Ensemble: A High Level Language for Sensor Networks

Yu David Liu
Computer Science Department
Johns Hopkins University

Scott Smith
Computer Science Department
Johns Hopkins University

Andreas Terzis
Computer Science Department
Johns Hopkins University

***Abstract*— In this paper we describe Ensemble, a proposed language framework for sensor network programming. Our goal is to provide a programming framework to scientists and engineers that will allow them to directly code sensor network applications, without the need for expertise in low-level device programming. The key concepts in Ensemble are high-level communication protocol *connectors*, and the ability for systems programmers to define new communication protocols as *metaprotocol extensions*.**

I. Introduction

Sensor networks are an emerging computing platform consisting of large numbers of small, low-powered, wireless sensor nodes each with limited computation and communication abilities. Several studies have been presented that show the feasibility to use sensor networks for a number of applications including habitat monitoring [1], [2] and target tracking [3]. Unfortunately, programming applications for sensor networks is extremely difficult, mostly due to the computation and communication limitations of individual sensor nodes. The low capabilities of individual sensors force application developers to write application-specific, low-level code to perform node coordination, communication and data collection. This development paradigm has in turn hindered the development of applicationssince that will realize the full potential of sensor networks, since only specialists can program the networks.

Existing projects have required teaming of application scientists with computer science experts, but this requirement significantly limits the growth potential for sensor network applications. Our goal is to create a programming framework for sensor networks which scientists and engineers can directly use to develop their applications. We believe such a high-level language will increase by an order of magnitude or more the number of sensor-based applications.Our approach is more future-looking in that it will entail slightly higher runtime overhead, but at a level we believe will be supported by the next generation of sensors.

The rest of the paper is structured as follows: In Section II we explain the philosophy behind Ensemble and give a high level description of the main language constructs. We use a sample application to show how these constructs can be used to write a sample sensor application in Section III. Section IV explains how metaprotocols are implemented while Section V describes how Ensemble code is translated to lower-level code. Finally, in Section VI we present the related work in this area.

II. Ensemble Described

The system architecture that we foresee is an event-driven, single-process, single-threaded system, similar to the one used in TinyOS [4]. Hardware generates events (via interrupt handlers) that queue up in an event queue; the application-level code consists of a set of event handlers, each of which will quickly compute. So, the top-level activity is to repeatedly pull events off the queue and invoke the appropriate handler. This event queue and the notion of these events is not directly exposed to application programmers; they provide the event handlers that get automatically called when a new event arrives.

The intended users of Ensemble are scientists and engineers (*field programmers*) with an understanding of basic programming concepts and a high-level understanding of sensor networks, but without any knowledge of low-level aspects of sensor implementations.

Our main design principle is a clear separation of low-level concerns: the field programmers will not ever see the low-level code, and they will have a simple, understandable, high-level interface to this code.

The field programmers are supported by independent *systems programmers* who can write new low-level metaprotocols to extend the Ensemble framework. They are called metaprotocols because they can be viewed as language extensions, enriching the Ensemble language provided to field programmers. The Ensemble system will come with a large library of these protocols that field programmers can choose from for their particular application; field programmers that team with systems programmers can produce new protocols that can be used by other field programmers. The high-level programs provided by field programmers and the low-level programs provided by systems programmers are semantically linked together, and yet their developments minimally interfere with each other.

Sensor network architectures have widely varying communication layers. The underlying route paths may be structured in many ways (e.g. a tree or a general mesh graph). Routing may also be content-driven in that a request is made just for data, not connection to a particular node. Furthermore, communication protocols need to deal with depleted battery lifetimes and network disconnection scenarios. Nonetheless, a common theme to all routing protocols is the presence of *publishers* (i.e. nodes that generate data) and *subscribers*, that is nodes that are interested in receiving this data. Our idea is to provide a number of metaprotocols matching the different underlying routing mechanisms, and field programmers can choose the most appropriate metaprotocol for their application.

In a content-based model, the publisher publishes formatted data without specifying who the subscribers are, and the subscriber does not have a specific fixed publisher; instead, it specifies what kind of message content it is interested in. On the other hand, in the node-based model, the publisher publishes data with a clear set of subscribers in mind. Although subscriptions could be done via different ways (static or dynamic, push or pull, one-to-one or one-to-many), the publisher always has a clear picture where to send data when publishing happens.

Another communication dimension is physical-location-based routing. This can e.g. be used in tracking moving objects, or by algorithms where nodes are for example recording temperature at adjacent nodes. The protocol would operate at a very high level, and assume there is some algorithm/device in place for pinpointing locations of all sensors. Connections can then be made to other sensor(s) in a particular direction, within a certain radius, etc, all independent of the radio.

Another important dimension of sensor communications is the need to manage power. What this means in terms of the communications protocol is to provide a means to rollback to a less power-hungry communications mode when the battery power has been depleted below a certain point.

Along with the need to manage power, in mobile sensor nets the case of completely disconnected operation should be supported. A single sensor or groups of sensors may roam away from the other nodes, and not be able to communicate data to the sensorhead. The communications protocol in this case will want to cache some of the data observations locally so that not all data is lost.

In the next two sections, we will describe the Ensemble language through an example.

III. Field Programming in Ensemble

Figures 2 and 3 illustrate what engineers and scientists need to write. The code is naturally divided into two parts: the module to be deployed in sensing nodes or intermediate aggregating nodes (Fig. 2) and that to be deployed in the sensor head (Fig. 3). In this particular example, sensors locally collect temperature measurements and forward them to the sensorhead. The forwarding happens over a routing tree rooted at the sensorhead. The sensorhead relays collected data measurements back to the scientists through a long-haul connection. The sensing and reporting frequencies are variable and can be altered through commands broadcasted by the sensorhead over the routing tree.

The basic code unit in Ensemble is an *assemblage*, a form of module.

Like most modules, assemblages have a means to import and export functions from other mod-

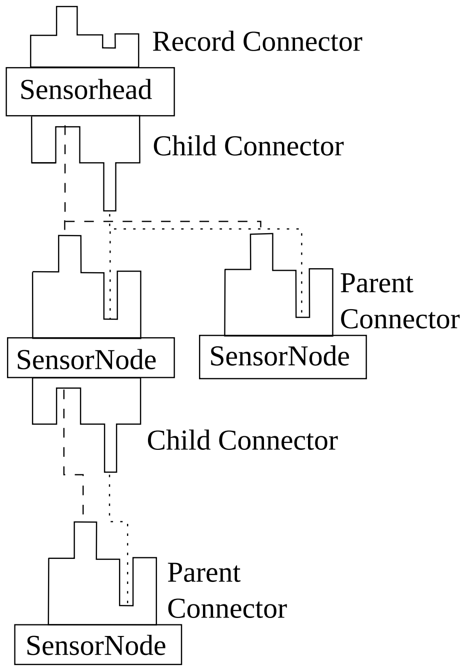

Fig. 1. Connectors

ules, via its *linkers*. Linkers are bi-directional interfaces which specify what functions they provide (**export**) and what functions they expect (**import**). Two assemblages with complementary linkers can be linked together via the process of static linking. Linkers in assemblages are similar to `interfaces` in NesC: functions defined in linkers are asynchronous, and cross-linker invocations are put into an event queue. Functions defined in a static linker are invoked by the syntax $f@v(\ldots)$, where f is the function name and v is the linker name importing/exporting the function.

The main novelty of Ensemble is how it provides an explicit communications interface in assemblages, its *connectors*. Two assemblages running on different nodes may be connected via their complementary connectors. For instance in Fig. 2, assemblage TempCollect has two connectors, Parent and Child. The Parent interface defines how a running `TempCollect` assemblage should communicate with its parent (recall that in a tree-based routing scheme, a sensor has a unique parent). The process of connection establishment is achieved by the expression $\mathbf{connect}_{\mathsf{Parent}\mapsto\mathsf{Child}}$, which connects the Parent connector of the current running assemblage with the Child connector of

```
#define SENSE 0
#define REPORT 1
#define BUF_SIZE 10

struct Point {int x, y, z};
struct ReportType {int tm; Point pt};


assemblage TempCollect {
    linker Sensor
    {
      import void getData();
      export void dataReady(int tm)
        { store( {tm, whereami()} );  }
    }
    linker Timer
    {
      import void start(char tp, int itvl);
      export void fired(char tp) {
        case tp of
        SENSE :  getData@Sensor();
                 start@Timer(SENSE, sitvl)
        REPORT : phandle ▷ report(aggr());
                 start@Timer(REPORT, ritvl)
      }
    }
    connector Parent implements ParentInTree
    {
      import void report(ReportType r);
      export void setItvl(int si, int ri){
        sitvl := si;
        ritvl := ri;
        forall c in Child {c ▷ setItvl(si, ri); }
      }
    }
    connector Child implements ChildInTree
    {
      export void report(ReportType r)
        {store(r); }
      import void setItvl(int si, int ri);
    }
    local Parent phandle;
    local int ritvl;
    local int sitvl;
    local ReportType rstore[BUF_SIZE];
    local Point whereami(){
       ...returns the location of the node itself ...
    }
    local ReportType aggr(){
       ...aggregation code over rstore ...
    }
    local void store(ReportType r){
       ...selectively stores report r in rstore ...
    }
    main{
      loc := whereami();
      phandle := connect_Parent↦Child
    }
}
```

Fig. 2. Ensemble Code for Sensing Nodes

```
#define BUF_SIZE 10

struct Point {int x, y, z};
struct ReportType {int tm; Point pt};


assemblage SensorHead {
    connector Record implements Satellite
    {
      import void init(void);
      import void process_done(ReportType r);
      export void setFreq(int si, int ri)
        { forall c in Child {c ▷ setItvl(si, ri); }}
      export void report(void)
        { current ▷ process_done(aggr()); }
    connector Child implements ChildInTree
      export void report(ReportType r)
        {store(r); }
      import void setItvl(int si, int ri);
    }
    local ReportType rstore[BUF_SIZE];
    local ReportType aggr(){
       ... aggregation code over rstore ...
    }
    local Point store(ReportType r){
       ... selectively stores report r in rstore ...
    }
    main{ }
}
```

Fig. 3. Ensemble Code for Sensor Head

some running assemblage on a different node. The precise meaning of the **connect** expression is defined in its metaprotocol, which will be explained in Sec. IV below.

In the tree-based routing example, a sensor might be connected with multiple children at the same time. The number of children of a given a node cannot be determined at compile time since it depends on the physical location of sensors during deployment. Thus, it is not possible to pre-define a list of connectors, each for a child. In Ensemble, the connectors are *generative*: a connector can simultaneously support multiple connections. For instance in Fig. 2, although there is only one connector Child, the parent can have multiple children connected, and all children communicate with their parent via the Child connector. To achieve this, the **connect** expression returns a *connection handle*, and we can distinguish different connections to the same connector by their unique handles. Invocations on connectors follow the $h \triangleright f(\ldots)$ syntax, where h is the connection handle and f is the function name.

Other expressions meriting explanation include **forall** h **in** v $\{e'\}$, used to obtain all connections currently connected to a specific connector v. Keyword **current** is reserved to mean "the current connection", useful for callbacks. Notice in sensor head's Record connector, when the aggregation is done, the sensor head needs to send back the report to the running assemblage which initiated the report invocation.

IV. Systems Meta-programming

The core Ensemble language contains no concrete communication protocol for connectors; it is only a framework. Systems programmers must write metaprotocols. This section describes how.

The connectors above have **implements** qualifiers such as **implements** Satellite for connector Record in Fig 3; these qualifiers specify the communication metaprotocol used by that connector. The low-level details of satellite communication is not specified by assemblages during field programming. Instead, the Ensemble architecture allows systems programmers to define these aspects at a meta-programming level. Notice that in this sense the Ensemble architecture is extensible at the system level: Satellite is not a reserved word of the language; system programmers could very well define metaprotocols for Satellite2 at some time, and field programmers can then qualify their connectors with **implements** Satellite2. Even better, if some version control is allowed, systems programmers could implement different versions of Satellite connector meta-protocols and field programmers do not change any of its level of assemblage code but still change the behavior of the communication.

We use the Satellite metaprotocol to show how metaprotocols are implements at the system level. Each protocol is defined as a library of NesC functions that must include the following:

- `Satellite_setup()`
- `Satellite_connect(src_cname, dst_cname, node)`

- `Satellite_disconnect()`
- `Satellite_conn_invoke(h, import_fname, args[])`

The `Satellite_connect()` function is invoked to set up the connection to the indicated destination. The argument is optional; if not specified, the body of the function will lead to some discovery of the node, which depending on strategies, could be the closest node, or the node with the strongest signal, or the most reliable node. This function defines the semantics of the $\mathbf{connect}_{v \mapsto v'}\ e$ expression.

`Satellite_conn_invoke(h, import_fname, args[])` defines the low-level protocol for invoking a particular imported function with name `import_fname` on a connection handle `h` of the Satellite connector. It includes how such an invocation is packed up in a message and how the message is sent via low level system calls. This function at a meta-level defines the semantics of an $h \triangleright f(e)$ expression. The other two meta-protocol functions `Satellite_setup` and `Satellite_disconnect` are used to setup and tear down any required connection state respectively.

The meta-protocol functions we sketched here are only part of the communication interface covering only basic communication between paired connectors. Other potential communication requirements include the need to implement a reliable channel on top of the unreliable wireless medium, the need to support dynamic topology change and the need to support disconnected operations. The separation between field- and meta-programming allows Ensemble field programmers to use these features by using an appropriate meta-protocol, and not programming them directly themselves.

V. Implementation

Here we briefly describe our proposed implementation methodology.

At the field-programming level, we plan to translate assemblages to NesC. Assemblage linkers may be directly translated to NesC `interfaces`. Connector declarations will not generate code in the target language, and the added expressions related to the use of connectors need to be translated into invocations of the meta-protocols functions, as defined in the previous section.

The meta-protocols may be programmed directly in NesC. Each connector meta-protocol, such as Satellite, ChildinTree, and ParentinTree, can be implemented by a NesC module. All Ensemble is doing is providing a uniform interface to communication protocols, and so there will be very little additional overhead introduced.

VI. Related Work

The first generation of sensor applications was written using languages such as NesC [5] and Mate [6] built on top of TinyOS [4]. While both were major engineering achievements given the severely limited resources of current generation sensors, languages like NesC are low-level and as such are more suited to systems programmers than *field programmers*, who are the intended users of Ensemble. Recently, Levis *et. al.* [7] have observed that many user-level applications are using similar sets of high level constructs, such as tree routing. This finding supports our claim that sensor applications share a set of common characteristics and that application writers would benefit if these building blocks would be supported by a high level language that lets them focus on the application domain.

The need for higher level programming abstractions has been observed by many researchers, who have proposed different ways to achieve this goal. One school of thought, articulated by Madden *et. al.* in [8], treats the sensor network as a *streaming database* in which sensor measurements correspond to database tuples. Users can extract different measurements by sending different SQL-like queries to the sensor network. Ensemble is a lower-level language abstraction compared to declarative languages such as SQL, but gives field programmers the freedom to implement a wide range of applications. Our work is closer in spirit to the proposal of Welsh and Mainland in [9], where they propose the concept of *abstract regions*. Their basic idea is that a sensor can define a “neighborhood” of sensors around it and share data with sensors in that neighborhood. Culler et al in [10] proposed a similar concept and showed how it can be useful in writing sensor applications such as sensor tracking. Our model supports the concept of neighborhood communication as one of the many communication models through the use of *meta-protocols*.

The Ensemble language design has several precedents. Ensemble. Assemblages, with interfaces both for static linking and inter-node communication, were developed in [11]. The aforecited project is a general-purpose language for distributed software design, and this paper represents a re-targeting of those general concepts to the specific domain of sensor applications. Module systems and calculi with a focus on static linking are numerous, and many support bi-directional static linking interfaces, such as Units [12] and mixins [13]. In these works, invocations across static linking interfaces are no different from normal function calls. We take the approach of NesC, where event queues are used across module boundaries due to the strong bias toward hardware of sensor networks. In this sense, our linkers are very similar to the `interface` in NesC. Connectors as a general language construct have been previously proposed in various forms including [14], [15]. Metaprogramming is well-known to serious Lisp or Smalltalk programmers; it also has been used to allow programmer-based specification of communications protocols, see *e.g.* [**?**].

This paper proposes a core design of a sensor programming language that can be directly used by scientists. Although the language as described will work, it only represents the first step down a long path. Connector metaprotocols should be composable, for example composing a tree routing scheme with a particular low-power rollback protocol to give a power-aware tree routing protocol. It is not hard to define a simple composition scheme, but it is very difficult to deal with potential interference between the composed protocols.

References

[1] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson, "Wireless sensor networks for habitat monitoring," in *Proceedings of 2002 ACM International Workshop on Wireless Sensor Networks and Applications*, Sept. 2002.

[2] A. Cerpa, J. Elson, D. Estrin, L. Girod, M. Hamilton, and J. Zhao, "Habitat monitoring: Application driver for wireless communications technology," in *Proceedings of 2001 ACM SIGCOMM Workshop on Data Communications in Latin America and the Caribbean*, Apr. 2001.

[3] K. Pister, "Tracking vehicles with a uav-delivered sensor network," Mar. 2001.

[4] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, "System architecture directions for network sensors," in *Proceedings of ASPLOS 2000*, Nov. 2000.

[5] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler, "The nesc language: A holistic approach to networked embedded systems," in *Proceedings of Programming Language Design and Implementation (PLDI) 2003*, June 2003.

[6] P. Levis and D. Culler, "Mate: A tiny virtual machine for sensor networks," in *International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, USA*, Oct. 2002, to appear.

[7] P. Levis, S. Madden, D. Gay, J. Polastre, R. Szewczyk, A. Woo, E. Brewer, and D. Culler, "The Emergence of Networking Abstractions and Techniques in TinyOS," in *Proceedings of NSDI 2004*, Mar. 2004.

[8] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "The Design of an Acquisitional Query Processor for Sensor Networks," in *Proceedings of SIGMOD 2003*, June 2003.

[9] M. Welsh and G. Mainland, "Programming Sensor Networks Using Abstract Regions," in *Proceedings of NSDI 2004*, Mar. 2004.

[10] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler, "Hood: A neighborhood abstraction for sensor networks," in *Proceedings of the International Conference on Mobile Systems, Applications and Services (MOBISYS '04)*, June 2004.

[11] Y. D. Liu and S. Smith, "Modules with interfaces for dynamic linking and communication," in *Proceedings of the Eighteenth ECOOP*, June 2004.

[12] M. Flatt and M. Felleisen, "Units: Cool modules for HOT languages," in *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, 1998, pp. 236–248.

[13] D. Duggan and C. Sourelis, "Mixin modules," in *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '96)*, vol. 31(6), 1996, pp. 262–273.

[14] R. Rinat and S. Smith, "Modular internet programming with cells," in *Proceedings of the Sixteenth ECOOP*, June 2002.

[15] J. Aldrich, V. Sazawal, C. Chambers, and D. Notkin, "Language support for connector abstractions," in *Proceedings of the Seventeenth ECOOP*, June 2003.