sdlib: A Sensor Network Data and Communications Library for Rapid and Robust Application Development

David Chu Kaisen Lin Alexandre Linares Giang Nguyen Joseph M. Hellerstein
EECS Computer Science Division
University of California, Berkeley
Berkeley, CA 94720
{davidchu, hellerstein}@cs.berkeley.edu, {kaisenl, alinares, cauthu}@berkeley.edu

ABSTRACT

Sensor network applications tend to exhibit significant high-level commonalities along several major dimensions that have heretofore been underexposed, particularly in the areas of collection and dissemination. We have developed a component library, Sdlib, which presents the fundamental abstractions of collection and dissemination as part of a dataflow sytem. This allows application developers to rapidly develop applications at the *nesC* level. This means that Sdlib maintains significant expressivity while operating efficiently.

We have built four applications, each faithful to a mature monolithic application, on top of Sdlib to compare its performance to that of original. We find that applications implemented with Sdlib are much simpler to write, just as resource efficient, and perform comparably to monolithic implementations.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures; D.2.13 [**Software Engineering**]: Reusable Software

General Terms

Design

Keywords

Wireless sensor networks, software library, collection, dissemination

1. INTRODUCTION

To date, the sensor network community has proposed many patterns of communication [1] [2] [3]. However, two communication patterns, *collection* and *dissemination*, have experienced far greater adoption and emerged as core to wireless sensor networks [4]. Collection, the gathering of data from all nodes in the network to one location, is, not surprisingly, a fundamental task of many sensor network deployments. Dissemination, the distribution of common data from one to all points in the network, has emerged as the complementary and equally critical task.

It is then surprising that little support has emerged for these often-used patterns, collection and dissemination. We too often observe the unlucky application developer still undertaking a sizable challenge when attempting to build her particular collection or dissemination-based application. For example, suppose our application developer desires to build a best-effort video monitoring application. If the popular nesC [5] is the programming language chosen, at a minimum, the novice must master split-phase asynchronous programming, sidestep insidious race conditions, and gracefully handle resource contention. Moreover, the non-expert and expert developer both face significant challenges building plumbing for handling control commands and network-wide data delivery.

It is unfortunate then, that our developer can only benefit minimally from another recently completed reliable-delivery vibration event detection application, which possesses both significant similarities (e.g. large data objects; query handling; Flash storage buffering) and differences (e.g. need for retries; monitored polling vs. event triggered) with her own. Yet in order to successfully make use of it, she must first know about the existence of this foreign application, entrust in its maturity, extract the relevant similarities and adapt them to suit her needs. Clearly this approach to reuse is not scalable (with the number of reusable applications), is error-prone, and is tedious.

Daunted by these obstacles, or simply by lack of knowing about pertinent similar applications, our application developer may choose to use systems exporting high-level languages, such as TinyDB [6], Snack [7], or Mate [8]. None of the aforementioned systems provides direct support for large data objects, a fundamental requirement for our example application. In general, such a system is a suitable choice only if the user's task is within the expressiveness of the chosen system.

These implications present an opportunity to build parameterized generic communication components that serve the purposes of a wide developer audience. The goal of this work is to identify common functionality among a broad range of sensor network applications yearning for appropriate abstractions, and develop a library of thoroughly-tested, reusable and efficient nesC components that present the fundamental high-level operations while parameterizing essential differences. We call this library *Sdlib*: Sensor Data Library. We draw an analogy to the traditional C++ Stan-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
IPSN'06, April 19–21, 2006, Nashville, Tennessee, USA.
Copyright 2006 ACM 1-59593-334-4/06/0004 ...$5.00.

dard Template Library. Sdlib provides powerful components for the recurring common cases. Simultaneously, because Sdlib is implemented as a collection of nesC components, the developer retains unfettered access to low-level operations when desired.

As a result, our objective has been to look at monolithic applications, extract the relevant communication patterns, analyze and distill dimensions along which variations were important, and then supply these variable pieces as subcomponents, from which the original communication protocol could be constructed. Along the way, we discovered that composition of these subcomponents could lead to incorporation into other well-known communication patterns, on top of which new classes of applications can be built. This process is summarized in Figure 1.

Sdlib will not eliminate asynchronous operations, race conditions, or resource contention. Eliminating these usually incurs an unacceptable system penalty. Rather, Sdlib enables the developer to relieve herself of a system full of such concerns and instead directs focus to the core application-specific module which can be more easily debugged. A set of composable components can greatly simplify the development task and mitigate the developer's worries.

Yet successful libraries offer generality without sacrificing efficiency. Efficiency of operations is particularly critical for sensor networks due to battery life, RAM/ROM and other resource constraints. Here Sdlib exposes policy decisions such as resource allocation and rate of operation to the developer, while hiding the mechanisms of policy enforcement. Sdlib aggressively uses compile-time information to create lean output.

Collection and dissemination then became the natural initial candidates around which to start. Sdlib is general enough to build a range of collection and dissemination applications while doing so efficiently. We demonstrate this by taking four real applications from across the collection and dissemination spectrum and building them on top of Sdlib. Our applications are comparable in terms of both performance and resource usage to the original, monolithic implementations. In some cases, we even outperform custom implementations in resource usage. We discuss the design and architecture of Sdlib in Section 2 and Section 3 respectively. Section 4 evaluates the implementation. Section 5 discusses the related work and Section 6 offers conclusions.

2. SDLIB DESIGN

Sdlib is comprised of a composable core set of services that work together to expose a consistent service interface to the developer. The user of our system is an application developer wishing to implement a new sensor network application rapidly, yet efficiently and correctly. To the developer, Sdlib provides a simple nesC get/set interface. The developer views the use of Sdlib primarily through the use of these interfaces. NesC, the primary language for the popular TinyOS wireless sensor network operating system [9], is a dialect of C that provides first class support for components as logical modules of code. NesC interfaces govern communication between these components. Communication between two components follows a hierarchical user/provider relationship, where the user calls down to the interface's commands and is the callee of the interface's events, whereas the provider is the callee of commands and calls up to events. NesC thus allows interface matching of

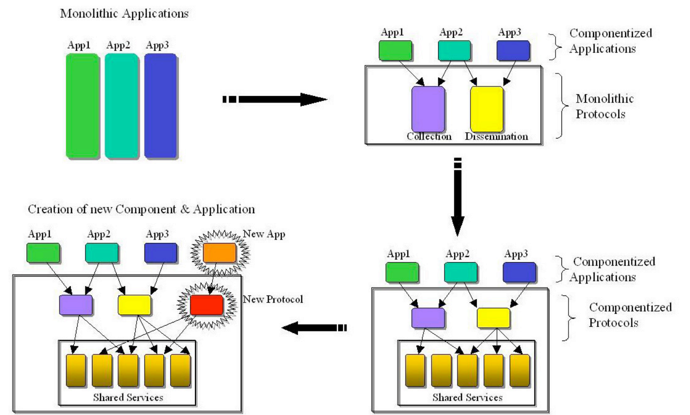

Figure 1: Transition from monolithic apps, to monolithic protocol implementations, to share protocol implementations, to protocols with new features.

components, or wiring, to be deferred until compile time.

A simple dataflow system allows for expressing a rich variety of possible tasks in Sdlib from a set of rudimentary components. In fact, this dataflow system has proven useful for implementing optimizations of dissemination, reliability of both collection dissemination, and is promising for a host of other features we mention below.

2.1 Primary Interfaces

Let us consider the case where the developer is interested in extracting a single object from the system. For example, this could be to read the node's routing table, or fetch the latest vibration sensor sample. Below, we show the get interface, `Producer`, for exposing such objects.

```
interface Producer {
  command result_t get(Buffer_t* buf,
                       bool first, bool last);
  event Buffer_t* give(Buffer_t* buf, uint8_t size,
                       bool first, bool last);
}
```

Here, the developer is the provider of this interface i.e. the developer implements the `get()` command and calls the `give()` event. On the other side, we can think of the user of this interface to be the collection operation (that wishes to dynamically get this data) or the dissemination operation (that wishes to update other nodes with the newest data held by this node). In reality, we interpose the *Sdlib Runtime Engine* between the user and provider such that all `get()` and `give()` calls are vectored through the engine. We will describe the operations of the engine in more detail below. The typical call sequence is:

```
engine: get()  // initiate stream
object: give() // filled buf
object: give() // filled buf
...
object: give() // filled buf, terminates stream
```

There is typically one `get()` call to initiate the stream, followed by one or more give calls that return the data. This provides a way for objects of any size to interact gracefully with Sdlib.

There are several points to note about this interface. First, the interface is inherently streaming. This provides a simple

baseline with minimal state maintenance needed by either party. It would be unreasonable to belabor an inherently streaming object with a non-streaming interface, though it is easier to adapt an inherently non-streaming object to export a stream. We later show the construction of more complex interfaces on top of this streaming interface.

Second, the units of data at the the level of this interface are in terms of `Buffer_t`. The engine is constantly providing buffers to the component, with which the component fills with the appropriate next piece of the object. The `Buffer_t` provided allows the component to fill in as much of a variable sized portion of the data object as desired, up to the size of the `Buffer_t`.

Third, the return value of the `give()` immediately provides a new `Buffer_t` upon which to operate. The process is driven at the producer's rate of data production, and is one of the keys to streaming effectively: Sdlib cannot make assumptions about the rate of data production on behalf of the object.

A primary goal of Sdlib is to achieve efficiency on par with hand-coded implementations. To do this, we assiduously avoid the use of extra buffers beyond those needed to accomplish the task at hand. For example, we could have taken a pointer to an arbitrary memory address and arbitrary length as arguments to `give()` and copied these into `Buffer_t` units on behalf of the developer. However, this unnecessarily requires allocation of more memory and requires more memory copies. Rather, each `Buffer_t` directly point to space in some preallocated network packet. (We elaborate on this in Section 2.2) The implications are that the Sdlib Runtime Engine operates with zero scratch buffers and needs zero memory copies in the entirety of the system. The tradeoff is application developers (1) become aware of these artificial `Buffer_t` size limitations and (2) are coerced to work with `Buffer_t` units. We have found that in practice, the former is not a concern, and that the object fragments generated at any one time by most applications are well within the size of one `Buffer_t`. With regards to the latter concern, we note that in some cases, it may be less convenient for the developer to manage operations in terms of `Buffer_t` units than in terms of units the application naturally generates. For example, a vibration sensor might sample at hundreds of hertz, at each interval generating an 10 bit sample. The natural thing to do here is work in the *Application Data Units* native to the application, e.g. 10 bit units. Therefore, we also provide a templated interface, `ProducerSimple` that achieves this goal when the ADU is known in advance at compile time and does not vary in size from one invocation of `give()` to another:

```
interface ProducerSimple<Adu_t> {
  command result_t get(Adu_t* buf,
                       bool first, bool last);
  event Adu_t* give(Adu_t* buf,
                    bool first, bool last);
}
```

Here, the interface `ProducerSimple` takes a type parameter `Adu_t` defined at compile time. Sdlib performs some internal manipulation to map this `Adu_t` to a space in some free `Buffer_t` on behalf of the developer, masking the rigidness of dealing in `Buffer_t` units. We again use compile-time parameterizations for efficient `Buffer_t` management.

At this point, it is helpful to discuss the interplay and distinction between large and small data objects in Sdlib. At the interface level, there is no distinction! With the `ProducerSimple` interface, we can iteratively signal `give()` as many times as necessary, including only once. If we only signal `give()` once, marking last right away, then we in effect provide a single small data object. In fact, this reduced case looks very familiar to Nucleus' `get()-getDone()` calls [10]. With only a small modification to the interface beyond what is needed for small objects, we are able to provide support for arbitrary sized objects as well. In the evaluation, we show support for data objects ranging from several tens of bytes to several tens of kilobytes.

Sdlib provides a set interface as well as a get interface for modifying data objects. We show this interface below:

```
interface Consumer {
  command result_t set(Buffer_t *pBuf, uint8_t size,
                       bool first, bool last);
  event result_t setDone(result_t r, Buffer_t *pBuf,
                         bool first, bool last);
}
```

Many of the same design decisions that influenced `Producer` are also visible here in `Consumer`. Similarly, Sdlib provides a `ConsumerSimple` interface that mitigates the effort of managing buffers.

2.2 Additional Interfaces

From the interfaces presented so far, we see that `Buffer_t` are passed back and forth between the Sdlib Runtime Engine, `Producer` and `Consumer`, but the responsibility of buffer allocation and management is not yet clear. On the one hand, forcing developers to manage buffers is extremely taxing and error prone. On the other hand, it is often the developer, not Sdlib Runtime Engine, that has knowledge about the appropriate quantity of resources to use. Sdlib offers a nice solution to this resource management issue. We partition the problem of allocation and management between the developer and Sdlib: the developer is simply responsible for resource allocation, whereas Sdlib Runtime Engine manages these on behalf of the user. To perform the allocation, the developer implements `giveBufs()` of the `Resource` interface:

```
interface Resource {
  command result_t giveBufs(WrapBuffer_t* bufs,
                            uint8_t* num);
}
```

With this flexibility in allocation, but without the burden of management, the developer is free to express any degree of resource sharing desired. For example, if the developer desires isolation from interaction with other objects, then an allocation strategy similar to the one below may be appropriate:

```
WrapBuffer_t localBufs[N]; // wrapper for Buffer_t
command result_t
  Resource.giveBufs(WrapBuffer_t* bufs, uint8_t* num) {
  *bufs = localBufs;
  *num = N;
  return SUCCESS;
}
```

On the other hand, the developer can express sharing of `Buffer_t` units from a single resource pool as well:

```
command result_t
  Resource.giveBufs(WrapBuffer_t* bufs, uint8_t* num) {
  call SomeSharedBufPool.giveBufs(bufs, num);
  return SUCCESS;
}
```

A problem that has increasingly manifested itself in sensor networks application development is the uncooperative interaction of services developed separately. The possibility for the developer to allocate resources wisely should help alleviate this problem. Later, when the sequence of `get()`-`give()` or `set()-setDone()` calls require `Buffer_t`, it is easy for the Sdlib Runtime Engine to manage the allocation and use of these across the entire system. In Section 3, we describe the internal handling of `Buffer_t` by Sdlib Runtime Engine.

So far the interfaces `Producer` and `Consumer` and their derivatives `ProducerSimple` and `ConsumerSimple` are passive: in the case of `Producer`, data is pulled from the interface. In the case of `Consumer`, data is pushed onto the interface. We observed this to be by far the most common programming usage model. However, there are certain cases where developers build components that actively push data outward (e.g. when the component itself is the generator of new program images) or that actively pull data inward to consume (e.g. when a component requests a program image to boot from at initialization). These push vs. pull issues often arise in dataflow systems, and have been explored extensively before [7]. We also provide for such *active* `Producer` and `Consumer` components with a slightly modified interface.

While the primary interface of Sdlib is stream-based, at times an additional random access interface to the data object may boost performance. A common example of this is when the underlying producer directly provides the fragments necessary for retransmission e.g. when the object is nonvolatile. In these cases, the developer implements a simple `seek()` call that permits random access. The Sdlib Runtime Engine calls `seek()` when it needs to perform such access.

2.3 Building Rich Dataflows

An object is not restricted to solely a producer or consumer role. In fact, permitting components to provide both `Producer` and `Consumer` interfaces offers great expressiveness. For example, a `ProgramImageStore` object can both provide and consume program images. In effect, the ability to interpose arbitrary intermediaries between production and consumption endpoints opens up a range of possible functionality. We provide both intra-node and inter-node examples below:

- **Storage manager:** A common intermediary that we frequently use in our construction of services for collection and dissemination, especially for reliability, is the general storage manager. By providing the `Consumer` interface for writing arbitrary data objects, and the `Producer` interface for reading arbitrary data objects, the storage manager uses no special mechanisms to store incoming data objects (e.g. program images) and deliver these to the appropriate endpoint (system reprogrammer); and buffer outgoing data objects for retransmission (e.g. reliably transported sensor values).
- **Object identifier and classifier:** Dependent upon run-time demands, the raw object data may not be necessary for the end user. For example, the developers of the Cyclops video camera mote [11] indicate it is often unnecessary to send an entire video frame back to the base station if the end user only needs to know whether or not an interesting entity is in the video frame. Here it is very sensible to incorporate a modular detection and classification filter [12] that consumes raw object data as input and produces a concise description of the event. Such a identifier and classifier could serve the purposes of a wide variety objects.
- **Binary verifier or rewriter:** A binary verifier or rewriter interposed between the program image store and the delivery mechanism for new images (e.g. inbound radio) can easily inspect all images delivered to the store. Authenticity checks can include verifying the key chain hash of the stream of program image fragments streaming by, as proposed in [13]. Alternatively, if the developer's deployment lacks preestablished authentication keys, an intermediary module can perform binary rewrites of the incoming image, mitigating manipulation of protected memory addresses [14].
- **Single-node compression:** It is often possible to compress data as it is being generated. Compression operators can either be data specific e.g. a Discrete Fourier Transform or general e.g. gzip. Such an intermediary operator appropriately placed immediately after the production of the sensor sample can serve to separate the roles of sampling from compression.
- **Production suppressor:** To regulate the flow of a particular object e.g. for application level network congestion avoidance, a suppression operator sits in the path between the object and its destination e.g. the radio's transmit queue. We construct dissemination suppression for our dissemination protocol with this technique.

These are all intermediary operators that act on the data generated at the same node. We are also able to construct operators that inspect the flow of data as it is routed in the network. These inter-node intermediaries are perhaps even more interesting:

- **Opportunistic consumption:** It is often the case that a node can benefit from the messages produced by neighbors in the network, even if the message is not destined for it. This is the case in dissemination: if a node overhears data messages of a greater version number than it currently possesses, it will proceed to snoop on the entire sequence of data messages. Such opportunistic consumption provides one of the main benefits of gossip-based dissemination. This is in fact the method by which we implement dissemination.
- **Neighborhood data sharing:** Hood [15] proposes a neighborhood abstraction for the purposes of data sharing. Such abstractions are shown to be useful for event detection applications. Providing a `Neighbor Consumer` and `NeighborProducer` should be similar to providing the opportunistic consumer above.
- **Model-based in-network compression:** We previously mentioned providing single-node compression via the Sdlib interfaces would be straightforward. Extending this idea, several works have found in-network model-based compression across nodes to be very effective [16]. The basic idea is for the network to save

communication costs by taking advantage of spatial correlations of data in the network: in effect, routing nodes do not need to send their own data if the data can be inferred from correlations with already transported data. Sdlib can similarly express these dataflow via interposed (across the network) operators.

Intermediary data operators are also by no means a new concept. However, despite dealing predominantly with data, sensor network tools have up until now provided few tools that expose data operators appropriately. Developers could have previously accomplished interpositioning without the aid of Sdlib by manually rewiring components at compile-time. However, this was a tedious operation that had to be carefully undertaken for each instance of interpositioning. In general, nesC and TinyOS do not provide the proper dataflow abstractions necessary for handling of data. The Sdlib Runtime Engine provides a common interface for any producer to have its output redirected to any consumer, and Sdlib provides a straightforward mechanism, *flow rules*, to easily accomplish this goal. We shall discuss the mechanisms of flow rules in more detail in the following subsection.

2.4 Flow Rules

Flow rules are simple specifications that link producers to consumers. An example flow rule takes the form: `Producer -> Consumer`. This is extremely similar to both component wiring in nesC and wiring in Snack [7]. The main differences here are that flow rules in Sdlib deal solely with data, and not arbitrary interfaces as in nesC nor packets as in Snack. Also, these flow rules can be redefined at run time.

Here we show two brief examples of an application writer's considerably less arduous task in specifying flow rules and writing application code.

2.4.1 Video

Our first example is for a Video sensor. Recently video sensors have been developed for several sensor network platforms, such as the Cyclops video camera for Mica motes and the onboard camera for the Intel iMote2. The application logic can be subdivided into the camera sensor driver, sensor parameter control and communication code.

To develop this application in Sdlib, the developer implements a very simple `get()` interface that consists of a few intuitive calls as shown above in Section 2.1. Committing to this interface is arguably much easier than dealing with the variability of multiple different sources to which to send data. Through either the use of a runtime tool or a compile-time configuration, the user may specify a simple flow rule that performs basic large data collection at every node:

```
Video -> Comm
```

Alternatively, the user might apply feature extraction on the images. Rather than sending the entire image, the extracted features of interest are then collected. This is similarly very straightforward with Sdlib:

```
Video -> FeatureDetector -> Comm
```

Another actor may wish to request video frames in their entirety. It is just as easy to incorporate reliability via retransmissions by specifying an additional set of rules. The rules below buffer each video frame into a backing store for servicing retransmission requests.

```
Video -> Comm
Video -> GeneralStore
GemeralStore -> Comm
```

So far we have restricted our demonstrations to intra-node flow rules. More interesting are data flows between different nodes. Below we show the specification of an inter-node dataflow rule which performs filtering at any intermediate node in the system:

```
Video -> Comm
Comm -> FeatureDetector -> Comm
```

The user only has to build the appropriate FeatureDetector and Video components in this case. The plumbing of routing the packet is taken care of by Sdlib. Here we omit the details of differentiating Comm streams. These are per data object type streams.

2.4.2 Mate

Our second example is an implementation of the Mate virtual machine using Sdlib. Mate is an application specific virtual machine runtime for TinyOS [8]. Mate uses Trickle-style dissemination [4] for propagation of capsules. Mate capsules are the virtual machine's units of runnable bytecode programs.

The capsule reprogrammer that uses capsule dissemination is shown below. Mate receives data from some source (typically the base station) and replaces its set of runnable programs with this data (first line). Concurrently, it participates in sharing this data with other network nodes (second line).

```
Comm -> CapsuleStore -> MateReprogrammer
CapsuleStore -> Comm
```

We can similarly expose an interface to allow any node in the network to reprogram other nodes, in the spirit of mobile agents. In fact, the remote reprogrammer would simply implement a `get()` interface similar to the one presented previously for video sample collection. Yet the flow rules in these cases use dissemination. We show a simplified exposition below. The second line indicates the node can also reprogram locally.

```
RemoteReprogrammer -> CapsuleStore -> Comm
CapsuleStore -> MateReprogrammer
```

This example replaces Mate's custom capsule dissemination mechanism with one written with Sdlib. The primary virtual machine runtime of Mate is still intact. It would also be reasonable to further use Sdlib to implement communication primitives for Mate bytecode programs.

3. SDLIB ARCHITECTURE

Figure 2 shows the component diagram of Sdlib, as well as several applications using the Sdlib interfaces. The Sdlib Runtime Engine is the central point of control. The user's parameterization of the system optionally brings in auxiliary components, such as the ReliabilityMan and CommandMan. In this way, Sdlib presents its library of services. Below we describe the interaction of these components with the Sdlib Runtime Engine.

The Sdlib Runtime Engine inherits from the Nucleus management system [10]. Sdlib continues to use the very flexible interface decorators and preprocessing techniques first

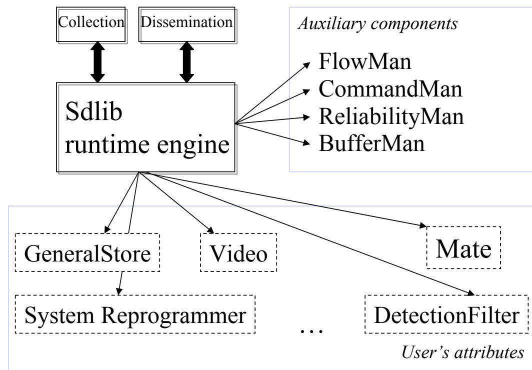

Figure 2: **Architecture of Sdlib.**

demonstrated in Nucleus, and borrows the small object collection mechanism directly from Nucleus.

The engine exports two communication interfaces, one through the collection channel and one through the data channel. Data received at either channel can be destined for any consumer and any producer can send data on either channel. In some cases, it is desirable for certain objects e.g. the system reprogrammer, to be strongly bound to a particular channel e.g. the dissemination channel. In other cases, it is desirable to defer the definition of producer-consumer relationships until run-time. Sdlib supports both of these cases. Compile-time configurations specify default flow rules among objects whereas run-time reconfiguration occurs through the insertion/deletion of new flow rules. Below we describe the main elements of the system:

Collection: The primary purpose of the collection component is to route data from the origin to the destination. Many specific instances of collection by routing tree have been built. These provide best-effort collection of individual packets. We simply use one of these and build our transport and application level-services on top [10].

Dissemination: Dissemination's primary purpose within the Sdlib is to exchange version messages among neighbors. Our dissemination implementation uses version vectors, similar to that described in [17]. When stale state is detected on a remote node, the Dissemination component contacts the Sdlib Runtime Engine to initiate data production to supply the neighbor with fresh data. When the stale state is detected on a local node, the Dissemination component requests the new data on behalf of the local node. Currently, the application developer specifies at compile-time which objects participate in dissemination and therefore need versioning support.

FlowMan: The FlowMan is the central router of the system. Given production from some object, the FlowMan determines the next consumer of the data object. FlowMan permits both runtime and compile-time flow rule definitions. Run-time reconfigurability is implemented by insertion into and deletion from a concise flow table. The FlowMan receives these requests from the CommandMan. Alternatively, compile-time flow rules do not require table entries nor lookups. Hence, FlowMan's resource usage and runtime overhead is low. By default, objects that produce values are sent up the collection tree and objects that consume values are idle.

CommandMan: The CommandMan is responsible for processing various commands that arrive externally. There are two types of commands. The first type of command installs a new set of flow rules into the FlowMan. The second type of command initiates a new data production or consumption in the system. These are roughly analogous to first setting up a series of stages for data processing and then invoking the initiating action that sets the dataflow in motion. The latter command, when for production, is similar to a query; when for consumption, is similar to a remote set operation.

ReliabilityMan: The ReliabilityMan controls the processes of reliably delivering and receiving data objects. The ReliabilityMan acts on behalf of both the collection and dissemination processes when either desires reliability. Reliability is always needed for Trickle-style dissemination, whereas it is often optional for collection. Our current reliability mechanism follows those reliability protocols commonly used in the literature and found in deployments: after initial data production and output to the radio, the producer awaits NACKs from the consumer indicating fragments it is missing. The ReliabilityMan is then responsible for retrieving the appropriate values and returning them to the requester.

Some object values, especially those that map to underlying hardware, such as sensor readings, may change without the involvement of the developer's application and are *volatile*. Other object values, such as the currently running program image, do not exhibit this behavior and are nonvolatile. Distinguishing between the two cases is important when we consider efficiency. In particular, objects that are nonvolatile do not need buffering for retransmissions - we can directly extract the missing fragments from the original object. On the other hand, volatile objects require distinct storage until the end of the reliable transmission sequence. By the application developer's assistance in marking objects either volatile or nonvolatile the ReliabilityMan handles these two cases appropriately.

BufMan: The BufMan is the manager of `Buffer_t` units throughout the system. The BufMan's primary responsibility is the delegation of buffers to producers for filling up, and then once filled, handing off of the buffers to consumers for consumption. The BufMan performs the locking and unlocking bookkeeping associated with proper management. Unlike monolithic application development, the application developer is largely relieved of the responsibility of resource management. The BufMan calls `getResources()` on the objects it is managing, as previously described in Section 2.2.

4. EVALUATION

4.1 Experimental Setup

We have built and evaluated an initial implementation of Sdlib, along with several applications using Sdlib. These implemented applications are faithful to widely-used and/or deployed applications:

- Mate (large, reliable dissemination) [8]: The Mate virtual machine runtime uses Trickle dissemination for propagation of capsules, Mate's units of packaged code, typically on the order of 100s of bytes.

- Deluge network reprogrammer (very large, reliable dissemination) [17]: The de facto network reprogrammer

for TinyOS is Deluge. It can reprogram up to a 64KB image. It also follows a Trickle dissemination protocol. It has particularly sophisticated interaction with the Flash backing store as well. The current implementation of Deluge is also very specific to network image dissemination.

- Golden Gate Bridge Application (GGB) (very large, reliable collection) [18]: This application monitors vibrations of the Golden Gate Bridge. Its requirements are that vibration spectra must be collected reliably from each node. Each vibration sample is a massive 500KB.

- UCLA Networked Cyclops (Video) (large, best-effort collection) [11]: A series of networked cameras each sends video frames back to the base station upon a user query. The video frames may be transferred without reliability. Depending on the image quality, a frame can range from 1KB to 16KB

It is also important to bear in mind the application space Sdlib supports but which we will not describe in detail in this paper. Unreliable dissemination *i.e.* a simplistic one-time flood has been well-studied in the literature. Also, since Sdlib builds on the Nucleus Management System [10], we are also able to provide unreliable collection of small attribute at marginal cost.

4.2 Study: Resource Usage

We next evaluated the resource usage of Sdlib. Resources measured are code size (ROM) and data size (RAM). We compared the size of our implementation of the four applications on Sdlib with the size of the original applications. Figures 3 and 4 show the comparisons for ROM and RAM respectively. In each graph, we show not only the total size of the final application, but also the breakdown between Sdlib and the application-specific portion. We expect the Sdlib-based implementation to use more resources since we are competing against custom implementations. We can see from the figures that in no case are we terribly less efficient than the custom implementation. For the ROM size comparisons, our implementations of Mate and Deluge are slightly smaller because our dissemination protocol omits several runtime optimizations, and hence compiles to a smaller image. For Video and GGB, we also built a component to handle the general case of user-initiated queries, accounting for the increase in ROM size in these two cases.

RAM usage is also comparable between Sdlib-based implementations and custom implementations, as shown in Figure 4. The discrepancy in GGB RAM usage results from GGB Monolithic storing its data readings in flash memory (not captured by these graphs), whereas the Sdlib-based implementation, stores them in RAM. Discounting this additional 2000B, the Sdlib-based GGB RAM usage falls to 1500B.

We also evaluated our resource consumption when combining multiple applications that use collection or dissemination in the same image. These scenarios present immediate opportunities to consolidate similar functionality, and are not uncommon in real deployments [19]. Figures 5 and 6 show our results when there is this opportunity to share. The Deluge/Mate combination yielded ROM savings of about two kilobytes and RAM savings of about 100 bytes over naively combining Deluge and Mate. The overlapping component between these two applications was only data dissemination. However, our Video/GGB combination yielded considerably more savings both in ROM and RAM use due to the fact that the data collection and buffer management were both done inside Sdlib. These results show the potential for a common library not only to ease application programmability, but also result in resource savings.

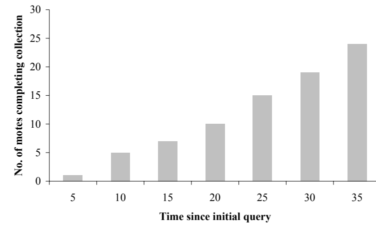

Figure 9: Collection comparison of average latency for a 24 mote network.

4.3 Study: Dissemination performance

To evaluate our dissemination performance, we compared the latency between our dissemination algorithm and the one used in Mate. We deployed a 22 mote network. Given an initial announcement of data, we timed how long it took for each mote to receive new data. Figure 7 shows our results. Our dissemination protocol performed comparably and in some cases even completed marginally faster.

Figure 8 compares the transmission counts of the three primary messages involved in dissemination for the Sdlib dissemination implementation and the Mate dissemination implementation: Version, Request and Data messages. For each class of messages, Sdlib sends a comparable number of these messages to those sent by Mate's implementation.

The conclusion to draw is that the Sdlib dissemination implementation uses a reasonably similar number of message transmissions to achieve similar levels of latency to the custom dissemination in Mate. These sanity checks assure us that applications implemented on Sdlib using dissemination do not suffer from using general dissemination protocols vs. customized alternatives.

4.4 Study: Collection Performance

We ran experiments testing our unreliable and unreliable collection implementations. We were interested in knowing whether our library implementations perform comparably to custom implementations on the metrics of collection bandwidth and latency.

On a 24 node testbed, we collected medium-sized objects (100s of bytes) from all the nodes in the network. Figure 9 shows the cumulative distribution function time to completion latency for the nodes in the network. Just as in dissemination, there is a long tail of nodes that eventually complete this procedure. Let us recall that this is due to our collection protocol which iteratively re-requests until we receive all the data we expect to collect.

We also measured the bandwidth in a simple 2-node scenario by collecting a very large 4 kilobyte data object, similar to that performed in [18]. With Sdlib, we achieved a bandwidth of 18.5 packets per second. This is slightly lower, but on the same order of magnitude as the 29.4 packets

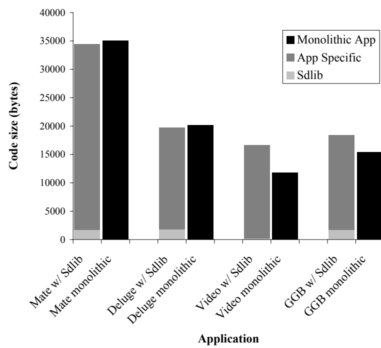

Figure 3: ROM comparisons for the original application and the application using Sdlib.

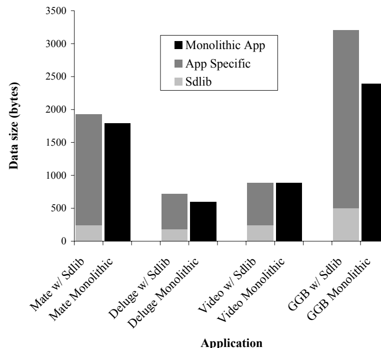

Figure 4: RAM comparisons for the original application and the application using Sdlib.

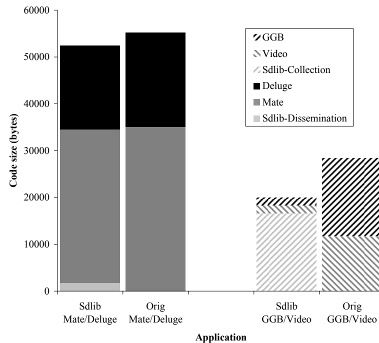

Figure 5: ROM comparisons for application combinations.

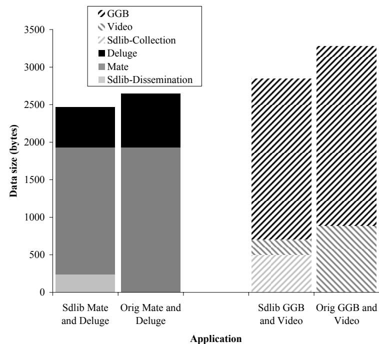

Figure 6: RAM comparisons for application combinations.

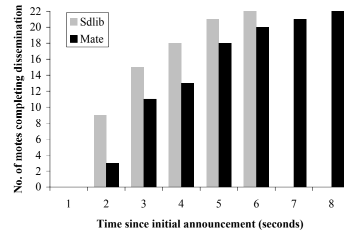

Figure 7: Dissemination comparison of latency times for motes to receive new data.

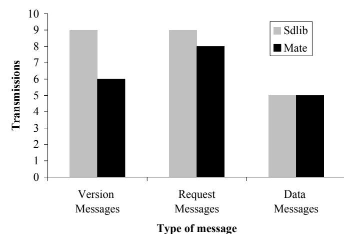

Figure 8: Dissemination comparison of the transmissions of various message types during a 30-second time interval.

	Lines of code	
App	Monolithic	App + Sdlib
Video	164	158
GGB	342	158
Mate	669	73
Deluge	410	60

Table 1: Lines of code comparison.

per second achieved in [18]. Several differences may have contributed to this, including the fact that our mote platform runs a slower microprocessor than the one used in [18]. Interestingly, we find that in the 2-node case, the proportion of retransmitted packets is negligible compared to the number of total packets transmitted. Therefore, our bandwidth of unique packets/second is very nearly also 18.5 packets/second. This mirrors results found in [18].

4.5 Study: Programming simplicity

We sought to make programming a simpler task for the application developer. We have accomplished this primarily by exposing a simpler interface for the application writer. Table 1 shows the lines of code written for the same operation with Sdlib and without in the original monolithic application.

5. RELATED WORK

The initial task of diffusion largely focused on the multihop nature of the network. The obvious first candidate is naive flooding, where successive receivers of a node would broadcast as long as they had received this message for the first time. Applications that truly used dissemination were very few: the unreliability of the communication often prohibited adoption for control data, though it was used for sending queries in TinyDB for example. Here, it was permissible, though undesirable, for portions of the network to continue executing under the old routines. Due to the high variance of the underlying radio hardware, this proved to be too unpredictable for many of target scenarios.

Due to demands for reliability, several robust flooding solutions have been developed. The Trickle protocol, a gossip-like protocol provided reliability in the form of eventual consistency, and efficiency with neighbor overhearing and self-suppression. As a result, several new applications emerged. Primary among these are Deluge, which disseminates program images for whole node reprogramming, and the Mate virtual machine [8], which disseminates VM programs for node retasking. More recent query systems, such as the Nucleus, use Trickle [4] for the purposes of disseminating a new query, that which TinyDB originally used flooding for. However, we note that each of these implementations, though written for the same platform interface, TinyOS, share much overlapping functionality, but almost zero code.

At the minimum, this vertical software silo construction has gone on long enough that we can do better, and we are missing the great opportunities associated with abstractions by not doing so.

Collection on sensor networks was traditionally done by constructing tree networks that dictated the data flow. For example, TinyDB originally used this method for data collection [6]. On the opposite spectrum, the collection of very large data items, such as video frames, as opposed to small items like sensor readings, has become important. Because of this, new abstractions had to be built to allow for large data transfer [18].

Nucleus [10] provides a nice foundation from which Sdlib has emerged. Nucleus is a management system that exports simple user variables in an easy to use format. However, Nucleus performs only collection of small (less than 1 packet in size) variables. We have made a series of modifications and redesigns to support the full dataflow system that Sdlib offers.

Sdlib also draws on work from two previous areas, namely data flows in networks and databases, and programming abstractions. Sdlib's internal flow rules from application to application borrows ideas from the Eddy adaptive query processing operator [20]. Eddies allow the dynamic flow of tuples to change while they are pipelined. In our system, we can also change the flow of data at runtime by adjusting flow rules through the radio. Similarly to how Eddies routes tuples through a query processor, we route packet buffers through a mote. This is also similar to other dataflow systems such as Snack [7]. However, Sdlib acts on data, rather than network packets. Data is arguably the more important concern in sensor networks.

6. CONCLUSION

We have constructed Sdlib, a library of components backed by a runtime engine that substantially alleviates the application developer's task of application development, especially in efforts that center around collection and dissemination, two main tasks in sensor networks. As ongoing work, we are actively investigating the incorporation of additional communication patterns into Sdlib.

In addition to significantly easing development, Sdlib offers a rich dataflow model on which more intricate dataflows can be constructed. We use this dataflow system to support many of the operations common to Sdlib, and in turn, use Sdlib to construct four mature applications. Benchmarking against the monolithic implementations of these applications, Sdlib-based implementations perform well. We invite application developers to try out Sdlib:

`www.cs.berkeley.edu/~davidchu/sdlib`

Acknowledgment

The authors thank Wei Hong, Phil Levis and Sam Madden for insightful discussions. This work was supported by NSF Grant 0205647, a gift from Microsoft Corporation, and the NSF Graduate Research Fellowship Program.

7. REFERENCES

[1] B. Karp and H. T. Kung, "GPSR: greedy perimeter stateless routing for wireless networks," in *Mobile Computing and Networking*, 2000, pp. 243–254.

[2] R. Fonseca, S. Ratnasamy, D. Culler, S. Shenker, and I. Stoica, "Beacon vector routing: Scalable point-to-point in wireless sensornets," in *In Second Symposium on Network Systems Design and Implementation (NSDI)*, 2005.

[3] C. Intanagonwiwat, R. Govindan, and D. Estrin, "Directed diffusion: a scalable and robust communication paradigm for sensor networks," in *Mobile Computing and Networking*, 2000, pp. 56–67.

[4] P. Levis, N. Patel, D. Culler, and S. Shenker, "Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks," in *In First Symposium on Network Systems Design and Implementation (NSDI)*, 2004.

[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 *In ACM SIGPLAN Conference on Programming Language Design and Implementation, 2003.*, 2003.

[6] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "Tinydb: An acquisitional query processing system for sensor networks," *Transactions on Database Systems (TODS)*, March 2005.

[7] B. Greenstein, E. Kohler, and D. Estrin, "Snack: Sensor network application construction kit," in *Proceedings of the Second ACM Conferences on Embedded Networked Sensor Systems (SenSys)*, 2004.

[8] 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, USA*, Oct. 2002.

[9] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister, "System architecture directions for networked sensors," in *Architectural Support for Programming Languages and Operating Systems*, 2000, pp. 93–104.

[10] G. Tolle and D. Culler, "Design of an application-cooperative management system for wireless sensor networks," in *Second European Workshop on Wireless Sensor Networks (EWSN)*, 2005.

[11] M. Rahimi, R. Baer, O. I. Iroezi, J. Warrior, D. Estrin, and M. Strivastava, "Cyclops: In situ image sensing and interpretation in wireless sensor networks," in *Proceedings of the Third ACM Conferences on Embedded Networked Sensor Systems (SenSys)*, 2005.

[12] L. Gu, D. Jia, P. Vicaire, T. Yan, L. Luo, A. Tirumala, Q. Cao, T. He, J. Stankovic, T. Abdelzaher, and B. Krogh, "Lightweight detection and classification for wireless sensor networks in realistic environments," in *Proceedings of the 3rd ACM Conference on Embedded Networked Sensor Systems*, 2005.

[13] P. Dutta, J. Hui, D. Chu, and D. Culler, "Secure network reprogramming," in *Information Processing in Sensor Networks 2006*, 2006.

[14] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," *ACM SIGOPS Operating Systems Review*, vol. 27, no. 5, pp. 203–216, December 1993.

[15] K. Whitehouse and D. Culler, "Hood: a neighborhood abstraction for sensor networks," in *ACM International Conference on Mobile Systems, Applications, and Services, Boston, MA, USA*, Jun. 2004.

[16] D. Chu, A. Deshpande, J. Hellerstein, and W. Hong, "Approximate data collection in sensor networks using probabilistic models," in *22nd IEEE International Conference on Data Engineering (ICDE 2006)*, 2006.

[17] J. W. Hui and D. Culler, "The dynamic behavior of a data dissemination protocol for network programming at scale," in *Proceedings of the Second ACM Conferences on Embedded Networked Sensor Systems (SenSys)*. ACM Press, 2004, pp. 81–94.

[18] S. Kim, "Wireless sensor networks for structural health monitoring," in *UC Berkeley Master's Thesis*, 2004.

[19] "Nest final experiment," 2005.

[20] J. Hellerstein and R. Avnur, "Eddies: Continuously adaptive query processing." in *Proceedings of the 2000 ACM SIGMOD international conference on on Management of data*, 2000.