A Sensor Network Application Construction Kit (SNACK)

Ben Greenstein Eddie Kohler Deborah Estrin
University of California, Los Angeles
{ben, kohler, destrin}@cs.ucla.edu

ABSTRACT

We propose a new configuration language, component and service library, and compiler that make it easier to develop efficient sensor network applications. Our goal is the construction of *smart application service libraries*: high-level libraries that implement concepts like routing trees and periodic sensing, and that combine automatically into efficient programs. Important language features include flexible control over component sharing and *transitive arrow connections*, which let independently-implemented services knit themselves into integrated control flow paths. Our language, library, and compiler are collectively called SNACK (Sensor Network Application Construction Kit). We describe them, and present and evaluate a simple SNACK-based multihop data collection application. This application uses SNACK language features to provide both simplicity (excluding reusable service definitions, its description is three lines long) and efficiency (it performs comparably to the well-known Surge application).

Categories and Subject Descriptors: D.3.2 [**Software**]: Language Classifications—*Specialized application languages*
General Terms: Languages, Design, Performance
Keywords: Configuration Languages, Sensor Networks, TinyOS, NesC

1 INTRODUCTION

An increasing number of increasingly complex deployment-grade applications are being developed for the mica sensor platforms [3, 11, 12]. These applications share the goal of efficient energy and resource usage, since efficiency can greatly prolong the life of a network. Radio communication is often the overriding long-term cost on untethered and unmanned systems [1, 15], so protocols have been designed to minimize bits sent and received. (Other potential energy consumers, such as the EEPROM and some sensors, also require care.) On the mica2, which has 128 KB of ROM and 4 KB of RAM, memory allocation must be minimized and code must be compiled to a small binary.

Unfortunately, efficiency comes at the great cost of program complexity. Application developers typically reach into and manipulate system-level code to provide optimized service for an application's needs, but this makes sensor systems more fragile overall.

Overcoming complexity requires language support. Just as high-level languages were developed when the complexity of programming with assembly language became too difficult for humans, component programming models such as nesC's [6] greatly reduce the complexity of embedded sensing systems development. Unfortunately for application developers, component models don't yet support the construction of efficient, independent *application-level* service libraries that can be reused from one application to the next. Individual components can be reused, of course, but outside low-level services like a radio stack, collections of components are difficult to reuse; and manipulating individual components is far too low-level for non-computer scientists.

This lack of reusability is generally due to the performance constraints under which sensor applications must run, since the cross-dependencies between services, and between services and lower-level components, necessary to gain efficiency simultaneously prevent modularity. For example, an efficient sensor application might want to aggregate sent data onto as few packets as possible, to reduce expensive transmission costs. This complicates library construction: How can two libraries, written independently, cooperate to share a single outgoing messaging path, and make that path activate at some rate acceptable to both libraries?

To address these problems, we have developed a new component composition language for sensor networks, and a library of components and services designed from the ground up to take advantage of that language's features. The resulting system is called SNACK, the Sensor Network Application Construction Kit. The SNACK system leverages nesC—its base components are written in nesC, and its compiler generates a nesC configuration and several nesC modules—but its language interface and compilation techniques are independent, and would continue to apply even if the underlying language were changed. SNACK's goal is the construction of *smart libraries* that weave themselves together into efficient applications: it should be possible to write a very short application description that is then compiled into a tightly-integrated, efficient application. For example, to write a simple multihop data collection application that periodically takes temperature and light readings and forwards the resulting data up to some sink, the application programmer should write two simple lines of code, roughly like:

```
SenseTemp -> [collect] RoutingTree;
SenseLight -> [collect] RoutingTree;
```

With SNACK's language, component and service library, and compiler, this code, and the generic services to which it refers, can be expanded into the tight, fast configuration shown in Figure 1.

Section 2 describes the SNACK language in detail, concentrating on its features that support smart library construction: configuration-level tunable parameters, controlled sharing of components and application-level services, and *transitive connections*, which construct integrated component paths from independent parts. Section 3 then presents our component and service library, which uses these language features to provide a higher-level API than current languages, and Section 4 shows how the language is compiled. To evaluate SNACK, we examine the performance and resource usage of several SNACK applications, including a more-developed version of Figure 1 (see Figure 2 for the code). This SNACK Forwarder application performs on par with a hand-configured application called Surge, despite its origins in very-high-level code. The remaining sections discuss related and future work and conclude.

SenSys'04, November 3–5, 2004, Baltimore, Maryland, USA.

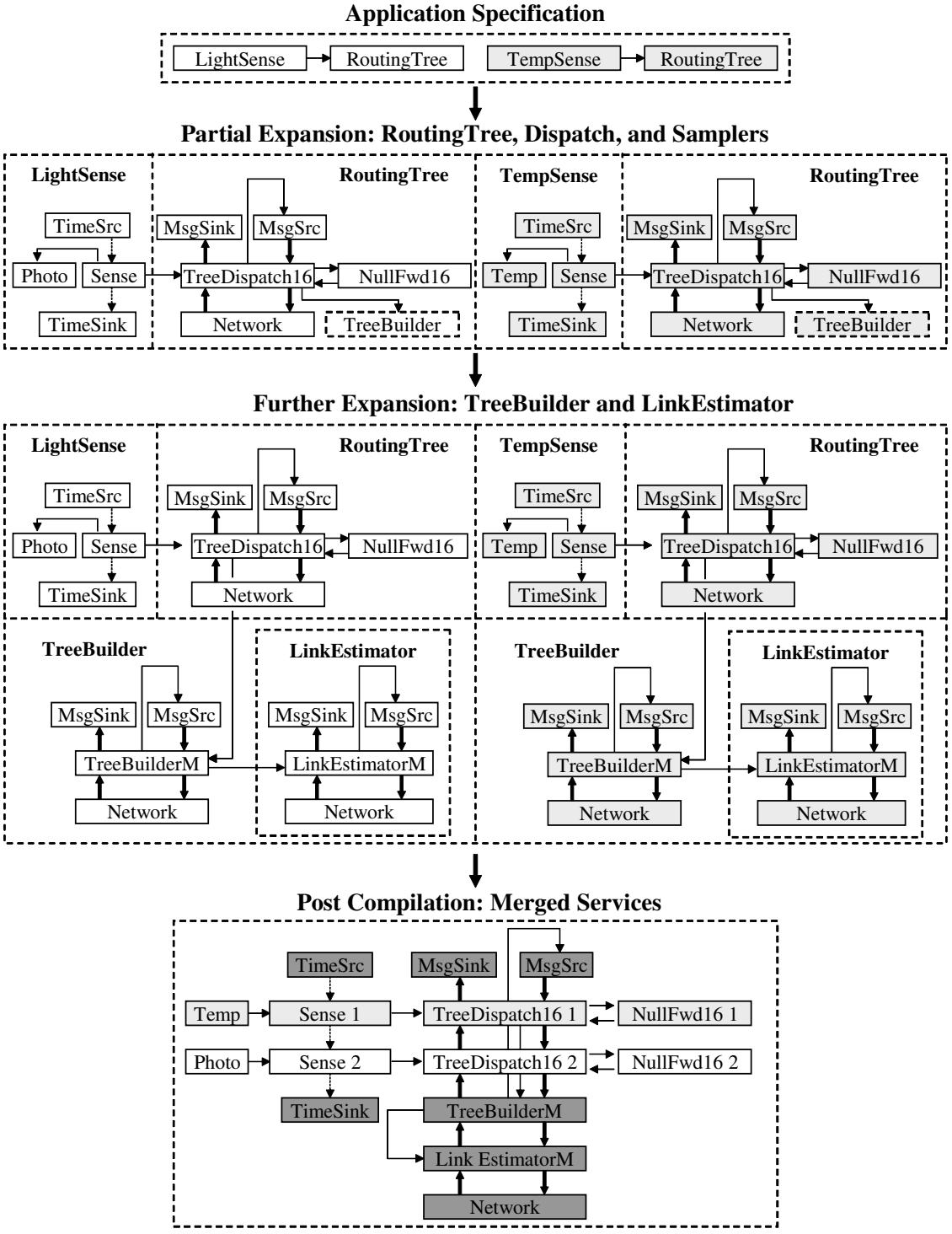

Figure 1—Expansions of a SNACK multihop data collection application (SNACK Forwarder, or SF). At the top, the application is specified by connecting a few application-level services. The middle stages expand those services into their components. The bottom is a subset of the maximally-shared expansion produced by the SNACK compiler. White components are derived from the light-sampling part of the application, light-grey components from the temperature-sampling half; dark-grey components are the result of the compiler merging compatible components from both halves into single instances. Dotted and thick arrows are transitive connections (Section 2.6) for timers and message processing, respectively. Figure 2 shows the SNACK code corresponding to this figure; Section 5 evaluates that code.

2 LANGUAGE DESIGN

Again, our goal is a *smart service library* of relatively easy-to-write services that combine automatically into efficient sensor network applications. We imagine three types of programmers interacting to develop an application. First, *systems programmers* use nesC, or another low-level programming language, to develop reusable *components* that interact with sensor hardware, send messages, handle pools of memory, and so forth. Second, *service programmers* combine those components into parameterized *services*, or component combinations, that implement higher-level semantics such as routing trees, periodic sensing and alerting, and so forth. Finally, *application programmers* select a handful of services to run on a given network, and define how those services will interact. Application programming should be simple enough for scientists to do, powerful enough to support a wide range of applications, and generate efficient enough programs to compete with hand-written code.

We first investigated whether nesC alone could reach this goal. The nesC language [6] is the current state of the art in sensor network programming; its component model, based on units [4], is a clear advance over earlier techniques. However, nesC programs are still hard to write, and existing services tend to implement low-level portions of the OS, such as the radio stack. Our analysis of existing nesC programs identified several issues preventing the development of smart application-level service libraries, all of which we addressed in SNACK. The most important are:

- **State.** Flexible control over object state is a key feature of object-oriented application development. Memory space is at a premium in embedded programming, however, making it important to share state between components wherever possible. NesC therefore implements the shared-state endpoint of the spectrum: components of the same type always share state; any private state, such as sensor values, timers, routing tables, and so forth, must be implemented ad hoc (by either defining different component types or using "parameterized interfaces"). This provides efficiency at the expense of considerable programming burden. Straight object-oriented programming, in contrast, defaults to separate state per object; this simplifies some kinds of programming but makes sharing and efficiency harder. SNACK, therefore, implements a new middle point on the spectrum: Component instances can have private state, but the SNACK compiler detects and takes advantage of all possible sharing opportunities.

- **Component parameters.** Say that a sensing service wants to use a routing tree whose internal timers are based on the sensing frequency. Currently, such frequencies are buried in nesC source, and must be set either at compile time (by `#defines` or `enums`) or at run time. SNACK lets components and services define explicit configuration-level parameters, making them easier to understand and reuse. Additionally, SNACK parameters accept value *ranges*, such as $[0, \infty)$ and $[1, 10]$. These flexible constraints facilitate sharing, since components with overlapping parameter ranges may be easier to find than components whose parameter values exactly match.

- **Control flow sharing.** One way to optimize a sensor network application is to drive multiple actions off a single shared trigger. For example, imagine that two sensors want to send data up the same routing tree. If the sensors can each attach their data to a single message—if they can share a single control flow path for message generation—this will reduce the number of messages sent by a factor of two: a huge advantage. To support this, we need selected control flow paths to be augmentable by other services. Unfortunately, nesC and other common component configuration models make this difficult, since every step in the control flow must be explicitly defined. SNACK introduces a new connector, the *transitive arrow* `..>`, designed specifically for control flow sharing, which we call *service weaving*.

These language features are necessary to build smart libraries, but not sufficient. In particular, a smart library can't be constructed solely of existing components, which were designed for nesC, not SNACK. But the SNACK language facilitates new component designs that, although cumbersome or difficult in straight nesC, take advantage of SNACK patterns to make application programming easier and more efficient. We describe such a new component set in Section 3. Together, the language and these new components achieve our goal: smart, high-level libraries that automatically combine into efficient application configurations.

2.1 Syntax

A short example will illustrate SNACK syntax, which is based on that of the Click modular router [8].

```
service Service {
   src :: MsgSrc;
   src [send:MsgRcv] -> filter :: MsgFilter -> [send] Network;
   in [send:MsgRcv] -> filter;
}
```

"*n* `::` *T*" declares an instance named *n* of a component type *T*. An instance is effectively an object of the given type: instances with different names have different private state (unless the SNACK compiler determines that the instances may be shared). This differs from nesC, which has no explicit instances—all components of a given type are shared. However, instances are sufficiently useful in practice that nesC programmers implement them anyway, using tedious ad hoc methods: state arrays or explicit code duplication ("`CRCPacket`", "`RadioCRCPacket`", and "`UARTNoCRCPacket`", for example). SNACK's explicit instances avoid the need for such workarounds, while its compiler recaptures all relevant sharing. A component type can also occur on its own, as with `Network` above; this declares a new anonymous instance of the given type.

"*n*`[`*i*`:`τ`]`" refers to an output interface on component *n* with name *i* and interface type τ, "`[`*i*`:`τ`]`*n*" to an input interface. A component "provides" its input interfaces and "uses" its output interfaces. The interface type can be omitted when declaring a connection; alternatively, if an interface type unambiguously identifies the interface, the name can be omitted. SNACK supports chained connections ("`a -> b -> c ...`"), which are equivalent to multiple connections. The compiler can infer interface declarations in the middle of a chain based on interface declarations at the ends.

Finally, a SNACK *service*, like a nesC configuration, is a collection of component declarations and connections that behaves like a component. A service's input and output interfaces are declared with the `in` and `out` keywords; the components inside a service cannot be accessed from outside except through explicit service interfaces.

2.2 Controlling Sharing

Although the user can declare different component instances with *n* `::` *T* syntax—or, frequently, by using a component type in different, independently-implemented services—the SNACK compiler ultimately decides whether instances will have distinct implementations in the compiled program. Again, its preference is for sharing, since this leads to smaller, more efficient code. For instance,

```
A[MsgRcv] -> Network;  B[MsgRcv] -> Network;
```

is compiled to the equivalent of this:

```
n :: Network;  A[MsgRcv] -> n;  B[MsgRcv] -> n;
```

Sometimes, though, instances should not be shared. For example, components that contain service-internal state shouldn't be shared between services. The SNACK language controls component sharing using three constructs. First, components with incompatible parameter values will not be shared. Second, *connectedness constraints*, which restrict the number of times an interface can be connected, can reduce sharing: if sharing two components would violate either of their connectedness constraints, then the components will not be shared. Third, the "my" keyword explicitly marks components intended for exclusive use. Absent these constraints, the default is to share; this follows nesC practice and encourages efficiency.

2.3 Parameters

SNACK components and services can take lists of named *parameters*, which are values defined at initialization or compile time. Common examples include sensing periods, queue sizes, forwarding table sizes, network diameters, moving-average stability parameters, and so forth. Parameters provide a clean, unified interface for defining and documenting run-time constants, such as timeout values or sampler rates; they also increase modularity, since services can supply values for their components' parameters. (Currently, the application programmer explicitly sets the values of all parameters, even many hidden within services.) For example:

```
s1 :: MsgSrc(period = 10);
s2 :: MsgSrc;  // use default value
```

Services can also take parameters, which are generally passed to their components. A temperature sampling service incorporates a generic ADC sensing component as follows:

```
service SmoothSenseTemp(period: max uint32_t $p = 1000) {
   my sense :: SenseTemp(period = $p);  ...
   // Could also multiply by a constant: 'period = 2*$p'
}
```

In practice, users often want to restrict an initialization parameter to some feasible range, without providing an exact value. SNACK explicitly supports this; for instance, this sampling interval will take a value between 20 and 60 seconds:

```
sense :: SenseTemp(period >= 20, period <= 60);
```

SNACK will choose a value within this range based on a per-parameter objective function, `max` (choose the high end of the range) or `min` (choose the low end).

Loose parameter constraints let users say what they mean, and broaden opportunities for component sharing. Two sampler instances with parameter constraints of "`period <= 60`" and "`period <= 30`" can be shared, for example, since the combination of their parameter constraints has a solution (namely, "`period <= 30`"). SNACK will not share instances with incompatible parameter constraints.

A program's service and component parameter constraints form a simple linear system, which the compiler solves in the obvious way.

2.4 Connectedness Constraints

Connectedness constraints determine the number of times a component interface may be connected. There are four constraints: the default, `@any`—the lack of constraint—means an interface can be connected zero or more times, `@once` means it must be connected exactly once, `@most` means at most once, and `@least` means at least once.[1] Connectedness constraints are assigned on component types—when one designs an interface, one generally knows how many times that interface should be used. The user may also explicitly restrict an individual interface's constraints; for example:

```
t :: TreeBuilderM [Put32 @most] -> ...
```

[1]The authors of nesC have informally proposed to support similar semantics.

Connectedness constraints reduce configuration errors and provide useful documentation, but they also make some instances naturally incompatible for sharing. For example, consider:

```
t1 :: TimeSrc(period = 10) [Timer] -> Sensor1;
t2 :: TimeSrc(period = 10) [Timer] -> Sensor2;
```

Sharing the two `TimeSrc`s will result in two connections from the output `Timer` interface, so they may be shared only if that interface has an `@least` or `@any` constraint.

2.5 Exclusive Instantiation with "my"

A service author may want to explicitly prevent components from being shared, usually because the components represent private state. Suppose we have a simple component called `EWMA` that provides interfaces for accepting new samples and for notifying consumers of a new exponentially-weighted moving average. Although different `EWMA` instances might often be sharable (they'd have the same `alpha` weight parameter, and `EWMA`'s interfaces are naturally at-least-once), `EWMA`'s state contains the current average—a service-specific value—so sharing a `EWMA` component between unrelated services doesn't make sense.

We therefore introduce a type qualifer keyword "`my`" for explicitly marking unshareable components. A component instance `c` marked "`my`" cannot be shared with any other instance `d` in the configuration, *unless* `c` and `d` were both expanded from the same instance inside some service type, and the services from which `c` and `d` were expanded are completely shared. For example:

```
service SmoothedSensor1 {
    Sensor1[Put16] -> my EWMA(alpha = 0.4) -> out;
}
service SmoothedSensor2 {
    Sensor2[Put16] -> my EWMA(alpha = 0.4) -> out;
}
ss1 :: SmoothedSensor1;   ss2 :: SmoothedSensor2;
```

`ss1` and `ss2`'s `EWMA`s are never shared, since `ss1` and `ss2` cannot be completely shared (the `Sensor1` and `Sensor2` have different types). However, in this code:

```
ss1, ss2 :: SmoothedSensor1;
ss1 -> d :: Discard;   ss2 -> d;
```

`ss1` and `ss2` may be combined into a single `SmoothedSensor1` instance, so there will be a single `EWMA` instance in the result. (However, the `EWMA`s would not be shared if either `ss1` or `ss2` were declared "`my`".)

2.6 Service Weaving

The conventional direct arrow connector "`->`" connects two components together over some interface. That is, the function calls made by the component on one side of the arrow are implemented by definitions on the other side of the arrow. Direct arrows position components within their local context. A component is connected to its neighbors, which in turn are connected to their neighbors.

Any deployed application consists of local connections, but local connections are not, in fact, the right abstraction for constructing a reusable library of interdependent services. Consider, for example, two services that want to check sensors at different rates. We might write the services this way, using a common `TimeSrc` component to modularize out the timer interrupt:

```
service TimedSensor1 {
   TimeSrc(period <= 10) [signal] -> Sensor1 -> TimeSink;
}
service TimedSensor2 {
   TimeSrc(period <= 20) [signal] -> Sensor2 -> TimeSink;
}
```

These services have a similar structure due to their similar functionality. But can they be efficiently combined? The optimal combination might piggyback both kinds of sensing off a single timer, sharing that control flow path and its attendant interrupt costs:

```
TimeSrc(period = 10) [signal] -> Sensor1 -> Sensor2 -> TimeSink;
```

This combination, of course, cannot be obtained from any local connection of the two services. Either the services must be written with knowledge of each other, violating modularity, or we need a new, non-local connector.

SNACK therefore introduces the *transitive arrow connector* "`..>`", which connects components *non-locally*. A transitive connection "`a [i] ..> b`" says interface `a[i]` is connected to interface `[i]b` *via some path of i interfaces*. Since the two `Sense` components don't care in which order they appear, our two services could be written using `..>` as follows:

```
service TimedSensor1 {
   TimeSrc(period <= 10) [signal] ..> Sensor1 ..> TimeSink;
}
service TimedSensor2 {
   TimeSrc(period <= 20) [signal] ..> Sensor2 ..> TimeSink;
}
```

When these services are used together in the same application, the SNACK compiler analyzes the transitive arrows and produces a minimal combination that meets all required constraints: namely, the minimal configuration above, or the similar one that switches `Sensor1` and `Sensor2`.

The transitive arrow facility makes it possible to build services that can combine themselves into minimal, excellently performing combinations. We call this *service weaving*. The transitive arrow, and service weaving, naturally gives rise to independently interesting sensor program design patterns: program methodologies that can improve the behavior of sensor applications. We discuss these in the next section.

3 Component and Service Library

The SNACK library of components and services contains components that sense, aggregate, transmit, route, and process data. It provides application programmers with both the high-level and low-level building blocks required for efficient and flexible applications. Many of these components fundamentally differ from the corresponding components in nesC, since SNACK's are designed to leverage language features like service weaving, and therefore to facilitate smart libraries. This section thus examines the components and services in the SNACK library, and shows how SNACK's language features are used in practice.

3.1 Messaging

The radio is a large consumer of power in a sensor node and, consequently, applications must be designed to minimize the total number of bits sent and received. Thus, if two or more components in a sensor node each need to transmit data, the data should be combined into a single packet whenever possible—the messages should be *aggregated*—to reduce per-message overhead (which reaches 14 bytes per message using BMAC and `TOS_Msg` at 100% duty cycle).[2] Message aggregation is unnatural in TinyOS and nesC because components and services generally create and send their messages directly to the radio stack (and because the TinyOS message format discourages aggregation). SNACK, in contrast, extracts message creation into a separate component, which can then be shared. Services *decorate* passing messages with application information in a tag-length-value format; multiple services can decorate the same message. Finally, service messaging paths use transitive arrow connectors, allowing the compiler to weave independently-written services into a single, cooperative, and efficient application messaging path.

[2]A proposed low-power listening scheme [14] increases packet preambles to many times the default payload length, making message aggregation even more important.

The `LinkEstimator` service provides a good example. This service monitors ingress link quality and periodically advertises what it has learned.

```
service LinkEstimator (period: max uint32_t $p = 5000) {
   lqe :: my LinkEstimatorM(..., period = $p);
   lqe [AttrAccess] -> AttrM;   // ...
   Network [inbound] ..> lqe ..> MsgSink;
   MsgSrc (period <= $p) [outbound] ..> lqe ..> Network;
}
```

The `LinkEstimatorM` component[3] processes both inbound messages, to monitor quality, and outbound messages, to decorate them with link quality advertisements. The inbound path starts at `Network` and ends at `MsgSink`; the outbound path starts at `MsgSrc` and ends at `Network`. The messaging architecture uses five core components:

`Network` receives messages from, and sends messages to, the TinyOS radio stack. SNACK messages sent to `Network`'s `outbound` interface are compacted into `TOS_Msg` packets and transmitted. On the receiving side, `NetworkM` expands `TOS_Msg`s into SNACK messages and pass them along an inbound call chain. Components on this chain are designed to extract and process relevant information from an incoming message, then forward it on for more processing.

`MsgSink` ends inbound call chains; its only role is to destroy buffers it receives.

`MsgSrc` periodically generates empty SNACK messages and passes them on via an `outbound` interface. It also provides interfaces to request a single message and to increase its production rate. It should be used by any service that periodically generates and transmits data, since message paths that use `MsgSrc` and transitive arrows can often be combined.

`AttrM` provides a library of functions to add, read, iterate over, and delete attributes, so that components that use it do not need to manipulate SNACK messages directly. In practice all components that access the contents of SNACK messages do so via `AttrM`'s `AttrAccess` interface.

`MemoryPool` dynamically allocates buffers from a managed pool, and is used by the `MsgSrc`, `Network`, and `MsgSink` services. It allocates blocks in 4 byte chunks, uses first-fit selection, and has a statically-configurable size up to 1020 bytes. Dynamic message memory avoids the high static memory overhead of conservative static allocation, and avoids buffer swapping errors. Mote programmers have been hesitant to use dynamic memory because it is assumed to be difficult to debug leaks, which can be fatal to a deployed application. SNACK components employ a simple rule to diminish the probability of creating a leak: a component that allocates memory or receives a pointer to allocated memory must either deallocate that memory or pass the pointer to another component. Furthermore, we restrict pool allocation to very few types of data, namely attributes and application-level messages, and require all components that pass these types of data to use the pool.

We designed our message path this way to enhance sharing among application services. Since packet creation, access, and transmission are placed in separate components, services can share messaging control flow; and when, as intended, those components are connected by transitive arrows, the compiler can automatically combine independently-specified paths into efficient combinations. This contrasts with traditional component composition languages, whose encapsulation semantics would prevent this combination unless the services were explicitly connected.

3.2 Timing

This pattern—a source, an open-ended set of forwarders, and a sink, all connected by transitive arrows—turns out to be useful for

[3]Base components often have an "`M`" suffix, to avoid confusion with services of the same name.

timer events as well as message generation. In particular, writing timer events in this style can reduce the number of independent timer paths in the system, which in turn reduces phase effects (meaning related timers with slightly different periods) and the number of pending tasks.

The SNACK timing system has two core components, analogous to `MsgSrc` and `MsgSink`: **`TimeSrc`** generates a timestamp signal, emitted over its `signal` interface at a specified minimum rate, and **`TimeSink`** consumes that signal. Application components provide both input and output `signal` interfaces. When the input `signal` interface triggers, the application component performs its task, then forwards the trigger downstream using the output `signal` interface.

Inside services, then, the usual timing paths look like those in `TimedSensor1` and `TimedSensor2` (Section 2.6, above): `TimeSrc [signal] ..>` *c* `..> TimeSink`, where *c* is an application component. Services supply `TimeSrc`'s `period` parameter with a range, rather than a specific value, to facilitate sharing; application components must be written to correctly handle any period in that range (perhaps by ignoring excess signals). When transitive connections are used, the result is the same as in the messaging case: independently-written services combine into shared timer paths—another type of shared control flow.

3.3 Storage

Separate components implement data storage in SNACK. `NodeStore64M`, for example, implements an associative array of eight-byte values keyed by node ID; other components implement arrays of 2-byte or arbitrarily-sized values. Access functions can find, add, delete, modify, and iterate over values. This generic data structure is used in many contexts, including path and link estimation. Since it implements private state, `NodeStore64M` should not generally be shared between services; it is therefore generally marked with "`my`".

Without instances, such a storage component would have to be written as a monolithic component that resorts to intricate arrays of private state and manual port mapping for access to that state. Such a monolith's implementation would be more complicated than our `NodeStore64M`, not least because it would have cope with the issues involved in partitioning its storage space into appropriately sized chunks for its clients.

3.4 Services

The SNACK service library contains a variety of services—combinations of primitive components—ranging in size, configuration complexity, and substructure depth.

The simplest services just wrap single components, connecting "infrastructure" interfaces (such as `StdControl`) to the required component (`Main` in this case). This makes the components easier to use and configurations easier to read. At the other end of the spectrum is `RoutingTree`, which incorporates subservices within subservices. `RoutingTree` implements a tree designed to send data up to some root. Concretely, it wires a `TreeDispatch16` to a `TreeBuilder` and exports `TreeDispatch16`'s transport interfaces:

```
dispatch :: my TreeDispatch16;
in [collect] -> dispatch -> out;
dispatch [lookup] -> TreeBuilder(period = $p);   // ...
```

The `TreeDispatch16` uses a `TreeBuilder` service to provide the ID of the best next hop towards the root of the tree. `TreeBuilder`, in turn, computes and maintains a list of the path qualities to a tree root via each of a node's communication neighbors. It uses a `LinkEstimator` to get the first hop probability, accessing it through the `lqeAccess` interface. Moreover, since the `TreeBuilder` periodically advertises soft state about its best known path to the root, it contains inbound and outbound messaging paths:

```
tree :: TreeBuilderM(period = $p);
lqe :: LinkEstimator(...);

src :: MsgSrc(period <= $p) [outbound] ..> tree ..> Network;
Network [inbound] ..> tree ..> MsgSink;

tree [access] -> my NodeStore64;
tree [lqeAccess] -> lqe;   //  ...
```

The compiler will, of course, combine these messaging paths with those of `LinkEstimator`, among others. Only those components that should not be shared (such as `TreeBuilder`'s `NodeStore64`) are declared "`my`", so that the compiler can reduce code bloat by maximizing sharing.

Figure 2 shows how this comes together into a real application. The figure presents the complete definition of SNACK Forwarder, or SF, a simple multihop data collector that samples light and temperature every 10 seconds and forwards the results along a distribution tree. We evaluate SF's resource allocation and bytes transmitted in Section 5.

3.5 Discussion

Despite the benefits of a component composition language like SNACK, writing components still requires expertise in interrupt-driven nesC programming, and hence the learning curve is still steep. A component developer must learn a component development language; she must understand concurrency and how to use tasks, guards, and atomic sections to safely trap interrupts and synchronize asynchronously executing sections of code; and she must be experienced with implementing distributed protocols that work even when data is lost. On top of all of this, she must learn a configuration language to compose components, and must learn the capabilities of the available components and services. Our work on SNACK does not address many of these problems. On the other hand, this learning curve is precisely the reason we believe a service composition language and library can ease application development. Since it is so difficult to develop functional and efficient components, we employ every effort in SNACK to make such components reuable and easy to configure when reused.

SNACK aims to enable the construction of services easy enough for scientists to use, but we do not believe that all high-level sensor network programming will take place exclusively in the SNACK language: higher-level mechanisms, either graphical or script-based, are natural and probably inevitable. However, our experience with earlier component languages indicates that SNACK is more suitable than straight nesC as a target language for such higher-level mechanisms: it facilitates a natural division of labor where higher-level mechanisms choose services, and SNACK figures out how to efficiently combine them.

4 COMPILER

The SNACK compiler's job is to parse a SNACK-language input file, such as this one:

```
service SA(period: max uint32_t $p) {
   c :: CA(period = $p) [send] -> out;
}
s :: SA(period = 20) [send] -> net :: Network;
```

and expand all services to produce a simple list of components and connections, such as this:

```
s.c :: CA(period = 20);
net :: Network;
s.c [send] -> net;
```

A back end then translates this list into a nesC application, by rewriting nesC component source code (to define parameter values, such as `CA`'s `period`, and create multiple instances when necessary) and generating a configuration.

```
service Sense(period: uint16_t $p = 5000){
  sense:: my SenseM(period = $p);

  Main [StdControl] -> sense -> out;
  sense [ADC] -> out;
  sense [NodePut16] -> out;

  sense [Timer @once] -> Timer;
  TimeSrc(period <= $p) [signal] ..> sense ..> TimeSinkM;
};

service SenseLight (period: max uint32_t $p = 1000) {
  sense :: my Sense(period = $p);
  sensor :: Photo;
  sense [NodePut16] -> out;
  sense [StdControl] -> sensor;
  sense [ADC] -> sensor;
};

service SenseTemp (period: max uint32_t $p = 1000) {
  sense :: my Sense(period = $p);
  sensor :: Temp;
  sense [NodePut16] -> out;
  sense [StdControl] -> sensor;
  sense [ADC] -> sensor;
};

service Network (sendQ: min uint8_t $q = 4,
                 recvQ: min uint8_t $r = 3){
  net :: NetworkM(sendQ = $q, recvQ = $r);
  am :: GenericCommPromiscuous;
  qs :: QueuedSend;

  in [outbound:MsgRcv] -> net;
  net [inbound] -> out;

  Main [StdControl] -> net -> am;
  net [StdControl] -> qs;
  net [toAM:SendMsg] -> qs;
  net [fromAM:ReceiveMsg] -> am;
  net [Memory] -> MemoryPool;
  net [AttrAccess] -> AttrM;
};

service LinkEstimator (alpha: min uint8_t $a = 32,
                       period: max uint32_t $p = 5000,
                       thresh: min uint8_t $q = 200) {
  lqe :: my LinkEstimatorM(alpha = $a, period = $p, thresh = $q);
  ns :: my NodeStore64;

  Main [StdControl] -> lqe;
  in [decreasePeriod] -> lqe;
  in [Access64] -> lqe -> ns;
  in [Toggle32] -> [decreasePeriod] lqe;

  MsgSrc(period <= $p) [outbound] ..> lqe ..> Network;
  Network [inbound] ..> lqe ..> MsgSink;
  lqe [AttrAccess] -> AttrM;
  lqe [Access64] -> ns;
};
```

```
service TreeBuilder (period: max uint32_t $p = 5000) {
  tree :: TreeBuilderM(period = $p);
  lqe :: LinkEstimator(alpha = 64, period <= $p, thresh >= 96);

  Main [StdControl] -> tree;
  in [lookup:Get16] -> tree;

  src :: MsgSrc(period <= $p) [outbound] ..> tree ..> Network;
  Network [inbound] ..> tree ..> MsgSink;

  tree [Put32] -> src;
  tree [AttrAccess] -> AttrM;
  tree [access] -> my NodeStore64;
  tree [lqeAccess] -> lqe;
};

service TreeDispatch16(qsize: min uint8_t $q = 8){
  td :: my TreeDispatch16M(qsize = $q);
  src :: MsgSrc(period > 0);

  Main [StdControl] ->td;
  in [collect] -> td;
  in [broadcast] -> td;
  td [collect] -> out;
  td [subtreeReady] -> out;
  td [fromChild] -> out;
  in [toRoot] -> td;
  td [broadcast] -> out;
  td [lookup] -> out;

  td [newMessage] -> src;
  td [AttrAccess] -> AttrM;

  src [outbound] ..> td ..> Network;
  Network [inbound] ..> td ..> MsgSink;
};

service RoutingTree (period: max uint32_t $p = 5000) {
  dispatch :: my TreeDispatch16;
  nf :: my NullForwarder16;

  in [collect] -> dispatch -> [collect] out;
  dispatch [fromChild] -> nf;
  nf [toRoot] ->  dispatch;
  dispatch [subtreeReady] -> nf;
  dispatch [lookup] -> TreeBuilder(period = $p);
};



tree1, tree2 :: my RoutingTree(period < 20000);
SenseLight(period < 10000) -> [collect] tree1;
SenseTemp(period < 10000) -> [collect] tree2;
```

Figure 2—The service configuration for SNACK Forwarder, or SF, a SNACK application that forwards temperature and light data up a multi-hop routing tree. Figure 1 presents a diagrammatic overview; Section 5 evaluates its performance.

Much of the compiler's work is more or less conventional: parsing the input language, checking static semantics, and expanding services into their components (which resembles macro expansion). But one aspect of its work is both unconventional and difficult—namely, introducing sharing. Most of this section describes the exhaustive search algorithm by which this is done.

4.1 Static semantics checking

The SNACK compiler checks for obvious errors—for example, unknown component types, multiple declarations of the same name, missing parameters, connections that join interfaces with different types, and so forth. It also checks for less obvious errors, including:

- A service interface named `x` may be used in a transitive connection only if, inside the service definition, all interfaces to which the service interface is connected are also named `x`. This ensures that transitive connections in the expanded component graph will connect interfaces with the same name.

- Interface constraints must be satisfied. That is, every `@least` interface is connected at least once, every `@most` interface is connected at most once, and every `@once` interface is connected exactly once.

However, an `@most` or `@once` interface may be associated with multiple transitive connections; the compiler will resolve those transitive connections later.

- The interface constraint on a service interface must equal the intersection of the internal interface constraints to which that service interface is connected. For example, consider this service:

```
service A {
   in [x @once] -> [x @any] C;
   in [y @once] -> [y @most] D;
   in [y] -> [y @least] E;
}
```

  The `@once` constraint on `A`'s `x` input interface is illegal, since it is more restrictive than the `@any` constraint to which `x` is connected. But the `@once` constraint on `y` is correct, since that is the intersection of `[y]D`'s `@most` constraint and `[y]E`'s `@least` constraint. If the user supplies no constraint, SNACK derives the correct constraint automatically.

- Parameter constraints must be solvable. For example, `C(period = 10, period > 20)` is illegal.

These constraints are easy to check, so the compiler can quickly report errors for many invalid configurations, without going through the expensive exhaustive search described in the next section. Some of the checks, such as the restriction on transitive connections' interface names, also simplify the exhaustive search itself. Each check is applied to every service, as well as to the top-level configuration.

4.2 Introducing sharing

Each SNACK program has a *trivial expansion* in which all service components have been replaced by the corresponding configurations, leaving only primitive components. For example, this program:

```
service S {
   a :: A [send @once] -> out;
}
s1 :: S [send] -> Network;
s2 :: S [send] -> Network;
```

has the following trivial expansion (note that the anonymous `Network` components have been given names):

```
s1.a, s2.a :: A;  Network@2, Network@4 :: Network;
s1.a [send] -> Network@2;
s2.a [send] -> Network@4;
```

Instances are never shared in a trivial expansion. In contrast, a *maximally-shared expansion* shares instances whenever possible. A maximally-shared expansion is an expansion of the input program, in that every input component corresponds to exactly one expansion component, every input connection corresponds to an expansion connection, and there are no components or connections not justified by the input program; but additionally, a maximally-shared expansion has the minimum number of instances of any valid expansion. In the program above, `s1` and `s2` can be shared, as can the two `Networks`, leading to a maximally-shared expansion with half the components of the trivial expansion:

```
s1.a :: A;   /* also stands for s2.a */
Network@2 :: Network;  /* also stands for Network@4 */
s1.a [send] -> Network@2;
```

The contrast only grows in real applications. Figure 2's trivial expansion has 62 `Mains` and 166 components in all, while its maximally-shared expansion has just 23 components.

The compiler's goal, then, is to produce a maximally-shared expansion for its input program. This is important because reducing the number of instances decreases program size, reduces the amount of statically allocated RAM, and maximizes shared infrastructure, effectively reducing timer interrupts, sensor hardware interrupts, and the aggregate number of packets that must be delivered over the radio and serial interfaces.

The compiler's strategy is, simply, *exhaustive search*. It traverses a list of all possible expansions, sorted in increasing order by number of instances, and returns the first valid expansion it finds. Clearly this will be a maximally-shared expansion.

An expansion is valid if it passes four tests:

- **Interface constraints.** No `@most` or `@once` interface may be directly connected more than once. (Constraints involving transitive connections are checked separately.) There is no need to check whether `@least` constraints are satisfied, or whether `@once` interfaces are connected, since those constraints were verified during static semantics checking and are satisfied by every expansion.

- **Parameters.** The linear system of all component and service parameters has a solution.

- **"my" constraints.** "`my`" constraints are satisfied, meaning that a "`my`" component c may be shared with another component d only if c and d are *my-shareable*. This, in turn, means that c and d have the same parent service type, they were expanded from the same instance within that parent service type, the parent services are completely shared, and either the parent services were not marked "`my`" or the parent services are recursively `my`-shareable.

- **Transitive connections.** A valid assignment of transitive connections exists. To find this assignment, the compiler groups the transitive connections by interface name and type; different groups may be assigned independently. Then for each group, it tries all $n!$ connection permutations, stopping at the first valid one. To check a permutation, the compiler tries sequentially to change each transitive connection into a valid direct connection.

  For example, consider the following configuration, based on the example in Section 2.6:

```
ts [signal] ..> s1 ..> te;  ts [signal] ..> s2 ..> te;
```

  All `signal` interfaces have `@once` constraints. Then the relevant group contains all four connections. Some permutations will fail, such as `ts..>s1`, `s1..>te`, `ts..>s2`, `s2..>te`: the first two steps create the configuration fragment `ts->s1->te`, to which no further connections can be added because of `@once` constraints. However, the order `ts..>s1`, `ts..>s2`, `s1..>te`, `s2..>te` will work, producing the configuration `ts->s1->s2->te`.

The exhaustive search space is huge. Consider a program with k distinct component types $T_1, \ldots, T_k$, where component type T_i has n_i distinct instances in the program's trivial expansion. An expansion, with respect to a given type T_i, consists of a partitioning of the n_i instances into nonempty subsets. In the best case, all of T_i's instances are shared, and the n_i instances are all in one set; in the worst case (the trivial expansion), none of T_i's instances can be shared, and the n_i instances are in n_i distinct singleton sets. The number of ways n_i elements can be partitioned into nonempty subsets is called a Bell number, and denoted B_{n_i} [18]. Bell numbers grow quite quickly:

$$B_m = \frac{1}{e}\sum_{i=0}^{\infty}\frac{i^m}{i!},$$

and $B_{20} \approx 5.2 \times 10^{13}$. The entire exhaustive search space for our program contains $N = \prod_{i=1}^{k} B_{n_i}$ expansions. For the configuration of Figure 2, N is approximately 4×10^{113} (!).

To reduce this search space to a manageable size, the compiler eliminates obviously incorrect expansions from consideration. In particular:

- The compiler maintains, for each component type, a list of pairs of instances that can never be shared. Two components c and d can never be shared if:

  - There is no joint solution to the parameter constraints of c and d (for instance, c.`period` $= 20$ and d.`period` > 30);
  - There is some interface i where at least one of $c[i]$ and $d[i]$ has an `@most` or `@once` constraint, and $c[i]$ and $d[i]$ are directly connected to components with different types (for instance, $c[i]$ `-> A` and $d[i]$ `-> B`: the compiler will never combine `A` and `B` into a single instance, so there would always be at least two direct connections leading from any combination of c and d, violating the interface constraint);
  - Either c or d is declared "`my`", and c and d are not `my`-shareable.

  Expansions that violate these "problem pairs" are eliminated at the component type level, without checking other component types' expansions. Thus, a single "problem pair" on component type T_j reduces the search space by a factor of $\prod_{i \neq j} B_{n_i}$.

- The compiler additionally memoizes, for each component type, the smallest expansion subset that contains no "problem pairs", avoiding repetitive work.

- When an expansion is invalid because of some problem, the compiler advances in one step to the next expansion that might affect the problem, rather than wasting time on expansions that would definitely have the same problem. It does this by making sure to change at least one component type involved in the problem. In particular:

  - If a component interface $c[i]$ was connected too many times, then the relevant types consist of c and the components connected to $c[i]$.
  - If a set of transitive connections could not be assigned, then the relevant types are those with an input and/or output interface of the relevant name and type.
  - If a "`my`" constraint was violated, then the relevant types consist of the type of the base component that violated the constraint, and the types of those components contained in the enclosing services that have different expansions.

  For future work, we will tighten these constraints further, and add a similar constraint for parameter problems.

The combination of these heuristics is quite effective for our applications. For example, the compiler finds the maximally-shared expansion of Figure 2 in less than a half-second on a recent Intel-based laptop, after testing just 28 partial expansions. Without heuristics, an exhaustive search would have tested more than 10^{36} expansions before encountering the right one.

4.3 Discussion

The compiler currently works quickly on configurations whose maximally-shared expansion is small, such as all the configurations we describe here. However, a configuration whose maximally-shared expansion is large—where most components cannot be shared—can cause the compiler to run for a long time. For future work, we will speed up the compiler in these cases.

Application	ROM	RAM
SF	**18,806**	**1,867**
SF-NoTransitive	24,096	3,077
Surge	**14,564**	**1,921**

Figure 3—Application resource allocation in bytes. SF uses 30% more ROM than Surge, but 3% less RAM; both values for both SF versions are well within the mica2's operating constraints.

Description	Component	ROM	RAM
Attribute Access	`AttrM`	2,038	0
Memory	`MemoryPoolM`	2,758	302
Link Quality	`LinkEstimatorM`	1,306	51
Communication	`NetworkM`	958	319
Periodic Messages	`MsgSrcM`	788	48
Tree Construction	`TreeBuilderM`	852	19
Tree Transport	`TreeDispatch16M`	280	36

Figure 4—Resource allocation in bytes for SF's largest components, which together account for 44% of SF's ROM and 42% of its RAM.

5 SAMPLE APPLICATIONS

This section shows how SNACK applications behave in practice. We compare two versions of a simple SNACK data collector with Surge, a similar nesC application, in terms of both memory usage and bytes transmitted in simulation. We also describe a query-and-collect application that uses SNACK principles to decompose the query engine, and compare it with the original monolithic query system. Our performance hypotheses are that SNACK applications are not worse than their predecessors—that they use roughly the same memory space, and transmit no more data—and that services using SNACK features, especially message aggregation via transitive connections, transmit less data than services not using those features.

5.1 Multihop Data Collection

The SNACK Forwarder application, or SF, produces temperature and light data every 10 seconds and forwards this data up a multihop routing tree to a root. Figure 1 shows its high-level structure; Figure 2 shows its code. Tree construction control messages and link quality estimator beacons are sent at least once every 20 seconds. Advertisements of ingress link quality estimates are sent every 160 seconds. Although simple, this application structure has been used in many real deployments. GDI, for example, takes weather board sensor data and forwards it up a tree [12].

For a comparison point, we use the Surge application, a prototypical implementation of multihop data collection distributed with TinyOS. Surge as distributed can handle one type of sensor value. Crossbow has contributed several variations of Surge, including one that supports multiple sensor types. We use this version of Surge with two sensor types and the same sampling rates as SF. When we ran our experiments, Surge used the MintRoute multi-hop routing service [21], an updated version of the code used in the GDI deployment. We configure MintRoute to generate the same rate of control messages as SF.

We also evaluate SF-NoTransitive, a version of SNACK Forwarder without transitive arrows. SF-NoTransitive's services cannot be woven together into combined call chains; we thus expect it to transmit more data, since it cannot aggregate multiple data sources into single messages. When compiled, SF-NoTransitive has four sources of empty messages, four network components for transmission, and two sources of time to activate sampling.

All code was compiled for a typical sensor platform: mica2s running the standard release of TinyOS 1.1 and a task queue size of 16. Experiments used the TOSSIM simulator [9].

Application	Pool	TOS_Msg
SF	**172**	**308**
SF-NoTransitive	188	1,232
Surge	N/A	**836**

Figure 5—Messaging RAM allocation in bytes, including maximum dynamic memory pool usage (during 49-node simulations) and static TOS_Msg allocation.

5.2 Resource Allocation

Figure 3 displays the ROM and RAM footprints of all four applications. The dynamic memory pool used by SF was configured with a maximum size of 300 bytes; SF's RAM footprints include this maximum size. SF-NoTransitive consumes considerably more resources than SF. This is as expected: the lack of transitive arrows leads to three more MsgSrcM and NetworkM components, and one more TimeSrcM component, than SF. Per-component ROM and RAM allocations, partially shown in Figure 5, confirm that these components account for most of the differences in resource allocation. SNACK Forwarder's memory allocations are competitive with those of Surge. Its RAM usage is slightly less than Surge's, and although it uses 4,242 more bytes of ROM (around 30%), this is still nowhere near the mica2's 128 KB maximum.

Figure 4 breaks down the ROM and RAM allocation for SF's largest components. As an approximation, we estimate a component's contribution to ROM by replacing that component with a minimum implementation that provides and uses its interfaces. We find that the code required to provide an application-level message format is considerable. The attribute access component, which is responsible for adding attributes to outgoing messages and extracting attributes from incoming ones, consumes 11% of the application's total ROM; the dynamic memory pool, which supports inter-component message passing and variable application-message sizing, consumes 15%. Fully one-fourth of the allocated ROM is dedicated to these services. Since the nesC compiler makes heavy use of inlining, in more complex applications, the ROM allocation resulting from using these components will likely increase. We are investigating nesC and gcc compilation techniques to address this.

Figure 5 shows how messaging memory is used in practice. The "Pool" column shows, for SF, the maximum number of bytes allocated from the dynamic memory pool at any time in our 49-node simulations. The results are encouraging: even though the dynamic memory pool was used for all incoming and outgoing application-level messages and their attributes, the largest allocation was just 188 bytes, at least in simulation. A maximum pool size of 300 bytes seems clearly sufficient for this application and set of topologies. The "TOS_Msg" column shows the number of RAM bytes allocated for TinyOS messages by each application; SF's NetworkM components use a pool of TOS_Msgs to communicate with the TinyOS radio stack, while Surge uses TOS_Msgs throughout.

5.3 Data Transmission and Aggregation

A large consumer of energy in sensor nodes is the radio. We therefore focus on radio operations to approximate the relative energy consumption of our various applications.

When comparing the performance of applications, we use bytes transmitted as an indicator of relative energy consumed. Since TinyOS 1.1 uses a CSMA MAC with no early termination, which will receive a packet in its entirety even if it was intended to be unicast to another node, there's no difference between unicast and broadcast packets in terms of energy consumption at the receiver. Furthermore, a mote without low-power listening functionality is always listening and in doing so is consuming as much energy (29 mW on the CC1000) as when receiving. The energy to transmit (42 mW at 0 dBm) is more than to receive [16], making bytes transmitted a good indication of energy consumed above and beyond listening or receiving. Alternatively, we might assume motes could switch to a low-power state and perfectly synchronize communication. Then, at least in uniformly random deployments, the number of communication neighbors is roughly constant across all nodes, making the aggregate energy required to transmit a packet and have all neighbors receive it a constant multiple of the energy of transmission alone.

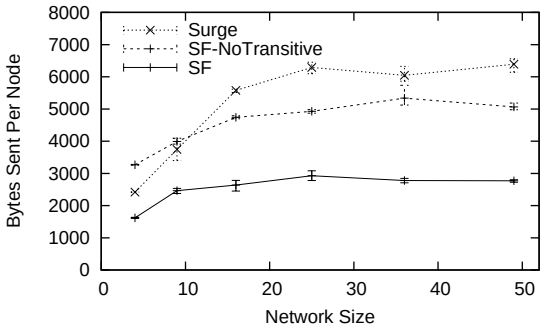

Figure 6—Bytes sent per node as a function of network size for 600-second simulations at varying network sizes. Each data point represents the average of three runs; error bars indicate the minimum and maximum runs.

Application	Packet	Control	Data	Total
SF [4]	876.0	300.3	442.5	1618.8
SF [16]	1240.7	362.5	1034.9	2638.0
SF [49]	1282.8	363.0	1126.3	2772.1
SF-NoTransitive [4]	2330.7	505.1	429.6	3265.3
SF-NoTransitive [16]	2930.3	602.7	1209.2	4742.2
SF-NoTransitive [49]	3096.2	630.9	1336.3	5063.4
Surge [4]	1073.3	1345.7		2419.0
Surge [16]	2491.4	3087.1		5578.5
Surge [49]	2892.1	3497.5		6389.6

Figure 7—Average bytes sent per node due to packet overhead, application control traffic, and sensor data. We do not distinguish control and sensor data for Surge. Network sizes are in brackets.

Since there's no difference in energy consumed by unicast and broadcast packets, our communication services will sometimes place data intended for a particular node and data intended to be broadcast in the same packet. However, to save space in our addressing fields, data destined for two different nodes cannot be aggregated.

The number of bytes sent by SF is determined by the amount of application control traffic (for link quality beacons and tree building), which is roughly linear in the number of nodes; the number of sensor data bytes sent, which depends on the number of nodes and the average depth of the routing tree; and the number of packets used to send this information, which depends on how much aggregation the applications perform. Each packet contains 16 bytes of non-application data: 5 bytes of preamble (assuming BMAC and 100% duty cycle), 2 sync bytes, a 7-byte TOS_Msg header, and a 2-byte SNACK header.

To evaluate SF, SF-NoTransitive, and Surge, we ran several 600-second TOSSIM simulations using topologies and connectivity models derived from measurements of a real mote testbed [2]. Network sizes varied from 4 to 49 nodes; each experiment was run three times. Figure 6 shows the number of bytes sent. Figure 7 breaks down these results into three categories: packet overhead, application control traffic, and sensor data.

SF-NoTransitive sends much more data than SF: for a 49-node network, the difference is roughly 2300 bytes per node. Figure 7 shows that SF and SF-NoTransitive send similar amounts of appli-

cation control traffic and sensor data, and almost 80% of the difference between applications is due to packet overhead. This follows our expectations. The two SFs use the same components, so they should send roughly the same application traffic. However, SF-NoTransitive's four independent message paths, due to its lack of transitive arrows, cause traffic from different services to be sent in different packets. We'd therefore expect SF-NoTransitive to send approximately 3 times more packets than SF. (This is because the two control traffic paths operate at half the speed of the two data paths.) The true ratio is about 2.5.

Both SF and SF-NoTransitive transmit more application data per node as network size increases. This is because the average routing tree depth increases with network size. Routing tree shape has a strong effect on the number of bytes sent per node, and seems quite sensitive to messaging characteristics of the applications under study; for example, subtracting light sensors from SF—and thereby reducing the data being generated—also reduces routing tree depth.

Our performance goal was for SF to perform roughly comparably to Surge, despite the advanced character of SF's services. SF achieves this goal, and in fact transmits less data than Surge. Possible reasons include different routing trees and, more fundamentally, Surge's lack of aggregation.

5.4 A SNACK Query Engine

We also implemented a more complex application to verify that SNACK strategies remain useful in other contexts. In particular, we refactored a monolithic query engine designed for habitat monitoring data collection into a set of SNACK services. The SNACK query engine has one service per query type supported by the original query engine; a `PeriodicQuery`, for example, samples a specified sensing modality at a requested rate:

```
service PeriodicQuery() {
   query :: PeriodicQueryM;  table :: QueryTable;
   datamap :: DataMap;   conn :: QueryConnector;

   query [QueryTableI] -> table;   query [MeasName] -> datamap;
   query [LibQueryI] -> LibQueryM;
   conn [QueryI] ..> query;   conn [DataI] ..> query;   // ...
}
```

Other queries include a one-shot query called `SingleQuery`, a query for sensing secondary modalities when a primary modality exceeds a threshold (`PeriodicConditionalQuery`), a query for detecting an event such as a rain bucket tipping (`EventQuery`), a query for reading a named set of sensors when an event occurs (`EventAggregateQuery`), and a query for deleting a running query (`DeleteQuery`). The structure of each query service is identical to the `PeriodicQuery`, except for the underlying query component itself.

Queries use a `QueryTable` to store running queries, a `DataMap` to translate human data requests into sensing channel IDs, and a `QueryConnector` to connect to the rest of the application (in our sample case, a single hop collection application). Like the messaging and timer paths described above, the interfaces for incoming queries (`QueryI`) and incoming data to be resolved by queries (`DataI`) can be specified with a transitive connector, allowing the compiler to construct interface chains if there are multiple query types. When a query service receives a new query or new data, it checks the type, processes the request if relevant, and forwards it along the chain. As in messaging and timrs, query service weaving lets independent services cooperate to form an efficient path; here, that path resembles the functionality of a `switch` statement embedded, unextensibly, in the monolithic query engine.

The following table summarizes the memory allocation for these different query processing implementations.

Query Style	ROM	RAM
SNACK Queries	30,460	1731
Monolithic Query Engine	30,226	1731

Thus, using the SNACK services results in identical functionality to the monolithic style, an identical RAM allocation, and only a slightly worse ROM allocation.

6 Related Work

Other component systems form SNACK's most closely related body of work. In particular, SNACK is currently built on top of nesC [6] and TinyOS [10], the de facto standards for mote programming and operating systems. NesC adds a component abstraction to the C language, primitives for managing interrupt concurrency, and extensive inlining for greater efficiency, and is a great improvement over previous models for constructing sensor applications. SNACK's base components are written in nesC, and its compiler generates nesC source files and a nesC configuration as output. SNACK replaces TinyOS's application-level components with its own library, but makes use of TinyOS's infrastructure components, such as the radio stack and timer interrupt access.

NesC's component composition model descends from units [4], a language designed for flexible linking specifications. Units have been used in other systems, such as Knit [17], a component model for the Flux OSKit [5]. However, units were originally designed to link components together, rather than instantiate their state in an object-oriented sense. This may be one source of nesC's inability to multiply instantiate components, which simplifies optimization but complicates programming.

More object-oriented component composition languages, such as the network stack systems Click [8], Scout [13], and *x*-kernel [7], remove this instantiation restriction. SNACK's syntax most closely resembles that of Click, with additions for interface names, types, and constraints and the transitive arrow operator.

A complementary approach to ours is to provide high-level access to particular sensor network application operations. The Hood neighborhood abstraction for sensor networks [20], for example, uses a library of perl scripts and nesC code to expand "macrodeclarations" describing neighborhood and data types shared into configurations of nesC components. Similarly, abstract regions [19] are a family of spatial operators designed to let a nesC programmer think in terms of the spatial relationships of nodes when sharing data. SNACK and these approaches should be complementary: SNACK provides general mechanisms for service construction, while Hood and abstract regions demonstrate ways to construct particularly important services.

Another complementary approach is the construction of whole applications that provide high-level interfaces to network users. For example, TinyDB [11] provides SQL-style query support for networks of motes. In this sense, it is a kit for constructing a specific type of application—one that queries. The benefit to this approach is simplicity, but its drawback is flexibility: TinyDB, unlike SNACK, cannot be used to build applications in which the nature of data production, filtering, aggregation, and routing can be arbitrarily chosen. We believe that as our library expands to a sufficient number of query and data collection services, SNACK can be used to easily and efficiently construct and extend applications like TinyDB.

7 Future Work

SNACK is ongoing work. Although we have written dozens of components that comprise the services in our library, more need to be written. In particular, we will focus on data filters and aggregators, and mechanism for triggered queries and actuation.

The compiler currently generates a single binary that is expected to be run on a set of motes. We will extend our service library and language so that mote *roles* can be specified and their interrelationships described. A single SNACK specification thus might be converted by our compiler into multiple binaries, one for each role. Ultimately, in addition to creating multiple binaries for a single platform, we would like to investigate to what extent our library can be extended to cross-platform tiered-architecture applications.

We expect syntax additions. We will be adding syntax for using and overriding default components and services, and for passing components or groups of components as parameters to services. We plan additions to type qualifiers (for instance, "singleton") to restrict component usage contexts. In the longer term, we wish to extend our parameter syntax for static initialization choices to dynamically control and optimize runtime behavior.

We will also optimize our compiler's performance by improving its exhaustive search heuristics.

8 Conclusion

We have presented SNACK, a configuration language, component and service library, and compler that enables application developers to harness the power of service abstraction without losing control over efficiency. Through facilities such as controlled sharing and transitive connections, SNACK enables the construction of smart, easy-to-use application-level libraries that can automatically weave themselves into efficient combinations. We constructed a data forwarding application in SNACK and have shown that its performance in terms of radio usage and footprint is comparable or better than an existing application in nesC alone. Although much work still needs to be done, we have made significant progress towards our goal: providing a way for application programmers to easily develop efficient sensor network applications. SNACK will be made freely available online.

Acknowledgments

This work was made possible by the NSF Center for Embedded Networked Sensing (CENS) under contract number CCR-0120778. Future work on SNACK will be funded by the National Science Foundation under award number 0435497. We would like to thank Ning Xu for evaluating our code, the Extensible Sensing System design team including Tom Schoellhammer, Eric Osterweil and Mohammad Rahimi for contributing query engine code, Matt Welsh for encouraging us to use nesC as our underlying component description language, and our anonymous reviewers for their valuable feedback.

References

[1] A.A. Abidi, G.J. Pottie, and W.J. Kaiser. Power-conscious design of wireless circuits and systems. *Proceedings of the IEEE*, 88(10):1528–45, October 2000.

[2] A. Cerpa, N. Busek, and D. Estrin. SCALE: A tool for simple connectivity assessment in lossy environments. CENS Technical Report 0021, Center for Embedded Network Sensing, UCLA, September 2003.

[3] Alberto Cerpa, Jeremy Elson, Deborah Estrin, Lewis Girod, Michael Hamilton, and Jerry Zhao. Habitat monitoring: Application driver for wireless communications technology. In *Proc. 2001 ACM SIGCOMM Workshop on Data Communications in Latin America and the Caribbean*, April 2001.

[4] Matthew Flatt and Matthias Felleisen. Units: Cool modules for HOT languages. In *Proc. ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI)*, pages 236–248, Montréal, Canada, June 1998.

[5] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The Flux OSKit: A substrate for kernel and language research. In *Proc. 16th ACM Symposium on Operating Systems Principles*, pages 38–51, Saint-Malô, France, October 1997.

[6] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesC language: A holistic approach to networked embedded systems. In *Proc. ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI)*, pages 1–11, San Diego, California, June 2003.

[7] N. C. Hutchinson and L. L. Peterson. The *x*-kernel: an architecture for implementing network protocols. *ACM Transactions on Software Engineering*, 17(1):64–76, January 1991.

[8] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[9] P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: Accurate and scalable simulation of entire TinyOS applications. In *Proc. First ACM Conference on Embedded Networked Sensor Systems (SenSys 2003)*, 2003.

[10] Philip Levis, Sam Madden, David Gay, Joseph Polastre, Robert Szewczyk, Alec Woo, Eric Brewer, and David Culler. The emergence of networking abstractions and techniques in TinyOS. In *Proc. 1st Symposium on Networked Systems Design and Implementation (NSDI '04)*, pages 1–14, San Francisco, California, March 2004.

[11] Samuel Madden, Michael Franklin, Joseph Hellerstein, and Wei Hong. TAG: a tiny aggregation service for ad-hoc sensor networks. In *Proc. 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, Massachusetts, December 2002.

[12] Alan Mainwaring, Joseph Polastre, Robert Szewczyk, David Culler, and John Anderson. Wireless sensor networks for habitat monitoring. In *ACM International Workshop on Wireless Sensor Networks and Applications*, Atlanta, GA, September 2002.

[13] David Mosberger and Larry L. Peterson. Making paths explicit in the Scout operating system. In *Proc. 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI '96)*, pages 153–167, Seattle, Washington, October 1996.

[14] Joseph Polastre, Jason Hill, and David Culler. Versatile low power media access for wireless sensor networks. In *Proc. Second ACM Conference on Embedded Networked Sensor Systems (SenSys 2004)*, 2004.

[15] G.J. Pottie and W.J. Kaiser. Wireless integrated network sensors. *Communications of the ACM*, 43(5):51–58, May 2000.

[16] V. Raghunathan, C. Schurgers, S. Park, and M. Srivastava. Energy aware wireless microsensor networks. *IEEE Signal Processing Magazine*, 19(2):40–50, March 2002.

[17] Alastair Reid, Matthew Flatt, Leigh Stoller, Jay Lepreau, and Eric Eide. Knit: Component composition for systems software. In *Proc. 4th Symposium on Operating Systems Design and Implementation (OSDI 2000)*, San Diego, California, October 2000.

[18] Eric W. Weisstein. Bell number. From *MathWorld—A Wolfram Web Resource*. `http://mathworld.wolfram.com/BellNumber.html`.

[19] Matt Welsh and Geoff Mainland. Programming sensor networks using abstract regions. In *Proc. 1st Symposium on Networked Systems Design and Implementation (NSDI '04)*, San Francisco, California, March 2004.

[20] Kamin Whitehouse, Cory Sharp, Eric Brewer, and David Culler. Hood: a neighborhood abstraction for sensor networks. In *Proc. 2nd International Conference on Mobile Systems, Applications, and Services*, pages 99–110. ACM Press, 2004. ISBN 1-58113-793-1.

[21] A. Woo, T. Tong, and D. Culler. Taming the underlying challenges of reliable multihop routing in sensor networks. In *Proc. First ACM Conference on Embedded Networked Sensor Systems (SenSys 2003)*, 2003.