou, and Emin Gün 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.

[14] Jie Liu and Feng Zhao, "Towards semantic services for sensor-rich information systems," in *Second IEEE/CreateNet International Workshop on Broadband Advanced Sensor Networks(Basenets 2005)*, 2005.

[15] R. Ricci, C. Alfeld, and J. Lepreau, "A solver for the network testbed mapping problem," 2003.

[16] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science, Number 4598, 13 May 1983*, vol. 220, 4598, pp. 671–680, 1983.