Robust Programming for Sensor Networks

Francisco Martins
LASIGE/DI-FCUL, Portugal
fmartins@di.fc.ul.pt

Luís Lopes, Miguel S. Silva
CRACS/DCC-FCUP, Portugal
{lblopes,mssilva}@dcc.fc.up.pt

João Barros
IT/DCC-FCUP, Portugal
barros@dcc.fc.up.pt

Technical Report Series: DCC-2008-01

Robust Programming for Sensor Networks

Francisco Martins
LASIGE/DI-FCUL, Portugal
fmartins@di.fc.ul.pt

Luís Lopes, Miguel S. Silva
CRACS/DCC-FCUP, Portugal
{lblopes,mssilva}@dcc.fc.up.pt

João Barros
IT/DCC-FCUP, Portugal
barros@dcc.fc.up.pt

Abstract

Aiming at a sound formal basis for the design and implementation of robust programming languages for sensor networks, we present a process calculus that captures their main characteristics in terms of computational resources and communication abilities. The calculus, which has straightforward semantics and is rather expressive, features a static type system that allows premature detection of application protocol errors. Our main results include subject reduction and type safety proofs, as well as an initial implementation of a modular interpreter.

keywords: Sensor Networks, Process-Calculi, Programming Languages, Virtual Machines

1 Introduction

An ideal programming platform for sensor networks would allow a software engineer to write a distributed application in a high-level idiom, to debug the code on a personal computer, and to deploy the program automatically onto potentially large numbers of small sensing devices [2] equipped with wireless transmission capabilities. In spite of abundant and very active research on this class of distributed systems, the majority of available sensor network programming tools [3] are still rather distant from the aforementioned ideal platform, most notably the module-based idiom nesC [8], which promotes a system level programming style on top of small-scale operating systems such as TinyOS [1] and Contiki [5]. In contrast, TinyScript/Maté [11] can be regarded as a step in the right direction, providing programmers with a suitable abstraction layer for the hardware. Other examples such as Deluge [10] and the Agilla middleware platform [7] enable higher level control for critical operations such as massive code deployment.

Deemed as a relevant step towards the definition of higher-level languages for sensor networks, Regiment [17] adopts a data-centric view of sensor networks and provides the programmer with useful abstractions to manipulate data streams and to manage network regions. Although Regiment is a strongly typed language — a technical characteristic widely recognized as an essential ingredient towards scalable development of robust applications — its construction is not based on a formal calculus and it is not clear that the semantics is amenable to prove correctness results for the system and applications.

Invariably, the state of the art in the design of sensor network programming languages follows a *top-down* approach, in which system engineers start by identifying useful patterns and abstractions based on case studies of applications and then attempt to provide the programmer with language constructs and system features, which must be written in nesC/TinyOS code or some other platform similar to a low-level operating system. The problem with such approaches is that the semantic gap between the original language specification and the actual implementation inevitably precludes a thorough analysis of the correctness of the envisioned sensor networking application.

Seeking a fundamentally sound path towards the development of programming languages for sensor networks, we propose a somewhat disruptive *bottom-up* approach. Inspired by process calculi theory [9, 16], our basic idea is to start by constructing a fundamental programming model, which (a) captures the specific computing and communication aspects of sensor networks and (b) enables us to reason about their fundamental operations. Previous work on process calculi for wireless systems exist [14, 18, 20], however the results are scarce and do not address the specificities of sensor networks. Seeking to close this gap, we presented in [12] a preliminary version of our Calculus for Sensor Networks (CSN).

In this paper, we go one step forward towards a robust higher-level programming language by making the following contributions:

- an improved model that extends the notion of application module;

- a new type-system that allows us to check statically the well-formedness of sensor network applications and to diagnose potential would be run-time errors prematurely;

- a *subject reduction* and a *type safety* result that together are fundamental in establishing the calculus as a rigorous framework to reason about sensor network applications;

- a prototype implementation for the system that enables us to simulate the behavior of CSN networks and applications.

After establishing the basic theoretical properties for the calculus, we propose using it as a specification for an intermediate-level language upon which other high-level programming abstractions may be implemented as derived constructs, thus preserving the semantics of the underlying calculus.

The remainder of the paper is organized as follows. Section 2 provides a formal description of our calculus, followed by a set of examples in Section 3. First steps towards a high-level language are outlined in Section 4, whereas Section 5 describes our new type system. Section 6 is devoted to a first implementation of an interpreter and Section 7 concludes the paper.

2 The Calculus

This section presents the syntax and the semantics of the Calculus for Sensor Networks. For simplicity, in the remainder of the paper we refer to a sensor node or a sensor device in a network as a *sensor*. The syntax is provided by the grammar in Figure 1, and the operational semantics is given by the congruence and reduction relations depicted in Figures 2 and 3.

S ::=	*Sensors*
$\mathbf{0}$	empty network
$\mid S \mid S$	composition
$\mid [\vec{P} \triangleright \vec{C}]_e^p$	sensor
P ::=	*Processes*
v	value
$\mid v.l(\vec{v})$	function call
$\mid$ **transmit** $X.l(\vec{v})$	transmission
$\mid$ **install** (v)	install module
$\mid$ **let** $x = P$ **in** P	new variable
$\mid$ **post** $\{P\}$	defer execution
C ::=	*Modules*
$X :: M$	module
M ::=	*Anonymous Modules*
$\{l_i = (\vec{x_i})\ P_i\}_{i \in I}$	anonymous module
v ::=	*Values*
b	built-in value
$\mid x$	variable
$\mid X$	module name
$\mid C$	module
$\mid M$	anonymous module

Figure 1. The syntax of sensors.

Syntax. Let $\vec{\alpha}$ denote a possibly empty sequence $\alpha_1 \ldots \alpha_n$ of elements of the syntactic category α. We assume a countable set of *labels*, ranged over by letter l, used to name functions within modules, a countable set of *module names*, ranged over by letter X, and a countable set of *variables* ranged over by letter x. The sets of labels, module names, and variables are pairwise disjoint. Variables stand for communicated values (*e.g.* basic values and anonymous modules) in a given program context.

A network is a flat, unstructured collection of sensors S combined using the parallel composition operator. The empty network is represented by symbol $\mathbf{0}$. We assume the sensors to be immersed in a (scalar or vector) field representing some physical quantity we want to monitor (*e.g.* temperature, pressure, humidity) in space. The sensors are able to measure field's intensity by calling appropriate functions in their code modules. They are also parametric in their position, p, and in their battery charge, e. The position is given in some coordinate system and may vary with time. As a first approach we use the sensor battery just to control whether the sensor can perform an operation. It is possible to refine the semantics to provide a finer grained control of resource usage in the sensors, namely, battery, memory, and cpu usage.

A sensor $[\vec{P} \triangleright \vec{C}]_e^p$ represents an abstraction of a physical sensing device running a sequence of processes $\vec{P}$ and with a collection $\vec{C}$ of code modules. Each code module in $\vec{C}$ consists of an independent name space, providing a collection of functions that implement sensor behavior. The code $l = (\vec{x})P$ represents a function with name l, parameters $(\vec{x})$ and body P. Intuitively, the collection $\vec{C}$ of modules of a sensor may be interpreted as the modules of a tiny operating system installed in the sensor at boot time plus the modules that are dynamically uploaded to the sensor.

Processes are ranged over by P. A function call, $v.l(\vec{v})$, calls function l (with arguments $\vec{v}$) in some value v. Value v must always evaluate to a code module. Calls to functions in other sensors in the network are written with the process **transmit** $X.l(\vec{v})$, which broadcast a calls to function l in a module X, with arguments $(\vec{v})$. Installing or replacing modules in a sensor can be done with the construct **install** (v), which adds the module v to the local collection $\vec{C}$. The **let** construct allows programs to create local variables to hold intermediate values in computations. In particular, it allows the construction of arbitrarily complex data structures when combined with the appropriate functions in the sensor's code modules.

We do not have a primitive sequential composition construct for programs. Such a construct can be easily obtained as syntactic sugar for: **let** $x = P$ **in** $P' \equiv P; P'$ where $x \notin \text{fv}(P')$. The semantics of the calculus forces the evaluation of P first and then P' exactly, since x does not occur free in P'. We make frequent use of this construct to impose a more imperative style of programming.

Values are the data exchanged between sensors and comprise basic values that can intuitively be seen as the primitive data types supported by the sensor's hardware, and anonymous modules.

A Simple Example. We start with a very simple ping program. We implement two Ping modules: one for the anonymous sensors in the network and another for the sink. The interface of the modules are the same but the implementation of the functions differs, reflecting the distinct behavior of each type of node. As for anonymous sensors function ping, when called with a time stamp, transmits a forward call to the network with its MAC address mac and the original time stamp. The function also re-transmits another ping call to model propagation in the network. The function forward just forwards the call. The sink has a distinct implementation of function forward. Any incoming call gets another time stamp and logs the MAC address with the round trip time. The sink does not require a ping function, and so it implements it as the default function that just returns an empty module. For simplicity we omit such functions in our programming examples.

```
[ install ( Ping ::                    // sink
    { forward = (then,mac)
        let now = System.getTime() in
        System.log(now−then,mac)         } );
  transmit System.deploy ( Ping ::
    { ping = (t)
        let mac = System.getMacAddress() in
        transmit Ping.forward(t,mac);
        transmit Ping.ping(t)
      forward = (t,m)
        transmit Ping.forward(t,m)       } );
  let now = System.getTime() in
  transmit Ping.ping(now)                      ]
|
[ install ( System ::                  // sensor
    { deploy = (x)
        install (x);
        transmit System.deploy(x)        } )   ]
|...|
[ install ( System ::                  // sensor
     { deploy = (x)
         install (x);
         transmit System.deploy(x)       } )   ]
```

Each anonymous sensor starts by installing a system module. We may think this operation is part the booting process of the sensors. Function deploy takes the module given as argument and installs it locally, while propagating the call to the network. In the sink the process is more involving. It first installs its local version of the Ping module and then deploys the Ping module for the sensors. The code in the Ping module of the sensors is activated by the call to Ping.ping from the sink node. So, the overall result of the call **let** now = System.getTime() **in transmit** Ping.ping(now) in the sink is that all reachable sensors in the network will, in principle, receive this call and will transmit their MAC addresses to the network in forward messages. These values eventually reach the sink and get logged with the associated round trip times.

Semantics. The calculus has two variable bindings: the **let** construct and function definitions. The displayed occurrence of variable x is a *binding* with *scope* P both in **let** $x = P'$ **in** P and in $l = (\ldots, x, \ldots)P$. An occurrence of a variable is *free* if it is not in the scope of a binding. Otherwise, the occurrence of the variable is *bound*. The set of free variables of a sensor S is referred as $\text{fv}(S)$.

$$S_1 \mid S_2 \equiv S_2 \mid S_1, \quad S \mid \mathbf{0} \equiv S, \quad S_1 \mid (S_2 \mid S_3) \equiv (S_1 \mid S_2) \mid S_3$$

(S-MONOID-SENSOR)

$$[\vec{P} \triangleright \vec{C}]^p_e \equiv [\vec{P} \triangleright \vec{C}]^p_e\{\mathbf{0}\}$$

(S-INIT-TRANSMIT)

Figure 2. Structural congruence for sensors.

Following Milner [15] we present the reduction relation with the help of a structural congruence relation. The structural congruence relation $\equiv$, depicted in Figure 2, allows for the manipulation of the syntactic structure of terms, making it possible for sub-terms to reduce. The relation is defined as the smallest congruence relation on sensors closed under the rules given in Figure 2.

The parallel composition of sensors is commutative and associative with **0** as the neutral element (*vide* Rule S-MONOID-SENSOR). When a sensor transmits a message it uses a conceptual *membrane* to engulf the sensors as they become engaged in communication. This abstraction prevents sensors from receiving duplicate copies of the message during transmission. The Rule S-INIT-TRANSMIT prepares a sensor for a transmission by creating such a membrane.

The reduction relation on networks, notation $S \to S'$, describes how a sensor S can evolve (reduce) to sensor S'. Reduction in a sensor occurs at the head of the sequence $\vec{P}$. In other words, in a sequence $P, \vec{P}$, program P is running while those in $\vec{P}$ are waiting in a queue. Since processes evaluate to values in the calculus, we must allow for reduction within the **let** construct. In other words, the P in the example above can be of the form **let** $x = P'$ **in** P''

and we allow the reduction *in situ* of P'. Naturally, we may have multiple levels of **let** constructs involved. For this reason we present our reduction relation using reduction contexts, or places where reduction may occur. These contexts, denoted $\mathcal{C}[\![]\!]$, are defined as follows:

$$\mathcal{C}[\![]\!] \ ::= \ [\,] \ | \ \textbf{let } x = [\,] \textbf{ in } P$$

Thus, $\mathcal{C}[\![P]\!]$ denotes the process P inserted in the [ ] hole of any of the above contexts.

The reduction relation is inductively defined by the rules in Figure 3, is parametric on two constants $\mathsf{e_{in}}$ and $\mathsf{e_{out}}$ that represent the amount of energy consumed when performing internal computation steps ($\mathsf{e_{in}}$) and when transmitting messages ($\mathsf{e_{out}}$). This is a very basic power checking mechanism, which can be significantly refined to simulate power management schemes, such as powering-off sensors or putting them off-line temporarily. However this is not the focus of this paper.

$$\frac{i \in I \qquad e \geq \mathsf{e_{in}}}{[\mathcal{C}[\![X_i.l(\vec{v})]\!], \vec{P} \triangleright \vec{C}]^p_e \to [\mathcal{C}[\![M_i.l(\vec{v})]\!], \vec{P} \triangleright \vec{C}]^p_{e'}} \quad (\text{R-MODULE})$$

$$\frac{i \notin I \qquad e \geq \mathsf{e_{in}}}{[\mathcal{C}[\![X_i.l(\vec{v})]\!], \vec{P} \triangleright \vec{C}]^p_e \to [\textbf{post } \{\mathcal{C}[\![X_i.l(\vec{v})]\!]\}, \vec{P} \triangleright \vec{C}]^p_{e'}} \quad (\text{R-NO-MODULE})$$

$$\frac{M(l) = (\vec{x})P \qquad e \geq \mathsf{e_{in}}}{[\mathcal{C}[\![M.l(\vec{v})]\!], \vec{P} \triangleright \vec{C}]^p_e \to [\mathcal{C}[\![P[\vec{v}/\vec{x}]]\!], \vec{P} \triangleright \vec{C}]^p_{e'}} \quad (\text{R-FUNCTION})$$

$$\frac{\text{inRange}(p_1, e_1, p_2) \qquad e_1 \geq \mathsf{e_{out}}}{\begin{array}{l}[\mathcal{C}[\![\textbf{transmit}\, X.l(\vec{v})]\!], \vec{P} \triangleright \vec{C}]^{p_1}_{e_1}\{S\} \mid [\vec{P}' \triangleright \vec{C}']^{p_2}_{e_2} \to \\ \quad [\mathcal{C}[\![\textbf{transmit}\, X.l(\vec{v})]\!], \vec{P} \triangleright \vec{C}]^{p_1}_{e_1'}\{S \mid [\textbf{post } \{X.l(\vec{v})\}, \vec{P}' \triangleright \vec{C}']^{p_2}_{e_2'}\}\end{array}} \quad (\text{R-TRANSMIT})$$

$$[\mathcal{C}[\![\textbf{transmit}\, X.l(\vec{v})]\!], \vec{P} \triangleright \vec{C}]^p_e\{S\} \to [\mathcal{C}[\![\{\}]\!], \vec{P} \triangleright \vec{C}]^p_e \mid S \quad (\text{R-RELEASE})$$

$$\frac{e \geq \mathsf{e_{in}}}{[\mathcal{C}[\![\textbf{install}\,(C)]\!], \vec{P} \triangleright \vec{C}']^p_e \to [\mathcal{C}[\![\{\}]\!], \vec{P} \triangleright C + \vec{C}']^p_{e'}} \quad (\text{R-INSTALL})$$

$$\frac{e \geq \mathsf{e_{in}}}{[\mathcal{C}[\![\textbf{let}\, x = v \textbf{ in } P]\!], \vec{P} \triangleright \vec{C}]^p_e \to [\mathcal{C}[\![P[v/x]]\!], \vec{P} \triangleright \vec{C}]^p_{e'}} \quad (\text{R-LET})$$

$$\frac{e \geq \mathsf{e_{in}}}{[\mathcal{C}[\![\textbf{post } \{P\}]\!], \vec{P} \triangleright \vec{C}]^p_e \to [\mathcal{C}[\![\{\}]\!], \vec{P}, P \triangleright \vec{C}]^p_{e'}} \quad (\text{R-POST})$$

$$[v, P, \vec{P} \triangleright \vec{C}]^p_e \to [P, \vec{P} \triangleright \vec{C}]^p_{e'} \qquad [\vec{P} \triangleright \vec{C}]^p_e \to [\vec{P} \triangleright \vec{C}]^{p'}_{e'} \quad (\text{R-NEXT,R-MOVE})$$

$$\frac{S \to S'}{S \mid S'' \to S' \mid S''} \qquad \frac{S_1 \equiv S_2 \qquad S_2 \to S_3 \qquad S_3 \equiv S_4}{S_1 \to S_4} \quad (\text{R-NETWORK, R-CONGR})$$

Where $\vec{C}$ stands for the sequence of modules $X_i :: M_i, i \in I$.

Figure 3. Reduction semantics for sensors.

A process P in a sensor $[\vec{P} \triangleright \vec{C}]^p_e$ may: (a) call a function in one of the modules in $\vec{C}$ (Rules R-MODULE and R-NO-MODULE), in anonymous modules (Rule R-FUNCTION), and in remote modules (Rules R-TRANSMIT and R-RELEASE); (b) install new modules in the sensor (Rule R-INSTALL); (c) compute intermediate values and assign them to new variables (Rule R-LET), and (d) schedule a process for execution at the end of the queue (Rule R-POST).

A call to a function l with arguments $\vec{v}$ in a module named X or in an anonymous module M, such that $l = (\vec{x})P$, results in the process P where the variables in $\vec{x}$ are replaced with the values $\vec{v}$. Traditionally, typed programming languages use a type system to ensure that there are no calls to undefined functions, ruling out all other programs at compile time. We also adhere to this principle. However, here we introduce an extra degree of flexibility. When a module X containing the function l being called is not installed in the sensor, we keep the call active waiting for the module to be installed (see Rule R-NO-FUNCTION). At run-time, another possible choice would be to simply discard invocations to functions on un-installed modules. Our choice to make function calls wait for the module to be installed aims to provide a more flexible programming model when coupled with the procedures for deploying code in a sensor network. We envision that if we call a function in the network on a module that has been deployed (*vide* the *Ping* example), some sensors may receive the function call before the code is actually installed locally. With the semantics we propose, the call actively waits for the code to be installed. Calling an undefined function in an anonymous module causes the process to get *stuck*.

Sensors communicate with the network by transmitting messages. A message consists of a remote function call on unspecified sensors in the neighborhood of the emitting sensor. In other words, the messages are not targeted to a particular sensor (there is no peer-to-peer communication). Several metrics may be used to define the network neighborhood of a sensor, *e.g.* based on the geometric distance between the sensors or on the received power at the destination node. Here, we use some abstract metric based on the positions of the sensors and on the battery power of the transmitting sensor. A message transmitted from a sensor may not reach all the sensors in its neighborhood. There might be, for instance, landscape obstacles that prevent two sensors, otherwise in range, from communicating with each other. Also, during a transmission operation the message must reach each neighborhood sensor at most once. Notice that we are not saying that the same message can not reach the same sensor multiple times. In fact it might, but as the result of the echoing of the message in subsequent transmissions. A transmission starts with the application of Rule S-INIT-TRANSMIT, proceeds with multiple (eventually none) applications of Rule R-TRANSMIT (one for each target sensor), and terminates with the application of

Rule R-RELEASE. Rule R-TRANSMIT calls a function $X.l$ in the remote sensor, provided that the emitting and the receiving sensors are *in range*. Each sensor that receives the call is put in the membrane associated with the emitting sensor, thus preventing multiple deliveries of the same message during the transmission. Observe that the rule does not enforce the interaction with all sensors in the neighborhood of the emitting sensor. Rule R-RELEASE consumes the operation (**transmit** $X.l(\vec{v})$) and dissolves the membrane, thus finishing the transmission.

Installing a module $X :: M$ in a sensor $[\vec{P} \triangleright \vec{C}]^p_e$ amounts to adding the module to $\vec{C}$. If the module already exists in $\vec{C}$, $X :: M'$, then the new code replaces the previous version such that: $C(X) = M + M' = (M \setminus M') \cup M'$.

A process **post** $\{P\}$ schedules the process P for execution at the end of the run-queue and allows the next process in the run-queue to execute. This feature is essential to ensure that sensors eventually process incoming network communication and that no application takes control of the sensor. This, however, must be taken into consideration by the programmer.

3 Programming Examples

The following examples assume that the *network layer* of the sensor network manages the distribution of messages in an energy-efficient way and filters out redundant messages using, for example, *multicast* protocols. The network layer can be programmed with CSN but here we concentrate on higher level abstractions.

Sampling the Network. In this example the sensors are requested to take field readings and report them to a sink for processing purposes.

```
[ install ( Sample ::                    // sink
    { sample = ()
        transmit Sample.sample();
        post { Sample.sample(t) }
      forward = (v,m,t)
        System.log(v,m,t)                 } );
  transmit System.deploy ( Sample ::
    { sample = ()
        let value = System.getReading() in
        let time  = System.getTime() in
        let mac   = System.getMacAddress() in
        transmit Sample.forward(value,mac,time);
        transmit Sample.sample();
      forward = (v,m,t)
        transmit Sample.forward(v,m,t)    } );
  Sample.sample()
]
|
[ install ( System :: {deploy = ...} ) ] // sensor
|...|
[ install ( System :: {deploy = ...} ) ] // sensor
```

We program the application from the bootstrap point. After the Sample module has been uploaded to the sensors, the sink just calls the function Sample.sample to begin the sampling cycle. The call to Sample.sample in any sensor propagates the call to the network neighborhood; then it calls sends to the network a call to the function forward with the values of the reading taken, the sensor's MAC address and the local time. The last two items are important since they allow the sink to identify the streams associated with each sensor.

Roaming Sensors in a Museum. This example illustrates the use of several sinks in a sensor network coupled with roaming sensor nodes. Each sink is imagined as a device that stores information about an item of a museum collection in a module Provider and that is physically located near that item. Sinks can be connected to a Museum-wide network (*e.g.* via WiFi) for easy centralized information update. Visitors carry small devices (e.g., BlueTooth based) that when within range of a sink and when the user presses the key "G" (an event handled by function GUI.buttonG in the user interface), read the data provided by the sink and present it to the user.

```
[ install ( Provider ::              // sink
    { getInfo = ()
        transmit Info.getInfo ( Info ::
         { printInfo = ()
            GUI.display("Title ..."); } ); } ); ]
| ... |
[ install ( Provider ::              // sink
    { getInfo = ()
        transmit Info.getInfo ( Info ::
         { printInfo = ()
            GUI.display("Title ..."); } ); } ); ]
|
[ install ( Info ::                  // sensor
    { getInfo = (x)
         install (x); Info.printInfo()     } );
  install ( GUI ::
    { buttonG = ()
         transmit Provider.getInfo();
      display = (x) ...
      main    = () ...                     } );
  GUI.main();                                  ]
```

The call to Provider.getInfo deploys the module Info, that encapsulates the data for the Museum item, to the mobile device. Once the module arrives at the device, it is installed and the call Info.printInfo displays the retrieved information.

Secure Code Deployment. In the previous examples the code deployment is inherently insecure. In fact, any sensor in the network or even external entities might send deploy calls to the network and install modules, unchecked. Here we show how we can prevent this problem assuming that the sink and the remainder of the sensors share a *secret key*. The distribution of the keys is assumed to be done using some currently available secure scheme for wireless sensor networks, *e.g.* trusted third party [19], public key [13] or key pre-distribution [4, 6, 21].

We implement a *secure deploy* function, secureDeploy, that receives one argument, assumed to be a module issued from the sink and encrypted using the shared secret key, key.

```
[ install ( Ping ::                    // sink
    { forward = (t,m) ... } );
  let key  = System.secretKey() in
  let emod = System.encrypt(k,Ping :: {...}) in
  transmit System.secureDeploy(emod);
  transmit Ping.ping()                          ]
|
[ install ( System ::                  // sensor
    { secureDeploy = (x)
        let key = System.secretKey() in
        install System.decrypt(k,x);
        transmit System.secureDeploy(x) } );    ]
|...|
[ install ( System ::                  // sensor
    { secureDeploy = (x)
        let key = System.secretKey() in
        install System.decrypt(key,x);
        transmit System.secureDeploy(x) } );    ]
```

The function attempts to decrypt the argument using the key and the resulting module is installed locally. However, if the sender is not the sink, decrypt returns an empty module which, although installed, does not alter the code in the sensor. The sink simply encrypts the Ping module (in this example), deploys it to the network and triggers its execution.

4 Towards a Higher-Level Language

The examples presented in the previous sections are quite low-level, in the sense that the programmer must to some extent be aware of the communication and routing requirements of the applications. Take the *Ping* example for instance, each call to the function Ping.ping() must explicitly propagate itself to the remainder of the network by repeating the call **transmit** Ping.ping() in the body of the call. As we mentioned in Section 1, more data-centric programming model for sensor networks, one that abstracts away from the communication details is a very desirable feature. In this section we describe how we can hide the network communication in CSN programs and create a higher level idiom for programming sensor networks. We do this by resorting to derived constructs that are pre-processed to CSN programs, and thus keep the original semantics of the calculus.

First we introduce a new data sending primitive, **send** $X.l(\vec{v})$, that is used instead of our **transmit** primitive. The new primitive is a higher-level remote call that can be customized for each type of sensor in the network. We implement **send** $X.l(\vec{v})$ as:

```
transmit System.send({execute = () X.l(v⃗)})
```

So, the new primitive simply calls a system level function, System.send(). In a sink node, this function would simply take its argument, an anonymous module that encapsulates the original remote call, and execute it. No re-transmission is required:

```
send = (x) x.execute()
```

In a sensing node however, re-transmitting the call is required and so the definition would be:

```
send = (x) x.execute(); transmit System.send(x)
```

In view of these adjustments, the *Ping* example would now be written as follows:

```
Sink [
  install ( Ping ::
    { forward = (then,mac)
        let now = System.getTime() in
        System.log(now−then,mac)        } );
  send System.deploy ( Ping ::
    { ping = (t)
        let mac = System.getMacAddress() in
        send Ping.forward(t,mac)
      forward = (t,m) {}                } );
  let now = System.getTime() in
  send Ping.ping(now)                       ]
|
Sensing [
  install(System::{deploy=(x)install(x)}) ]
|...|
Sensing [
  install(System::{deploy=(x)install(x)}) ]
```

The keywords **Sink** and **Sensing** are there to signal the pre-processor which implementation of send() should be attached to the System module for each sensor. Note that the deploy is also simplified by this procedure. As in the *Ping* example we could further simplify the code by omitting the forward function from the Ping module of the sensing devices, since it just returns an empty module.

Further customization of the System.send() function allows all sorts of operations to be performed on the data sent in the call (e.g., encryption) and even the implementation of more complex routing protocols. Using this approach we could re-write the *secure deployment* example from the previous section *without* the need for a secureDeploy function, by just customizing the communication function send to encrypt the data. The code would be written as follows:

```
Sink [
  install ( Ping :: { forward = (t,m) ... } );
  send System.deploy(Ping :: {...});
  send Ping.ping()                            ]
|
Sensing [
  install(System::{deploy=(x)install(x)}) ]
|...|
Sensing [
  install(System::{deploy=(x)install(x)}) ]
```

The assumed implementation for **send** $X.l(\vec{v})$ hides the encryption/decryption of the communicated data as follows:

```
send(X, l ,v⃗) ≡
    let key  = System.secretKey() in
    let mod  = {execute = () X.l(v⃗)} in
    let emod = System.encrypt(key, mod) in
    transmit System.send(emod)
```

At the receiving end, the implementation of the System.send function handles the decryption of the data and calls the local function X.l encapsulated in the received module:

```
System :: {
  send = (emod)
    let key = System.secretKey() in
    let mod = System.decrypt(key, emod) in
    mod.execute();
    transmit System.send(emod)
}
```

5 The Type System

In this section we present a simple type system for CSN, discuss run-time errors, and prove a type safety result guaranteeing that a well-typed sensor network does not get "stucked" while computing.

Type checking. The syntax for types is depicted in Figure 4. Types τ are built from the built-in type β using the constructors for the type of anonymous modules $\{l_i\colon \vec{\tau_i} \to \tau_i\}_{i \in I}$ and for the type of modules $X :: \{l_i\colon \vec{\tau_i} \to \tau_i\}_{i \in I}$. A type $\{l_i\colon \vec{\tau_i} \to \tau_i\}_{i \in I}$ describes an anonymous module represented as a collection of (distinctly named) functions. Each function l_i has type $\vec{\tau_i} \to \tau_i$, where $\vec{\tau_i}$ is the type of the parameters of the function and τ_i is its return type. A module's type $X :: \{l_i\colon \vec{\tau_i} \to \tau_i\}_{i \in I}$ records the name of the module (X) together with the type of the anonymous module that constitutes it. For instance, type $Ping :: \{ping\colon \beta \to \{\}, forward\colon \beta\beta \to \{\}\}$ is the type of module $Ping$ presented in Section 2. It represents a module named $Ping$ with two functions named $ping$ and $forward$. Function $ping$ has a parameters (a timestamp) and returns an empty module (the result from the final $Ping.ping()$ operation). Function $forward$ accepts two built-in values and returns $\{\}$, like $ping$.

τ ::=	*Types*
β	built-ins
$\mid \{l_i\colon \vec{\tau_i} \to \tau_i\}_{i \in I}$	anonymous modules
$\mid X :: \{l_i\colon \vec{\tau_i} \to \tau_i\}_{i \in I}$	modules

Figure 4. The syntax of types.

The type system is defined in Figures 5, 6, and 7. A typing Γ is a partial function of finite domain from variables and modules' names to types. We write $\mathrm{dom}(\Gamma)$ for the

$$\Gamma \vdash b\colon \beta \qquad \Gamma, x\colon \tau \vdash x\colon \tau \qquad \frac{\forall i. \Gamma \vdash v_i\colon \tau_i}{\Gamma \vdash \vec{v}\colon \vec{\tau}}$$

(T-BUILT-IN, T-VAR, T-SEQ)

$$\frac{\forall i \in I. \Gamma, \vec{x_i}\colon \vec{\tau_i} \vdash P_i\colon \tau_i}{\Gamma \vdash \{l_i = (\vec{x_i})\, P_i\}_{i \in I}\colon \{l_i\colon \vec{\tau_i} \to \tau_i\}_{i \in I}} \quad \text{(T-AMOD)}$$

$$\Gamma, X\colon \tau \vdash X\colon \tau \qquad \frac{\Gamma \vdash M\colon \tau \qquad \Gamma \vdash X\colon \tau}{\Gamma \vdash X :: M\colon X :: \tau}$$

(T-MNAME, T-MOD)

Figure 5. Typing rules for values.

$$\frac{\Gamma \vdash v\colon \{l_i\colon \vec{\tau_i} \to \tau_i\}_{i \in I} \qquad \Gamma \vdash \vec{v}\colon \vec{\tau_j} \qquad j \in I}{\Gamma \vdash v.l_j(\vec{v})\colon \tau_j} \quad \text{(T-CALL)}$$

$$\frac{\Gamma \vdash X.l(\vec{v})\colon \_}{\Gamma \vdash \textbf{transmit}\, X.l(\vec{v})\colon \{\}} \qquad \frac{\Gamma \vdash v\colon X :: \tau}{\Gamma \vdash \textbf{install}\,(v)\colon \{\}}$$

(T-TRANS,T-INST)

$$\frac{\Gamma \vdash P_1\colon \tau_1 \qquad \Gamma, x\colon \tau_1 \vdash P_2\colon \tau_2}{\Gamma \vdash \textbf{let}\, x = P_1 \textbf{ in } P_2\colon \tau_2} \qquad \frac{\Gamma \vdash P\colon \_}{\Gamma \vdash \textbf{post}\ \{P\}\colon \{\}}$$

(T-LET, T-POST)

Figure 6. Typing rules for processes.

domain of Γ. Let χ range over x and X. When $\chi \notin \mathrm{dom}(\Gamma)$ we write $\Gamma, \chi\colon T$ for the typing Γ' such that $\mathrm{dom}(\Gamma') = \mathrm{dom}(\Gamma) \cup \{\chi\}$, $\Gamma'(\chi) = T$, and $\Gamma'(\chi') = \Gamma(\chi')$ for $\chi' \neq \chi$.

Type judgments are of three forms: $\Gamma \vdash v\colon \tau$ means that value v has type τ, under the assumptions in typing Γ; $\Gamma \vdash P\colon \tau$ asserts that program P has type τ, under the assumptions in Γ; and $\Gamma \vdash S$ means that sensor network S is well typed, assuming the typing Γ.

The rules for typing values (Figure 5) are straightforward. As for processes, function calls are separated into local calls (rule T-CALL) and remote calls (rule T-TRANS). In a local call, function l_j must be part of the target module ($j \in I$), either named or anonymous, the type of the arguments must agree with the type of the parameters ($\vec{v}\colon \vec{T_j}$), and the type of the invocation (T_j) is the return type of the function. A remote call, using **transmit**, is type checked as a local call, apart from its return type that is always the empty module ($\{\}$), meaning that the return value of a remote call is ignored. This means that the interface of the sensors is identical and is fixed by the application programmer, before type checking takes place. When installing a module X (rule T-INST) its type must coincide with the type fixed for the interface of the sensor. Therefore, value v must contain a full implementation of the module being installed. The result of an **install** operation is the empty module.

Regarding the typing rules for sensors, we focus on rule T-SENSOR, as the remainder of the rules should be simple to follow. When typing a sensor we make sure that the modules available in the sensor's interface conform with the global, network-wide set interface ($\Gamma \vdash \vec{C}\colon \_$). Notice that

$$\Gamma \vdash \mathbf{0} \qquad \frac{\Gamma \vdash \vec{P} : \_ \qquad \Gamma \vdash \vec{C} : \_ \qquad \Gamma \vdash pe : \vec{\beta}}{\Gamma \vdash [\vec{P} \triangleright \vec{C}]^p_e}$$

(T-OFF, T-SENSOR)

$$\frac{\Gamma \vdash [\vec{P} \triangleright \vec{C}]^p_e \qquad \Gamma \vdash S}{\Gamma \vdash [\vec{P} \triangleright \vec{C}]^p_e\{S\}} \qquad \frac{\Gamma \vdash S_1 \qquad \Gamma \vdash S_2}{\Gamma \vdash S_1 \mid S_2}$$

(T-BSENSOR, T-PAR)

$$\frac{\Gamma \vdash P : \_ \qquad \Gamma \vdash \vec{P} : \_}{\Gamma \vdash P, \vec{P} : \_} \quad \text{(T-SEQP)}$$

Figure 7. Typing rules for sensors.

a sensor may offer just a subset of the interface modules, since some of them may not yet be available (installed) in the sensor. Nevertheless, the installed modules must be fully available.

The following result ensures that types are preserved during reduction.

Theorem 1 (Subject Reduction)**.** *If* $\Gamma \vdash S$*,* $S \to S'$*, then* $\Gamma \vdash S'$*.*

The proof proceeds by induction on the derivation tree for the reduction $S \to S'$ and is a straightforward case analysis.

Type safety. Our claim is that well-typed processes are free from run-time errors. The unary relation $S \xmapsto{\text{err}}$, defined as the least relation on networks closed under the rules in Figure 8, identifies processes that would get "stucked" during computation (reduction). We write $S \not\xmapsto{\text{err}}$ for $\neg(S \xmapsto{\text{err}})$.

Our Sensor Networks may exhibit two kinds of failures upon computing: when calling a function or when installing a module. In the former, the call may result in a run-time error when the target of the call is neither a module name, nor an anonymous module (Rule E-CALL); or when the function name is unknown or there is a mismatch between the number of arguments $(v_1 \dots v_n)$ and the number parameters $(x_1 \dots x_m)$ (Rule E-CFUNCTION). In the latter, an error may occurs if we are installing some value that is not a module (Rule E-INSTALL).

As an example, recall module $Ping$ sketched below.

```
Ping :: {
  ping = (t) ...
  forward = (t,m) ...
}
```

The sensor network

```
[let t = System.getTime() in
 transmit Ping.forward(t)]
```

exhibits a run-time error, since function forward is being called with one argument instead of two. In fact, the above network may reduce using Rule R-MODULE, but then we can not apply Rule R-FUNCTION since the substitution is not defined. Run-time error Rule E-CFUNCTION captures this kind of failures.

$$[\mathcal{C}[\![v.l(\vec{v})]\!], \vec{P} \triangleright \vec{C}]^p_e \xmapsto{\text{err}} \quad \text{if } v \text{ is not } X\text{, nor } M$$

(E-CALL)

$$[\mathcal{C}[\![M.l(v_1 \dots v_n)]\!], \vec{P} \triangleright \vec{C}]^p_e \xmapsto{\text{err}} \quad \text{if } l \notin \text{dom}(M) \text{ or } (M(l) = (x_1 \dots x_m)\,P \text{ and } n \neq m) \quad \text{(E-CFUNCTION)}$$

$$[\mathcal{C}[\![\textbf{install}\,(v)]\!], \vec{P} \triangleright \vec{C}]^p_e \xmapsto{\text{err}} \quad \text{if } v \text{ is not } C \quad \text{(E-INSTALL)}$$

$$\frac{S \xmapsto{\text{err}}}{S \mid S' \xmapsto{\text{err}}} \qquad \frac{S \equiv S' \quad S \xmapsto{\text{err}}}{S' \xmapsto{\text{err}}} \quad \text{(E-PAR, E-STR)}$$

Figure 8. Run-time errors for sensors.

The Type Safety result states that well-typed networks do not incur in run-time errors.

Theorem 2 (**Type Safety**)**.** *If* $\Gamma \vdash S$*, then* $S \not\xmapsto{err}$*.*

We prove the contrapositive result, namely $S \xmapsto{\text{err}}$ implies that $\Gamma \nvdash S$, proceeding by induction on the definition of $S \xmapsto{\text{err}}$ relation.

Finally, a well-typed network is free of security flaws, at any time during reduction.

Corollary 3. *If* $\Gamma \vdash S$ *and* $S \to^* S'$*, then* $S' \not\xmapsto{err}$*.*

The proof follows from Theorems 1 and 2.

6 The CSN Interpreter

We have implemented an interpreter for CSN in Java and use it both for debugging applications and as a testbed simulator for analyzing the behavior and performance metrics of CSN applications. The interpreter is divided in several modules for parsing, type-checking, and interpreting the applications directly from the abstract syntax tree.

A CSN network is implemented as a set of Java threads, each representing a sensor, running concurrently and communicating via shared-memory (Figure 9). Each thread manages two run-time data structures: a map that keeps track of the modules currently installed and, a queue for the processes currently scheduled for execution.

The map assigns *module names* to *modules*. Each module is itself a map that assigns *function names* to code blocks, which in this case are just *sub-trees* of the abstract syntax tree. Sensors transfer modules between them by passing a reference for the map that implements the module. A module is installed in a sensor by adding the map implementing the module to the map of installed modules. The queue is a queue of references for sub-trees of the abstract syntax tree that are scheduled to be interpreted.

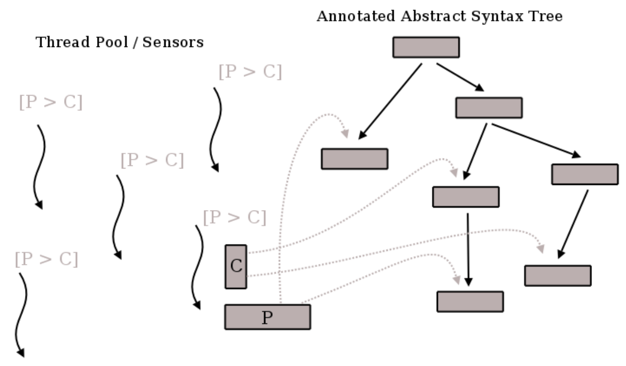

Figure 9. Run-time snapshot.

We aim to produce a machine independent, low overhead, byte code representation for CSN programs. To execute this byte-code we are designing a virtual machine based on the operational semantics of the calculus.

7 Conclusions and Future Work

Building on our previous calculus [12], CSN, we provided a strongly typed programming discipline for sensor networks. The type system provides a static verification tool, which allows for premature detection of application protocol errors. In addition, we proved two fundamental properties of the operational semantics and of the type system, namely, *subject reduction* and *type safety*. Together, these results establish the calculus as a sound framework for developing programming languages for sensor networks.

As part of our ongoing work, we are pursuing two different lines of research. First, we are exploring the theoretical properties of the calculus. By applying techniques from process calculi theory we hope to be able to prove a set of fundamental *laws* about sensor networking applications and protocols. Secondly, we are targeting an implementation of the model for actual systems. So far, we have a working simulator that allows us to experiment with programming virtual networks.

Acknowledgements. This work has been partially funded by project CALLAS of the Fundação para a Ciência e Tecnologia (contract PTDC/EIA/71462/2006).

References

[1] The TinyOS Documentation Project. Available at `http://www.tinyos.org`.

[2] I. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. A Survey on Sensor Networks. *IEEE Communications Magazine*, 40(8):102–114, 2002.

[3] D. E. Culler and H. Mulder. Smart Sensors to Network the World. *Scientific American*, 2004.

[4] W. Du, J. Deng, Y. Han, P. Varshney, J. Katz, and A. Khalili. A Pairwise Key Predistribution Scheme for Wireless Sensor Networks. *ACM Transactions on Information and System Security*, 8(2):228–258, 2005.

[5] A. Dunkels, B. Grönvall, and T. Voigt. Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors. In *EmNets'04*, 2004.

[6] L. Eschenauer and V. Gligor. A Key-Management Scheme for Distributed Sensor Networks. In *CCS'02*, pages 41–47. ACM Press, 2002.

[7] C.-L. Fok, G.-C. Roman, and C. Lu. Rapid Development and Flexible Deployment of Adaptive Wireless Sensor Network Applications. In *ICDCS'05*, pages 653–662. IEEE Press, 2005.

[8] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC Language: A Holistic Approach to Network Embedded Systems. In *PLDI'03*, pages 1–11. ACM Press, 2003.

[9] K. Honda and M. Tokoro. An object calculus for asynchronous communication. In *ECOOP'91*, number 512 in LNCS, pages 133–147. Springer-Verlag, 1991.

[10] J. W. Hui and D. Culler. The Dynamic Behavior of a Data Dissemination Protocol for Network Programming at Scale. In *ENSS'04*, pages 81–94. ACM Press, 2004.

[11] P. Levis and D. Culler. Maté: A Tiny Virtual Machine for Sensor Networks. In *ASPLOS X*, pages 85–95. ACM Press, 2002.

[12] L. Lopes, F. Martins, M. S. Silva, and J. Barros. A Process Calculus Approach to Sensor Network Programming. In *SENSORCOMM'07*. IEEE Press, 2007.

[13] D. Malan, M. Welsh, and M. Smith. A public-key infrastructure for key distribution in TinyOS based on elliptic curve cryptography. In *SECON'04*. IEEE Press, 2004.

[14] N. Mezzetti and D. Sangiorgi. Towards a Calculus for Wireless Systems. In *MFPS'06*, volume 158 of *ENTCS*, pages 331–354. Elsevier Science, 2006.

[15] R. Milner. A Calculus of Communicating Systems. Number 92 in LNCS. Springer-Verlag, 1980.

[16] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, (Parts I and II). *Information and Computation*, 100:1–77, 1992.

[17] R. Newton and M. Welsh. Region Streams: Functional Macroprogramming for Sensor Networks. In *DMSN'04 Workshop*, 2004.

[18] K. Ostrovský, K. V. S. Prasad, and W. Taha. Towards a Primitive Higher Order Calculus of Broadcasting Systems. In *PPDP'02*, pages 2–13. ACM Press, 2002.

[19] A. Perrig, R. Szewczyk, J. Tygar, V. Wen, and D. Culler. SPINS: Security protocols for sensor networks. *Wireless Networks*, 8(5):521–534, 2002.

[20] K. V. S. Prasad. A Calculus of Broadcasting Systems. In *TAPSOFT'91*, number 493 in LNCS, pages 338–358. Springer-Verlag, 1991.

[21] S. Zhu, S. Setia, and S. Jajodia. LEAP: efficient security mechanisms for large-scale distributed sensor networks. In *CCS'03*, pages 62–72. ACM Press, 2003.