Data Collection and Beyond for Sensor Networks

Farley Lai
Department of Computer Science
University of Iowa

Introduction

In recent years, wireless sensor network (WSN) researchers made significant advancements in managing low-level hardware resources using techniques such as low power listening, time synchronization, or efficient data collection protocols. However, in spite of this progress, WSNs remain difficult to program because existing programming paradigms such as NesC [4] and TinyOS [5] focus on low-level resource management rather than providing programmers with the higher-level abstractions necessary to support effective coordination among distributed sensors. This report evaluates that use of dataflows as an alternative programming abstraction that simplifies programming by allowing the user to specify the application behavior at the network level, rather than the level of individual nodes. In this model, an application is defined as multiple dataflows that facilitate in-network processing, data aggregation, and coordination among sensor nodes. As a first step, I will place dataflow programming in the context of other WSN programming abstractions. Then, I will present the design of three systems that build on dataflow abstractions and evaluate them from the perspective of ease of programming and implementation efficiency.

Mottola and Picco [8] proposed a taxonomy of WSN programming abstractions based on their language and architectural features. Figure 1 reproduces the entire taxonomy. However, for the purposes of this report, I will focus on two categories: data access model and communication scope from the language aspect that differentiate different levels of programming abstraction. First, the data access model captures the core idea of a programming abstraction; other design decisions are mainly made to support it. Second, the communication scope not only determines the efficiency of accessing the data model but also the complexities exposed to application developers. For example, if only one-hop communication is supported, a developer has to implement a routing protocol to find the path to send a message to a node multiple hops away,

not to mention the exception handling. In the following, we will go through the three data access models: data sharing, mobile code and database and introduce example systems with different communications scopes. Table 1 lists those programming abstractions for quick reference.

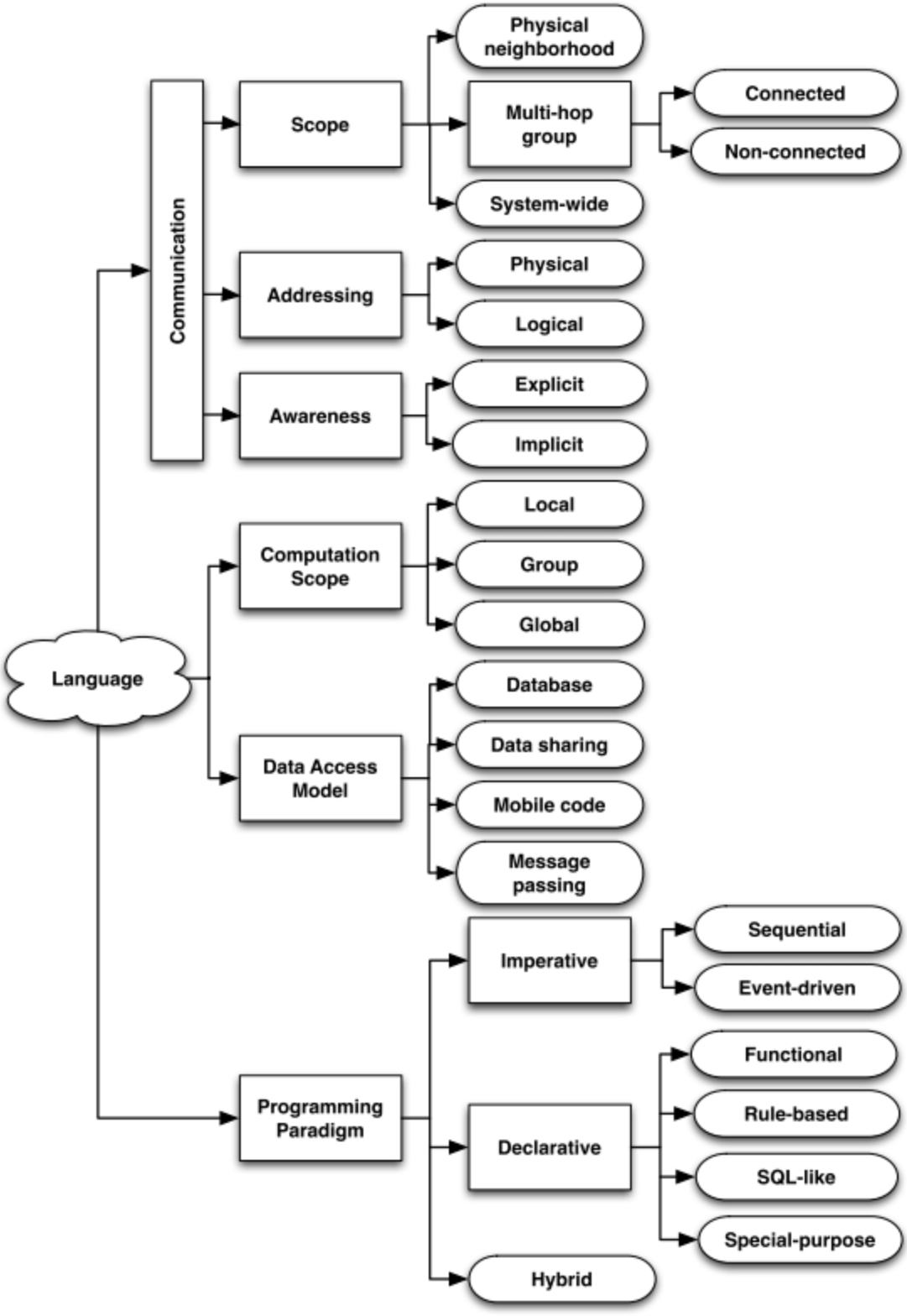

Figure 1. A taxonomy of language aspects in WSN programming abstractions.

Data Sharing: In contrast with TinyOS and NesC that provide primitives for explicit message exchanges, data sharing abstractions permit sensors to read and write shared variables via remotely accessible variables or tuples. However, unlike traditional distributed systems for which data sharing models were originally proposed, data sharing implementations in WSNs must also address the challenges of unreliable communication, minimization of energy consumption, and neighbor discovery (e.g., discovery of sensors with shared properties). These challenges are typically addressed by controlling the scope of communication. In general, this implies a trades-off between the expressiveness of the abstraction and implementation efficiency. However, it is worth noting that for certain applications (e.g., intruder detection and tracking [1]) limiting the scope of communication has minimal impact on the expressiveness of programming abstraction as in such applications only proximate nodes must communicate. Examples of mechanisms for tuning the communication scope include:

- In Hood [12], a neighborhood is defined to span the one-hop nodes that have common attributes. A node can participate in multiple neighborhoods and share different variables within each neighborhood. Hood hides the low-level communication details from the programmer.
- Envirotrack [1] is a programming abstraction that supports target tracking applications by establishing groups surrounding targets that move through a sensor field. The membership of the group changes dynamically to include the sensors that can detect the target. Multihop communication is necessary for collaborative target detection and tracking.
- Abstract Regions [11] defines a region as a set of nodes according to some criteria such as including all nodes within a given distance from each other. Similar to Envirotrack, the multi-hop communication scope is necessary but not every node on the multi-hop path belongs to the group.

Data sharing models are effective in reducing the programming effort by hiding the low-level communication details. However, this model does not scale to support groups consisting of many nodes or that require communication over multiple hops. Additionally, most implementations of neighborhood abstractions such as Hood [12] do not provide integrated

mechanisms to handle communication errors or neighborhood changes (e.g. neighbors leave or join) which may lead to unexpected results. Lastly, the parallelism in network operations is not captured explicitly within the model for optimizations.

Mobile Code: Aside from data sharing model, an alternative way to exchange data is through agents that carry both the code to execute and the execution state. Agilla [3] is a mobile agent system for WSNs where agents can publish data in tuple spaces to be read by (other) agents at A later time. The communication scope depends on the underlying routing protocol.

Database: Treating an entire WSN as a relational database offers an abstraction that allows an entire network to be programmed via simple SQL-like queries (e.g., TinyDB [7]). While this abstraction provides an intuitive mechanism for developing simple WSN applications, its applicability is limited: it is difficult (if not impossible) to program general-purpose WSNs that require in-network processing and coordination within local regions of the network [10].

Programming Abstraction	**Communication Scope**	**Data Access Model**
Hood	Physical neighborhood	data sharing
Envirotrack	Multi-hop group, Connected	data sharing
Abstract Regions	Multi-hop group, Non-connected	data sharing
Agilla	Routing dependent	mobile code
TinyDB	System-wide	database

Table 1. Example programming abstractions and their associated data access models and communication scopes.

Dataflow: An alternative data access model to the previously presented models is dataflow. According to this model, a system is modelled as a graph with components that encapsulate user code at the vertices. Local or remote links connect components. The fundamental advantage of this abstraction is that it provides a good trade-off between flexibility and specificity. On one hand, dataflow can be used to develop software in a modular manner by connecting components in an arbitrary fashion. In this way, the developer's responsibility can be split into component creation and wiring. The former can be taken by people with domain knowledge (e.g. signal

processing) while the latter allows for general people to carry out. This relieves programmers' burden to some extent. On the other hand, dataflow also makes both parallelism and communication explicit, opening significant opportunities for automatic reasoning and optimizations. Due to these important advantages, we choose to focus on the dataflow model and present the state of the art work in this area.

With all the design aspects in mind, we are going to explore a building block design in the next section which is originally employed in network routers. A building block design is a programming approach that can be used in conjunction with other solutions, each targeting independent issues [8]. Though the application context and hardware platform differ from WSN applications, the programming approach is still applicable for WSN applications with respect to the dataflow abstraction.

A Building Block Design: Click

Kohler, Morris and Chen [6] developed Click, a toolkit for developing modular routers. Click uses a dataflow programming abstraction: a router is modeled by linking together components called **Elements** that encapsulate basic router functionality to process packets. Elements have one or more input/output ports. **Connections** are established between ports to forward packets between linked elements. Overall, dataflow programming enables programmers to implement the router's functionality as elements which are connected in directed acyclic graphs to build a modular router.

Execution Model: The Click scheduler determines the order in which elements are executed. In general, an element may be executed when there are packets available to process on any of its input ports. The flow of execution follows directed paths that connect sources elements (e.g., **FromDevice**) to sinks elements (e.g., **ToDevice**) through intermediary elements. Source elements typically **push** packets to the next element via the connection that links its output port to the next element. The scheduler continues to execute a sequence of elements without preemption until either a sink (where a packet is consumed) or a **Queue** element is reached (where a packet may be stored). Upon reaching a queue, the scheduler may execute a different queue or source in the directed graph (or the same queue).

Data pushing is not always an efficient method for exchanging data. For example, a component may be busy and unable to process an incoming push. To handle such cases, Click also supports data **pulling** as follows. A downstream element such as a data sink element may **pull** packets from its previous element via an input port. The pull may be triggered by the underlying hardware device being ready to transmit a packet or completion of a previous operation. The pull operation propagates upstream until a queue element. If the queue is empty, null is returned. Otherwise, a stored packet is returned. As a result, in this execution model, a port can only be either pushed or pulled but not both. To clarify the Click's execution model, let us consider the following example.

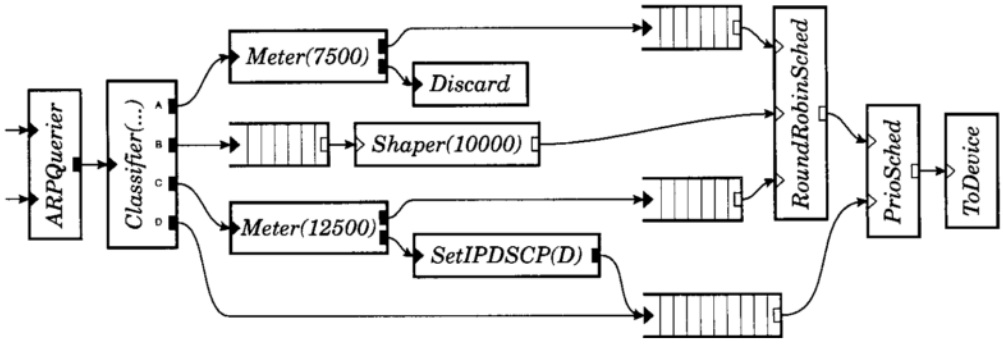

Figure 3. A sample traffic conditioning block. Meters and Shapers measure traffic rates.

Traffic Conditioning: Figure 3 shows a Click block that implements basic traffic conditioning. Elements are represented as rectangles. The black and white ports are push and pull ports respectively. The static parameters used to configure an element are shown in parentheses. The pull requests generated from the ToDevice are prioritized through the PrioSched element in favor of selecting the right input which points to a RoundRobinSched element that selects its inputs alternately. If both of the queue elements selected by the RoundRobinSched element are empty, the PrioSched element selects its left queue element to pull. The Meter regulates the flow of traffic: if the number of packets per second exceeds the preconfigured rate, the excess packets are forwarded to the Discard element. Similarly, the Shaper element also restricts the rate to pull packets below the given number argument. If the rate is equal to or more than the given rate, the Shaper element returns null.

Taxonomy: According to the previously introduced taxonomy, Click is a building block design

which is not limited to router applications despite its focus. Interfaces for programmers to access low level hardware mechanisms are exposed and real machines from the architectural aspect are targeted. The taxonomy from the language aspect makes less sense for Click since the router application mainly runs on a single machine even though there are packet exchanges with other machines. However, in terms of the programming paradigm, Click elements are written in an imperative language while the configuration is declarative for ease of use as shown in Figure 4. Its modular design principle should still work for WSN applications.

```
// Declare three elements ...
src :: FromDevice(eth0);
ctr :: Counter;
sink :: Discard;
// ... and connect them together
src -> ctr;
ctr -> sink;

// Alternate definition using syntactic sugar
FromDevice(eth0) -> Counter -> Discard;
```

Figure 4. Two Click-language definitions for the trivial router configuration.

Implementation: Click is implemented in C++ and runs on top of Linux OS. In order to provide an extensible and modular design without impacting routing performance, the authors had to optimize how packets receptions and transmissions are handled when a large volume of packets (of small size) are routed. The authors found that such a workload resulted in poor routing performance due to the significant overhead incurred by the numerous I/O interrupts necessary to handle packet transmissions and receptions. To avoid these performance penalties, the network driver was modified to use polling and check the DMA buffers to receive/transmit packets (and avoid interrupt processing). This approach significantly boosts the performance of the router by three times over the standard Linux IP routing implementation in Figure 5 w.r.t. the maximum loss-free packet forwarding rate (MLFFR) while maintaining extensibility for future changes. Notably, why the forwarding rate stops growing is because of the bottleneck of the router's CPU.

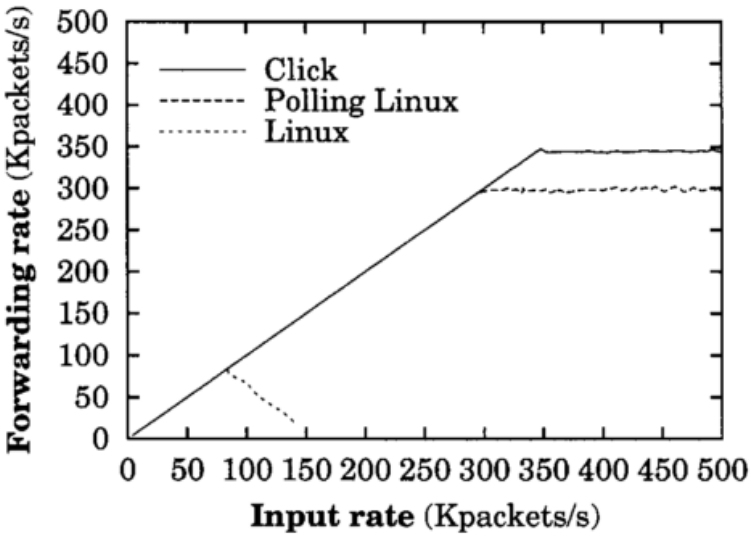

Figure 5. Forwarding rate as a function of input rate for Click and Linux IP routers.

In summary, Click allows dataflows to be executed on a single machine in an asynchronous and efficient manner. Its applicability will be discussed in the Evaluation section. Next, we will consider the problem of executing such dataflows across multiple nodes in a distributed manner.

Macroprogramming: WaveScript

Macroprogramming offers the promise of simplifying the programming of complex distributed systems by allowing programmers to reason about systems in a centralized manner. A dataflow programming abstraction could be used to this end: application domain experts would concentrate on describing the system in a concise high-level language that can be transformed into platform-dependent code for data transfer and synchronization. Such an approach makes it necessary to address the following two key challenges. First, we must define synchronization primitives to synchronize dataflows across multiple nodes effectively. Second, we must define a procedure to map the components of a system's dataflow onto nodes where they will be executed. This mapping has a significant impact on the system performance. In the following, we will introduce WaveScript [9] and its associated compiler that address the aforementioned challenges.

Programming abstraction: The basic programming abstraction of WaveScript is similar to that of Click: programmers can define functions that take streams as inputs and produce streams as

outputs. A stream is a *first-class* primitive in WaveScript and supports two basic stream operators: **iterate** and **merge**. The **iterate** allows specifying how to process each element sequentially in a stream and returns an output stream. To facilitate the **merge** operation, each element of a stream is timestamped. With the time information, the merge operator combines the two streams according to the timestamp of each element. More complex operators may be derived from **iterate** and **merge.** For example, the **map** operator iterates a stream and applies a function to each data item. The return values of the function constitute the output stream.

Hybrid Synchronicity: WaveScript adopts a hybrid synchronous model: at its core, WaveScript is asynchronous in nature as data items that may arrive at arbitrary intervals over communication channels. However, samples from sensors can be grouped into segments -- called **SigSeg** -- which contain uniformly sampled data. The timestamp of each sample in a SigSeg can be determined based on the timestamp of the first sample and the sample rate at which the SigSeg is collected. The target application of WaveScript necessitates only simple synchronization primitives between associated streams that can be implemented without requiring global time synchronization. For example, for two or more event streams without pre-existing relationship, it is possible to build arbitrary synchronization policies on top of merge such as **zip** and **syncselect**. The zip stream operator outputs a stream of tuples by pairing elements from its two input streams in FIFO order. The **syncselect** stream operator takes a control signal that specifies a time period and selects at least one data item within that time period for each input stream to produce the synchronized output stream.

Distributed Programs: A stream graph can be derived from a WaveScript program that defines a WSN application with functions of streams. Nonetheless, code generated that implements the stream operators in the stream graph eventually has to be distributed across multiple devices and/or machines to execute the application. In a WaveScript program, developers can specify which devices or machines where a set of stream operators and streams are located by defining namespaces. A namespace stands for a device or machine and informs the compiler to generate code of those enclosed stream operators for that platform. If a stream appears in multiple namespaces, the compiler will insert communication code to wire the stream between those devices and machines.

Programming Example: To highlight the programming model used by WaveScript, let us consider the marmot-detection application as an example. Marmots make loud alarm calls when their territory is intruded. The sound is captured by four microphones and analyzed to detect such alarms.

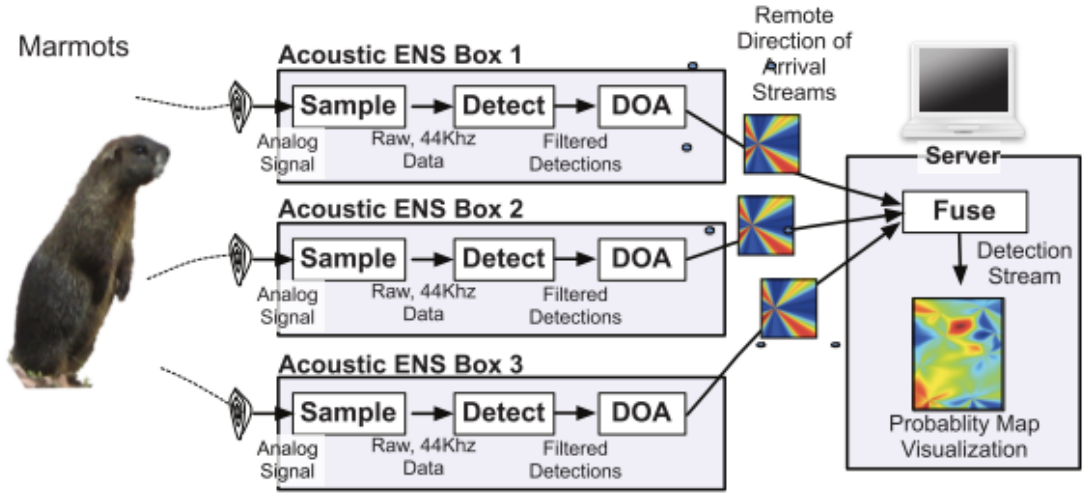

Figure 6. In the marmot-detection application, each acoustic ENS Box captures a channel of audio to process, filter and detect alarm events. The directions of arrival of alarm events are then analyzed to upload to a remote server to locate the marmot.

If an alarm is detected, all the samples during the event period from the four microphones are processed to estimate the location of the marmot by evaluating the likelihood of sound arriving from a particular angle. By combining these likelihoods from the four microphones, the application creates a grid map indicating the predicted location of the marmot. Figure 6 illustrates the major components of the application.

In the following, Figure 7 and Figure 8 show the WaveScript program of the marmot-detection application.

```
fun marmotScores(strm) {
  filtrd = bandpass(32, LO, HI, strm);
  freqs  = toFreq(32, filtrd);
  scores =
   iterate ss in freqs {
     emit Sigseg.fold((+), 0,
            Sigseg.map(abs, ss));
   };
  scores
}
```

Figure 7. A stream transformer sums up the energies of windowed audio samples in the frequency domain.

```
// Node-local streams, run on every node:
NODE "*" {
  (ch1,ch2,ch3,ch4) = VoxNetAudioAllChans(44100);
  // Perform event detection on ch1 only:
  scores :: Stream Float;
  scores = marmotScores(ch1);
  events :: Stream (Time, Time, Bool);
  events = temporalDetector(scores);
  // Use events to select audio segments from all:
  detections = syncSelect(events, [ch1,ch2,ch3,ch4]);
  // In this config, perform DOA computation on VoxNet:
  doas = DOA(detections);
}
SERVER {
  // Once on the base station, we fuse DOAs:
  clusters = temporalCluster(doas);
  grid = fuseDOAs(clusters);
  // We return these likelihood maps to the user:
  main = grid;
}
```

Figure 8. A WaveScript program of the marmot-detection application.

In Figure 7, the function `marmotScores()` computes and returns the energy of an input audio stream across several frequency bands as the output stream. `bandpass()` is a filter stream operator that allows only sound samples in the frequency range of the marmot's calls to pass. `toFreq()` transforms the output stream of `bandpass()` into a stream of sound energies in different frequency bands. The return stream `scores` is the sum of the sound energies in different frequency bands.

In Figure 8, the WaveScript program of the marmot-detection application demonstrates how events are detected through stream operators and when the WaveScript compiler determines to bridge partitioned streams with communication channels. The namespaces **`NODE`** and **`SERVER`** define a data stream to be deployed on different types of devices. The links between namespaces are determined based on shared variables across namespaces. In the considered example, since the stream `doas` appears in both the **`NODE`** and **`SERVER`** namespaces, the WaveScript compiler will implicitly creates a communication channel to wire stream `doas` from **`NODE`** devices to the **`SERVER`** machine. The mechanisms given by WaveScript provide a convenient way to express an application as dataflows. Developers need only to focus on application logic instead of low-level communication and synchronization details.

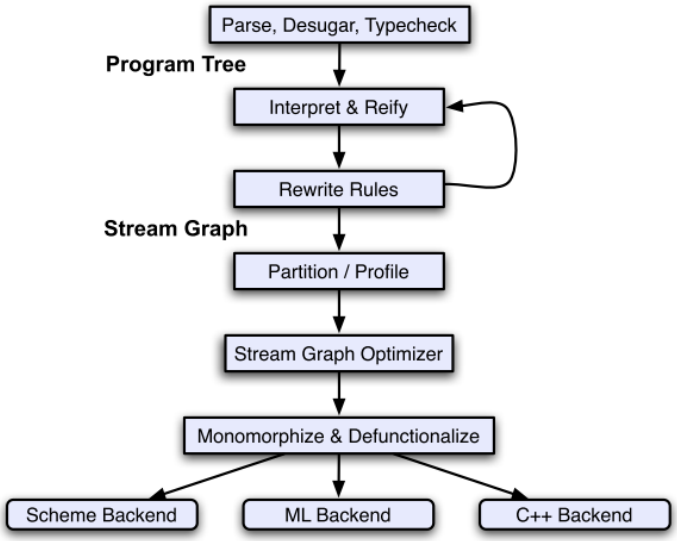

Figure 9. Compilation workflow and compiler architecture.

Optimization Framework: In addition to the high level syntax, WaveScript compiler also performs several optimizations during compilation by reducing a program into a stream graph. Figure 9 illustrates the compilation process. The following explanations consider the marmot-detection application. In the Interpret & Reify pass, the WaveScript program is fed into an interpreter to construct a stream graph by tracing back from the last returned main stream through stream operators to the streams sources. In the Rewrite Rules pass, algebraic rewrite rules are applied to simplify stream transformations or facilitate parallelism for other

optimizations. For example, the left hand side of the following expressions is equal to the right hand side but less computation complexity is incurred.

$$dewindow(window(n, s)) = s$$
$$window(n, dewindow(s)) = rewindow(n, 0, s)$$
$$rewindow(x, y, rewindow(a, b, s)) = rewindow(x, y, s)$$
$$rewindow(n, 0, window(m, s)) = window(n, s)$$

There is another example shown below. The left hand side expression seems to leave no room to extract parallelism but the equivalent right hand side is suitable for parallel processing of applying function *f* to *x* and *y* streams.

$$map(f, merge(x, y)) = merge(map(f, x), map(f, y))$$

Rewrite rules allow for optimizations across libraries built by different developers. For instance, some filters work on the frequency domain so usually an fft is performed to convert time domain samples in the beginning. If an ifft is not always performed to convert the samples back at the end, composing such two filters together may cause trouble. Otherwise, when two of such filters are adjacent, rewrite rules help cancel the fft/ifft pair, removing unnecessary overhead while retaining consistent interfaces. Next comes the Partition/Profile pass where those movable stream operators which can execute on `SERVER` or `NODE` side in the resulting stream graph from the previous pass are profiled by feeding instrumented program on target devices to evaluate running time for the operator work functions and internal loops. The profiling data is then fed into a graph partitioning algorithm to determine which side a movable operator should belong to in terms of CPU utilization and network bandwidth consumption. After the partitioning of the stream operators, stream graph optimizer performs a series of intra-node optimizations as follows since each stream operator in this pass is fixed at a particular node (e.g. `SERVER or NODE`).

- **operator placement:** parallel paths at graph split-joins are assigned to different CPUs in a round-robin fashion.
- **fusion:** neighboring stream operators are assigned to the same CPU if all of them are lightweight.
- **fission of stateless operators:** a stateless stream operator is duplicated by inserting a round-robin splitter and joiner before and after so that the resulting parallel paths can be

assigned to different CPUs for parallelism.

- **fission of array comprehension:** whenever a stream operator needs to search a parameter space exhaustively and records the results in an array or matrix, the compiler splits the operator based on the number of CPUs available by dividing the parameter space.

At this point, the compiler has gone through all the optimizations and the remaining task is to reduce the stream graph into a monomorphic and intermediate language as well as send through one of WaveScript's backends to generate native code for target platforms. We omit the details here since this process is not our focus and no new optimization techniques are proposed. To sum up, compared with building block based design, macroprogramming does have more opportunities of optimizations during compilation. Code generation for multiple devices and communication also relieves the burden of developing WSN applications. Nonetheless, manual program partitioning, the lack of power-aware, fine-grained distributed protocols such as multi-hop communication and time synchronization as well as consistent debugging across heterogeneous devices might still be the the limitations of building practical large-scale WSN applications using this macroprogramming approach. Therefore, further research is necessary.

After going through the building block design and the macroprogramming approach, I am wondering if it is possible to benefit performance or energy savings by migrating an computation intensive element in Click or a stream operator in WaveScript at runtime to a nearby device or machine. In traditional WSNs where sensor nodes are fixed, this idea does not make much sense but in Mobile sensor networks (MSNs), powerful computing resources at hand are not uncommon. Why not exploit those external resources seamlessly? In the next section, we will consider the issues of remote execution and explore a recent implementation.

Remote Execution: MAUI

Remote execution is a mechanism that enables an application to overcome its hardware limitations by offloading parts of its execution on remote, resource-rich devices that have continuous power. There are three challenges that must be addressed. The first is to determine what code to offload. Next, it is to dynamically decide when and where to offload code with

minimal overhead. Finally, additional effort by programmers should be minimized.

MAUI [2] addresses the above challenges as follows. First, a set of rules are specified to determine what can be offloaded. These rules preclude code pertaining to UI, I/O, multi-threading, and transactions that involve multiple parties from being offloaded. Next, MAUI continuously profiles the device, application, and network conditions to evaluate the energy cost and delay, making a runtime decision based on an integer linear programming (ILP) solver. Finally, to minimize programming effort, MAUI asks programmers to explicitly annotate methods to identify code that can be offloaded for remote execution. Of course, programmers are responsible for correct annotations and the language should support code annotations such as Java and C# .NET.

How it works: As is shown in Figure 10, the mobile application is installed on nearby machines in advance. A mobile device profiles method invocation, application state transfer costs and network connection status to decide whether to migrate the execution of a method to a remote machine or not. If the decision is to execute the method on another machine different from where the previous method is executed, all the associated states of that method have to be migrated to that machine before the invocation. If the method to invoke remotely takes longer than allowed, the MAUI runtime simply falls back to executing the method locally.

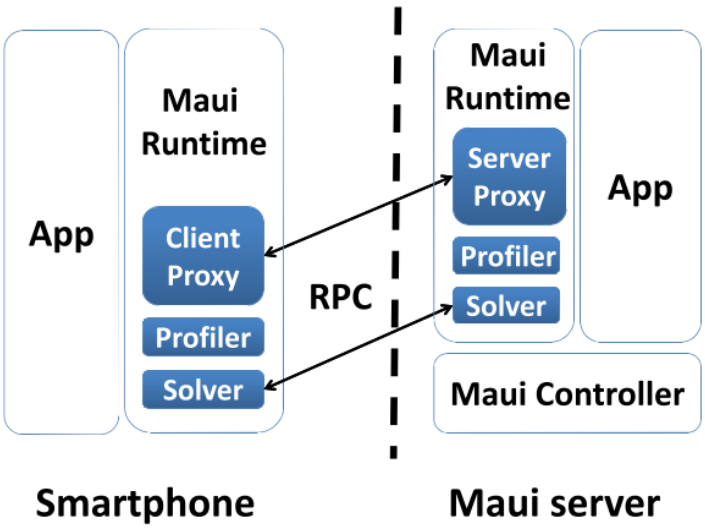

Figure 10. High-level view of MAUI's architecture.

Global Program Analysis: The profiled information of the mobile device is fed into the MAUI

IPL solver to make decisions by solving an 0-1 ILP problem formulated in terms of the call graph of the application as below.

$$\text{maximize} \sum_{v \in V} I_v \times E_v^l - \sum_{(u,v) \in E} |I_u - I_v| \times C_{u,v}$$

$$\text{such that:} \sum_{v \in V} ((1 - I_v) \times T_v^l) + (I_v \times T_v^r))$$

$$+ \sum_{(u,v) \in E} (|I_u - I_v| \times B_{u,v}) \leq L$$

$$\text{and} \qquad I_v \leq r_v, \ \forall v \in V$$

The objective function evaluates to be the total energy saved due to remote invocations minus application state transfer cost. I_v is an indicator variable. $I_v = 0$ if $method_v$ is invoked locally and $I_v = 1$ if $method_v$ is invoked remotely. E_v^l represents the energy cost when $method_v$ is invoked locally. $C_{u,v}$ represents the application state transfer cost that is paid when both neighboring $method_u$ and $method_v$ on the call graph are not invoked in the same location. The first constraint stipulates that the total time to execute the program including the application state transfer be within L. The second constraint stipulates that only methods marked remoteable can be executed remotely. From what is shown in the ILP, the solution is based on a global program analysis of the call graph of the application. What if the decision is made according to only application state transfer cost and method invocation cost of a particular method? Take a sample call graph shown in Figure 11 for example.

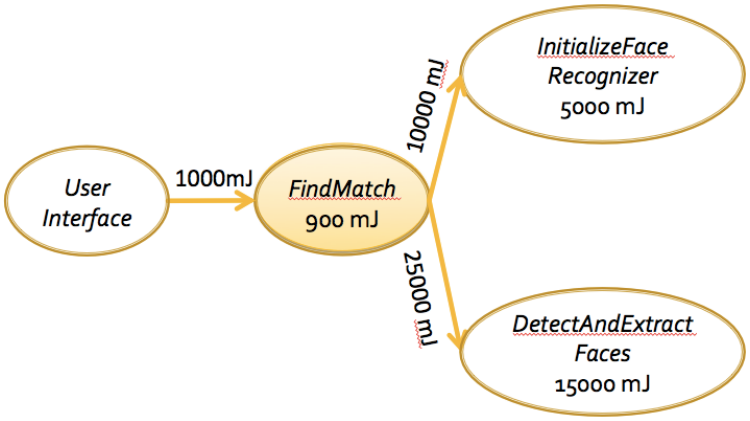

Figure 11. A sample call graph with edge weight for application state transfer cost and

node weight for method invocation energy cost.

If the decision is made for each method respectively, all the methods will be decided to execute locally because for an individual method, the application state transfer cost is always larger than the method invocation cost. But if the decision is made globally, all the three methods FindMatch(), InitializeFaceRecognizer() and DetectAndExtractFaces() will be deemed to execute remotely since the total energy savings dominate the one-time application state transfer cost of FindMatch().

Bottom Line: Their evaluation is impressive when the network latency is small. Otherwise, the execution falls back to be local. Some computation such as face recognition is so heavy that even remote execution through 3G connection is worth it. Overall, though their results are promising, the drawback is that the application has to be deployed on nearby machines beforehand and for each method, programmers have to annotate if the method can be remoteable. Fortunately, the former condition can be overcome by using a network classloader to download application code on demand while the latter hassle should be addressed by performing an offline bytecode analysis to enable fully automatic annotation. With those improvements, I believe remote execution would be a supportive mechanism for MSN applications in near future.

Evaluation

This section evaluates the building block design and the macroprogramming approach introduced earlier. In the building block design of Click, the flexibility and ease of programming provided by the underlying model should benefit WSN applications. For example, several IP router extensions for packet scheduling and dropping policies are available to switch and its modular configurations are easy to extend. However, it cannot be directly applied to WSNs for the following two reasons. First, the dataflow model must be generalized to allow for distributed execution of dataflows. Second, Click's push/pull operations are not well-suited for devices that operate on batteries. Consider the following example where a sink pulls data from a source. In the case when the sink pulls data at a higher rate than the source produces, many of the pull requests cannot be fulfilled, resulting in wasted computation. While this is not a problem to a

desktop machine, on mobile platforms this results in spending precious energy to no avail. A similar problem occurs in the case when push is used. If the source pushes a packet when the sink is not ready to process it, the packet would have to be buffered or dropped. A more intelligent mechanism is necessary to address this variant producer-consumer problem. For example, a pull may act as a request that does not have to be fulfilled immediately but rather when the data becomes available, preventing unnecessary pulls. On the other hand, from the perspective of parallelism optimization, task and pipeline parallelism can be realized by the scheduler but in-element data processing parallelism has to be manually coded.

WaveScript is promising macroprogramming approach that employs effective compiler optimization techniques. However, a limitation of WaveScript is that it provides the programmer with little control over where the stream operators may be deployed. This could result in uneven workloads that need to be performed by nodes, leading to shorter system lifetimes compared to when the workload could be. On the other hand, its data access model is based on data sharing internally. Hence, there is a scalability issue in nature when a stream is shared with a very large number of nodes.

Further, some requirements may not be easy to carry out without overriding built-in stream operators or extending the execution model. For instance, the random early detection (RED) element which implements a packet dropping policy in Click requires to locate and probe the queue usage of upstream or downstream queue elements. WaveScript might not support it well for non-neighboring stream operators to exchange information. Besides, applications that require to dynamically change dataflow paths are not suitable [10] unless the underlying routing protocol bears the stream graph in mind to enable load balance and maximal parallelism while constructing the dataflow paths.

Lastly, the remote execution framework MAUI opens the door to break physical computing resource limitations of mobile devices. This is particularly useful for dataflow-based WSN applications at runtime to relocate the processing elements or stream operators to nearby powerful machines for more energy savings or lending computing power. Nevertheless, the restrictions mentioned previously such as manual annotations to mark remoteable methods and pre-installation of applications on nearby machines should be addressed properly in ahead for

practical use.

Conclusions and Future Work

This report explored programming abstractions for WSN applications with a focus on dataflow. We believe the dataflow model is a promising solution for the following reasons. First, it is easy to compose and extend applications by wiring components. Second, it captures parallelism and communication explicitly, allowing further compiler optimizations. Last but not least, it also facilitates dynamic optimizations to enable fine-grained energy-aware load distribution in such systems.

All the aforementioned dataflow-based programming abstractions are related to our current work, CSense, a data collection toolkit for MSNs. We base CSense on the idea of dataflow because in addition to the advantages mentioned earlier, it is an intuitive abstraction to model data collection applications. Besides, incorporating foreign code is also straightforward in CSense. For instance, on the Android platform, CSense simplifies the process of exploiting compiled MATLAB functions in C to enable efficient math operations. We keep it in mind to maintain interfaces for low-level functionality and opportunities for extensions while developing high-level abstractions to boost programmers' productivity. In the near future, we are going to realize communication between devices and optimizations in terms of pipeline graphs. In conclusion, we expect developing WSN applications would be made easier and gain more popularity after overcoming those challenges discussed in this report.

References

[1] T. Abdelzaher, B. Blum, Q. Cao, Y. Chen, D. Evans, J. George, S. George, L. Gu, T. He, S. Krishnamurthy, L. Luo, S. Son, J. Stankovic, R. Stoleru, and A. Wood, "EnviroTrack: towards an environmental computing paradigm for distributed sensor networks," in *24th International Conference on Distributed Computing Systems 2004 Proceedings*, 2004, pp. 582–589.

[2] E. Cuervo, A. Balasubramanian, D. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "MAUI : Making Smartphones Last Longer with Code Offload," in *Energy*, 2010, pp. 49–62.

[3] C.-L. Fok, G.-C. Roman, and C. Lu, "Agilla: A mobile agent middleware for self-adaptive wireless sensor networks," *ACM Transactions on Autonomous and Adaptive Systems*, vol. 4, no. 3, pp. 1–26, 2009.

[4] D. Gay, P. Levis, R. Von Behren, M. Welsh, E. Brewer, and D. Culler, "The nesC language: A holistic approach to networked embedded systems," *PLDI 03 Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, vol. 38, no. 5, pp. 1–11, 2003.

[5] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, "System architecture directions for networked sensors," *ACM SIGARCH Computer Architecture News*, vol. 28, no. 5, pp. 93–104, 2000.

[6] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," *ACM Transactions on Computer Systems*, vol. 18, no. 3, pp. 263–297, 2000.

[7] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "TinyDB: an acquisitional query processing system for sensor networks," *ACM Transactions on Database Systems*, vol. 30, no. 1, pp. 122–173, 2005.

[8] L. Mottola and G. P. Picco, "Programming Wireless Sensor Networks : Fundamental Concepts and State of the Art," *ACM Computing Surveys*, vol. V, no. 3, pp. 1–51, 2011.

[9] R. R. Newton, L. D. Girod, M. B. Craig, S. R. Madden, and J. G. Morrisett, "Design and evaluation of a compiler for embedded stream programs," *ACM Sigplan Notices*, vol. 43, no. 7, p. 131, 2008.

[10] R. Newton, G. Morrisett, and M. Welsh, "The regiment macroprogramming system," *Proceedings of the 6th international conference on Information processing in sensor networks IPSN 07*, p. 489, 2007.

[11] M. Welsh and G. Mainland, "Programming Sensor Networks Using Abstract Regions," in *usenixorg*, 2004, p. 3.

[12] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler, "Hood: a neighborhood abstraction for sensor networks," in *Proceedings of the 2nd international conference on Mobile systems applications and services*, 2004, pp. 99–110.