Programming Language Support for
Sensor Networks

Yu David Liu
The Johns Hopkins University
Baltimore, MD 21218, USA
yliu@cs.jhu.edu

1 Introduction

In this project, we design new language abstractions in better support of sensor network programming. Sensor network has received wide attention from both academia and industry in recent years. As a somewhat entertaining testament, a recent special issue of TIME magazine (“WHAT’S NEXT”, Fall 2003) has singled out sensor network as one of the few technologies ready to bring revolution to the high-tech world. The fast growth of sensor networks demands good support for the design of sensor network applications. However, as a new computing infrastructure, sensor network carries some traits strikingly different from traditional networks, and they put forward new requirements and challenges in the realm of software design.

1.1 New Requirements

Our language is especially motivated by the following requirements distinctive in the design of sensor network applications:

First of all, sensor network applications are typically straightforward for intra-node data processing, but complicated for inter-node communication. Each node in a sensor network normally does not achieve a task alone; instead, they collaborate via complicated communications to achieve ambitious goals. To write a sensor network application, it is vital for software developers to have a clear understanding of what kind of communications each piece of software is involved, which will help lead to less error-prone software development.

Second, sensor network nodes are energy and memory constrained. Sensor network nodes are typically wireless nodes powered by battery. Depending on applications, they might be scattered in battlefields, inside volcanos, or lion-roaming prairies. Once deployed, it is usually hard to recharge or change battery. In terms of memory, a sensor network is typically composed of numerous nodes, and each node has to be installed with limited memory to keep the cost of the whole network low.
Third, sensor networks are densely populated and in a typical communication scenario, it is less important to ask the question “which physical node am I going to talk with?”, than the question “what kind of node am I going to talk with?” In fact, for many sensor applications, software developers can not anticipate the final deployment and topology of the whole network at all, therefore it is in many case unwise to hard-code applications like “communicating with node with certain MAC address”.

1.2 Our Approach

In this project, we propose a programming language for sensor networks, together with a lightweight design of a virtual machine on top of which the compiled form of our programming language is interpreted. It has the following novel features to adapt to the need listed earlier:

Communication Interface In our language, each unit of software running on a sensor network node is associated with a list of connectors, which serves as communication interface for the software unit. All communications involved by the software unit is through the interface. Thus, simply by looking through what connectors each software unit has, we can easily know how the unit communicates with the outside, which is especially helpful for developing sensor network applications, where inter-node communications are complicated. In addition, since inter-node communications consume far more energy than intra-node computations (transmitting a single bit of data is equivalent to 800 instructions [MFH02]), a clear specification of communication interfaces also helps developers find possible optimizations to save energy. With connectors, inter-node communications thus become a problem of connecting compatible connectors of two software runtime on different nodes. Readers should note our connector design is conceptually different from interfaces proposed by nesC [GLvB+03]; a comparison is followed in Sec. 1.3.

Dynamic Module Loading and Unloading In a sensor network composed of hundreds of nodes, it is quite usual some of the nodes are not necessary; however, due to difficulty of deployment, this kind of redundancy does happen, or even unavoidable. For instance, a battlefield surveillance sensor network might be deployed by an airplane and it is not possible to reach the optimum desired deployment. A more difficult scenario is that sensor network nodes are sometimes in motion: a good example might be a sensor network project for understanding the habit of zebras on African grasslands. It is desirable if some algorithm can detect the necessity of each node, and when it is not necessary, the application module is unloaded from the virtual machine, and when it is necessary, the application is reloaded. Our language supports dynamic loading of modules and unloading of its runtime forms.
**Runtime Migration Support**  As energy depletion is the Achilles’ heels of sensor network, we not only need to consider the question about how to avoid energy depletion, but also do we need to consider what can be done when energy depletion does happen. For instance a relay-race-like strategy can be proposed: when a module runtime notices itself sitting on a soon-to-be-depleted node, it can choose to migrate to another node with enough power. A sensor network can even be intentionally equipped with some redundant nodes that are running minimum software, such as only the operating system and virtual machine, at the very beginning, and then became the destination of the module runtimes whose master nodes will soon wear out the power. Strategies like these can effectively lengthen a sensor network’s functioning time. Our language provides a language construct to help with the runtime migration process. Readers should note our migration support is conceptually different from code replication proposed by Mate [LC02]; in our case, the states of the module runtime will also be carried over. Besides, our language ensures that all previous connections are kept alive even when a module runtime moves from one physical location to another. A comparison with Mate is followed in Sec. 1.3.

**Location-Carefree Connection**  When a node needs to establish a communication, our language provides a language mechanism, location-carefree connection, to allow programmers not to specify which physical node the current software is to communicate with, but to specify what kind of connector the other node should have. Connector as an interface of the service provider, better captures the intention of the initiator. Any node with compatible connector could respond and set up the connection.

**Modular Design**  Similar to nesC and many others, our language also provides support for writing software in smaller pieces of modules. Sound and simple static linking language constructs enable modules to be pieced together easily. Thus, deployers can customize the software for each node, and only take the few absolutely necessary for the correct functioning of the node. The resulting software will usually be small, and hence consume less energy and memory space.

**1.3 Related Work and Comparison with our Project**

Among various research projects on sensor networks, the ones that are closest to ours are nesC and Mate, both of which are programming language efforts to support sensor networks. We here give a brief overview for each of the systems; the emphasis will be put on comparison with our language. We hope, through comparison, a few fundamental and important concepts of our language can be made clear.

**nesC Programming Language**  nesC proposed a module system supporting bi-directional static linking, where each module can specify what it imports
and what it exports, and modules with complementary needs can be statically linked together. An interesting aspect of this module system is its proposal on split-phase operations, where imported/exported functions on nesC module’s interfaces do not return values, except the success indication with type result_t. Invocation to these functions will follow asynchronous semantics and returns immediately. Long latency tasks, such as sensing, can thus be issued without program-blocking or complicated thread context switching, which otherwise would be too costly to be implemented in memory and energy constrained sensor networks. Unlike other projects on asynchronous methods where returned values are still expected and will be joined with the invoking program eventually [JS00], nesC depends on programmers to program callback functions explicitly when long latency tasks are completed and results are necessary to get sent back. For instance, if a module intends to import a temperature sensing function which triggers the sensing module and returns the current temperature, it instead split-phases the function into two:

- import function: result_t sense();
- export function: result_t senseDone(TempType t);

Thus in the temperature sensing module, when the sensing procedure is over, it will trigger its imported function senseDone; the callback is thus fulfilled.

Though nesC provides an appealing design to promote software modularity, its heavy dependence on static linking makes it unable to model inter-node communication on module interfaces. Thus, even though nesC module interfaces can successfully model the interactions between various components of TinyOS [HSW’00], a sensor network operating system running on top of one sensor node, its module interfaces can not even model the simplest application where two sensor nodes are communicating with each other. In a way, static linking excludes distributed communication.

To understand why this happens, we need first to make clear the important difference between modules and module runtimes. Modules are code pieces without any states; even though modules can be combined together via static linking, the results of static linking are still stateless code pieces. It is only when modules are loaded into memory, their corresponding stateful module runtimes are created. In static linking case, even though two component modules could be stored at different locations, the resulting composed module will create only one module runtime sitting on one node when the module runtime is created. Note that static linking is between modules, not module runtimes. On the other hand, inter-node communication happens between module runtimes, not modules. Module interfaces for inter-node communication need to assume a form of late-binding.

In this project, we design a new category of interfaces for modules, called connectors, to deal with inter-node communication. Thus, inter-node operations can appear on connector interfaces (which are also bi-directional), and programs can dynamically control the connection and disconnection between sensor nodes via complementary connectors. Static linking are also supported
for similar reasons as nesC suggests, via a different interface category static link-ers. Split-phase operations are used for functions defined in connectors, since they potentially lead to inter-node communications typically in long latency. They are not necessary in static linkers, since all computations through static linkers happen on one network node, but programmers can implement them in the same way if needed.

**Mate Virtual Machine** Mate proposed a virtual machine design for sensor networks and an intermediate bytecode-like language interpreted by its virtual machine. In Mate, code is broken up into small capsules of 24 instructions, which can self-replicate through the network. Basic communication primitives, like `send` and `receive` are provided by Mate’s instruction sets, or can be easily implemented by Mate. At any given moment, a Mate virtual machine only allows one application running on top of it.

Our project also proposes a virtual machine design, where a list of system services are defined. These services play a part in defining the meaning of some of our language abstractions. Instead of proposing a virtual machine design superseding Mate, our virtual machine is built on it, which conceptually can be considered as a higher level on top of Mate, defining higher level primitives (services) that can be consequently translated to Mate instructions at static time. Note that this conceptual extra layer does not incur extra runtime overhead. Readers are also encouraged to think of our virtual machine design is simply a macro-definition layer with routines written in Mate.

Mate does provide a notion of migration, but in its case only code migrates from one sensor network node to another, not the application. Important issues like state transferring are ignored. Mate code migration is more out of the purpose of code deployment, while ours is more for increasing the longevity of an energy-constrained sensor network.

**Other Related Work** Module systems and modular software design have been popular in programming language and software engineering community for decades. Calculi and programming languages dealing with static linking are numerous. Influential ones include mixins [BC90] and Units [FF98]. The underlying core theory of our static linking mechanism is similar, although different systems may have slightly different syntaxes, semantics and expressivity. In recent years, it has also had impact on systems research, where many operating system and virtual machine designs also follow the modular approach, in which case system software is coded as the composition of several smaller module pieces. Examples of programming language efforts in support of this goal include Knit [RFS+00] and nesC [GLvB+03].

Traditionally, low-level inter-node communications can be modeled using socket programming, where message sending and receiving can be coded. These language abstractions are generally considered too low-level for application-level programming. Many errors that should have been able to get detected by static type systems could now only be discovered at runtime when problems like mes-
sage mismatch happens; these situations are typically bad for sensor networks since they are in many cases unreclaimable after being deployed; a runtime error could send the whole network to non-repairable paralysis.

Systems like RPC and Java RMI are also widely used, but these systems are typically synchronous, where sending a remote invocation could either mean a long blocking, or a costly thread context switch. More importantly, these systems almost always merit interface contracts one-directionally, which makes callback functions a difficult topic. Consider the following example. If invoker A invokes $\text{getTemperature}()$ of B, only B’s contract is agreed upon: function $\text{getTemperature}$ with no argument is an export of B. Suppose now function $\text{getTemperature}$ assumes a callback function exported by A, the communication would fail if A does not provide one. In a typical network communication case, we believe the successful communication between A and B is actually hinged on the bi-directional mutual contract satisfaction.

2 Informal Discussion

In this section, we informally discuss the various features of our programming language, with an emphasis on features that distinguish ours from existing ones.

2.1 Architectural Overview of Cell-based Sensor Networks

The Grand Picture From the perspective of application developers, the software on top of each sensor network node has a three-tier architecture: operating system, virtual machine and application. Each operating system has one virtual machine running on top of it, and each virtual machine can have multiple sensor network applications running on top. Each application programmed in our language is wrapped up into several well-encapsulated code piece entities which we call modules. When an application starts up, its composing modules are loaded and a single runtime is created, which we call a cell. There is a one-to-one correspondence between applications and cells. However, each cell can have multiple modules combined as its code base. Globally, a collaboration sensor network task can be viewed as the communication process among multiple cells.

A typical communication process goes as follows: If a cell $\text{Ca}$ needs to communicate with another cell $\text{Cb}$, its request will first be interpreted by its own virtual machine, and the virtual machine will then invoke low-level operating system protocols (such as MAC protocols) to send the request to the sensor network node where $\text{Cb}$ is located. After the request is received by $\text{Cb}$’s operating system, it is then forwarded to the virtual machine holding $\text{Cb}$, and subsequently to $\text{Cb}$.

Virtual Machine as Middle Layer Our primary goal being a programming language design, questions might be asked why we also design a virtual machine as the middle layer. The fundamental reason is that virtual machines provide an abstract specification on what our language expects from low-level system
software. To define a programming language, the first question will be how each source language expression is translated into some machine-understandable code. As an extreme, we could define the meaning of a communication expression like send as a bit-by-bit invocation to some device drivers of network cards, but an approach like this would highly depend on system software and hardware, making the language only implementable in a specific software and hardware environment. With virtual machines, our source language can just be translated to the intermediate language defined by the virtual machine. How the virtual machine is implemented and how the intermediate language is mapped to a specific platform can thus be factored out as separate issues. Programming language design with virtual machines as the middle layer is fairly common; Java Virtual Machine [LY99] is one of the most famous ones.

On the technical side, a design with virtual machines alleviate the burden of cell migration. The intermediate language defined by the virtual machine offers higher level language primitives than binary instruction sets, which means reducing the size of code significantly during cell migration, a crucial factor related to energy consumption. Besides, portability is also achieved if different sensor nodes might run different operating systems or hardware.

In this report, we do not give a full-fledged definition (like bytecode definition of Java) to the intermediate language of the virtual machine. Instead, our virtual machine is defined with a list of services. Expressions of our source language will be translated to the use of these services, or simple operations like memory allocation, data structure access, or send/receive messages. We then show how these services and simple operations can be easily implemented by an intermediate language like Mate.

**Threading Model**  The architecture detailed in this report has a simple threading model. Each virtual machine is only composed of two threads: 1) one is shared by all cells running on the virtual machine; and 2) the other used to receive messages from other sensor network nodes, maintain the message queue, and dispatch messages to the cell-running thread. Although we do allow multiple cells running on top of the virtual machine, there could only be at most one cell in execution at any given point. We base this decision on the following reasons:

- Sensors are usually memory and energy constrained. Although multithreading is a nice feature for many modern general purpose programming languages, it would introduce high overhead for sensor networks. In a multithreading environment, each thread would need to preserve its own call stack (with local variable frame and operand stack); this could constrain memory significantly when the number of threads goes up. Besides, since multithreading involves automatic frequent context switching, CPU would spend a significant percentage of energy doing non-application tasks, which is prohibitively expensive in a condition as sensor networks.

- With split-phase operations, all long latency operations have been split
Applications running on top of a sensor virtual machine are mostly related; the relationship between multiple cells in terms of CPU usage is more of coordination than of competition. This fact makes our design different from general purpose platforms where multiple users fight for resources. It is without going too far to assume CPU occupation can be negotiated via program logic. For instance, in a scenario temperature and humidity are both needed to be measured on a sensor network node, a main cell can first load itself up. It can then load a temperature measuring cell and a humidity measuring cell in, and connect to each of the two. Periodically, the main cell issues queries to each of them and get the result back.

2.2 Static Linkers: Composition Interface

Like nesC, our language supports modular design, where each module has a list of static linkers as composition interface, specifying what code fragments need to be imported and what code fragments are to be exported. Fig. 1 gives an example of two modules, TimerM and ClockM, and their composition TimerWithClockM. TimerM has two static linkers, Timer and Clock (declared with keyword slinker). Each static linker is bi-directional, which means the module needs to import some functions to be complete, and it has the ability to export some functions to modules to be statically linked to it. The composition of two modules is defined as a simple “+” operator (see the TimerWithClockM example). The composition matches static linkers by name, and for each pair of matched static linkers, each imported function in one module will be satisfied by the exported function in the other. The resulting composed module will have all imported functions of matched linkers satisfied, and re-export the exported functions. Composition process has the nice flattening property stating that the composed module is exactly the same as an atomic module semantically. This can be illustrated by TimerWithClockMEquiv module definition in Fig. 1, which is the semantically equivalent module of TimerWithClockM. Functions in static linkers do not have to be split-phase operations.

Technically, unlike nesC, our system does not distinguish components (atomic modules of nesC) and configuration (composed modules of nesC). Our composition mechanism has the flattening property such that composed modules can also be treated as atomic modules. We therefore have a smaller language core, and spare the somewhat ad-hoc syntax for module composition like the one in nesC. It is true in nesC interfaces of different names can be hooked together, and
-module TimerM {
    slinker Timer {
        export result_t start(char type, uint32_t interval){...};
        export result_t stop(){...};
    }
    slinker Clock {
        import int setRate(char interval, char scale);
    }
}
-module ClockM {
    slinker Clock {
        export int setRate(char interval, char scale){...};
        export result_t maintain(){...};
    }
}
-module TimerWithClockM = TimerM + ClockM;
-module TimerWithClockMEquiv {
    slinker Timer {
        export result_t start(char type, uint32_t interval){...};
        export result_t stop(){...};
    }
    slinker Clock {
        export result_t setRate(char interval, char scale){...};
        export result_t maintain(){...};
    }
}

Figure 1: An Example of Modules with Static Linkers

the resulting composed module can have interface names different from those of the parts. However this is just up to an interface renaming, which our language also supports (see Sec. 3 for details). Separating interface renaming from composition represents a separation of concern, a sign of good programming language design. nesC configuration-like construct can actually be encoded as a macro if necessary based on these more basic constructs.

2.3 Connectors: Communication Interface

As explained earlier in Sec. 1, each module can define a list of communication interfaces called connectors, through which all communication behaviors of the module happens. To illustrate how connectors work, we here give a simplified example of battlefield sensing in Fig. 2. Specific algorithm used for this scenario varies from application to application, and is independent of programming language design. We here just provide one of the possible implementations to demonstrate our language features.

Here a battlefield teemed with tanks is scattered with hundreds of densely populated sensor network nodes for tank detection. The ultimate goal is for the
module TankDetectionM {
    connector TankDetector {
        export result_t computeSpeed() {...};
        import result_t speed_ready();
    }
    slinker LocationSensor {
        import int getLocation();
    }
}

module LocationSensorM {
    slinker LocationSensor {
        export int getLocation() {...};
    }
}

module FullTankDetectionM = TankDetectorM + LocationSensorM;

module MediatorM {
    connector TankDetector {
        import result_t computeSpeed();
        export result_t speed_ready();
    }
    connector BattlefieldReport {
        export result_t reportStatus() {...};
        export result_t trendEstimation() {...};
    }
}

Figure 2: An Example of Modules with Connectors

commander to learn the global status of the whole battlefield at certain point, and its trend for next moment. Each sensor in this sensor network keeps track of the location and speed of tanks in its range. However, since the battlefield is geologically vast, each sensor can only cover a small sub-area. To deal with this problem, nodes in this sensor network is formed in a 3-level tree structure, with leaves to be tank detectors, intermediate nodes as mediators and root as used by the commander. Mediator sensor network nodes have two responsibilities, collecting data from tank detection nodes, and reporting to root about the battlefield status and trend. Since different tank detector nodes might detect movement of the same tank, it is important mediators process the data first and eliminate duplicates before sending data to their parent node. Technically speaking, it is possible a sensor network node both serves as tank detector and a mediator, but we here just consider the simplified case where two responsibilities do not overload.

Each tank detection node will have a FullTankDetectionM module loaded once it is up and running. FullTankDetectionM is the result of static linking of a location sensor driver module LocationSensorM and the main tank de-
tection module \texttt{TankDetectionM}. Here \texttt{TankDetectionM} has a connector called \texttt{TankDetector}, which according to our semantics for static linking, will also become a connector for \texttt{FullTankDetectionM}. The connector has one export function, \texttt{computeSpeed}, to compute the speed of the tank. Because of the split-phase nature of operations in connectors, a corresponding import function \texttt{speed\_ready} is provided, and at the end of the function body of \texttt{computeSpeed}, \texttt{speed\_ready} is supposed to be called.

Each mediator node will have a \texttt{MediatorM} module loaded once it is started. It has two connectors, \texttt{TankDetector} to communicate with tank detection nodes (collecting data), and \texttt{BattlefieldReport} to communicate with the root.

With this, \texttt{MediatorM} module can have a piece of code like the following, which might show up in any function defined in \texttt{MediatorM}:

// cref is a reference to module runtime created from FullTankDetectionM
x = connect cref at TankDetector;
x.computeSpeed();
x.disconnect;

It means the cell created from \texttt{MediatorM} initiates a connection with a cell created from a tank detection module \texttt{FullTankDetectionM}. The connector used is \texttt{TankDetector}. The returned value of \texttt{connect} expression \texttt{x} is of connection type. With it, \texttt{computeSpeed} can be called. The connection can also be disabled after mission is completed. This is the most basic use of connectors. Note \texttt{cref} could be obtained in various ways, such as after an explicit dynamic loading (details in Sec. 2.5), or a value passed in as function parameters.

\textbf{Stateful Connections} The example in Fig. 2 gives an oversimplified example where connectors are composed of a list of exported functions and a list of imported functions. In reality, connections are almost always stateful (recall TCP for instance). Each connection always has something specific to itself to remember. For fulfilling this goal, our connector design also allows mutable fields as part of a connector definition.

In contrast with connection fields are cell fields. Our language also allows module runtimes to own application-specific states. The difference between connector fields and cell fields is that each connection keeps a separate copy of the connector fields, but all connections to a single cell share the same cell state. For instance, if a cell has a connector \texttt{TankDetector}, all states that are specific to the connection, such as detection time length, should be kept as connector fields. On the other hand, states that are deemed as the status of cell, such as the location information of a sensor, should be kept as cell states.

\textbf{Connector Generativity} As a P2P operation, the established connection via \texttt{connect} expression is largely symmetric. However, since there is always a party who initiates the connection and the other party who receives the request, the semantics of \texttt{connect} is also divided into two parts:
When a cell receives a request to connect to its certain connector, it generates and initializes a new copy of connector fields. After that, all export function calls via this connector will be operating on this copy of connector states. Our approach is *generative*, which means new request to connect to the same connector will not supersede the previous one, or be denied due to the existing one; instead, new connections will be set up and multiple connections to the same connector can co-exist. Different connector requests to the same connector will not interfere with each other: their connection states are separate, unless programmer intentionally wants to; in that case cell states can be used for state sharing. Whenever a connection is established, a fresh session ID is generated to tag the store for connection instances. The ID will be shared by the two parties of the connection.

When a cell sends out a request for another cell to connect to its certain connector, obviously the same generative approach is taken. Different from the receiving party where all generative state handling is all hidden under the rug, programmers can actually get a handle to each connection, just as shown in the tank detection example. Here we brought to reader’s attention the rebindability of connectors:

```
// cref1, cref2 are pre-existing cell references
x = connect cref1 at TankDetector;
y = connect cref2 at TankDetector;
x.computeSpeed();
...  
y.computeSpeed();
```

We therefore say, connections are generative in both directions.

Connections are long-lived, in the sense that they are always there until they are explicitly disconnected, via `disconnect` expression. In this case, all connection states are removed from both sides for the specific connection. In Sec. 2.6 we will see even if a cell migrates to a new physical location, its existing connections will be carried over and kept alive.

Last we briefly discuss the static semantics of connector connection, the typing issues. Basically the general rule is every import from one party has to be satisfied by an export from the other party. However, it is allowed for either of the parties to have extra exports that are not matched. Connection fields are not matched.

### 2.4 Location-Carefree Connections

The use of connectors in last section requires the connection initiator specifies who it intends to communicate with, such as the `cref1` and `cref2` in last example. However, in many cases, the initiator does not really care what physical node it connects with, what it does care is the other party must satisfy its need for connection. The satisfaction process happens to fit well with connector compatibility checking. Therefore, any node that has a compatible connector
(with the same name, and all imported functions in the initiator’s connector are exports in the compatible node) will be connected. We call connection of this kind location-carefree connection. The syntax for it is shown in the following example:

```java
x = connect any at TankDetector;
x.computeSpeed();
```

Compared with regular Internet, sensor network is a very good fit for location-carefree connections. For one thing, the infrastructure of sensor network is wireless, which implies for each sensor network node, it is very easy to broadcast the request and wait for responses from any of the neighbor node. After all, all sensor network communication will be through broadcasting eventually. Since sensor network is also typically dense, the latency for any neighbor to respond is usually low. For another reason, sensor network is typically formed by hundreds of nodes who are functionally the same, which means they have the same connectors. Normally when a request is sent, like retrieving a temperature or checking the movement of a tank, it does not matter which sensor responds in a small locality. The bottom line is one of the nodes does respond.

### 2.5 Dynamic Module Loading/Unloading

Our programming language provides support for loading modules and unloading their runtime form cells. This is important because running cells receive requests and make responses, which would consume energy. Algorithms can be designed to detect in a given situation, what minimum set of sensor network nodes are needed to keep the whole network correctly functioning. All the other nodes can just be shut down or put to a sleeping state. When some nodes reach a power depletion, or the topology of the sensor network changes, the nodes can be again reactivated. This will involve dynamic loading and unloading of modules and cells.

**Loading/Unloading Semantics** A basic example to show the use of our loading/unloading language construct is as follows:

```java
// Module_0 is a pre-existing module name
x = load Module_0;
...
unload x;
```

The load process involves the creation of the cell runtime memory layout, including allocation of cell fields. The language construct assumes an implicit lookup process, which is given a module name, returning the code piece for the module. For a sensor network node without permanent storage, this needs the support of virtual machine, where a code repository is associated.

The unloading process can unload cells from memory if its reference is given. The reference does not necessarily come from load. It can also be passed param-
eters or any sort already known to the program. A difficult issue is to consider the handling of connection states. After all, a successful unloading will automatically disconnect all connections associated with the cell. We need a notification mechanism in the semantics to explicitly disconnect all the connections before unloading.

2.6 Cell Migration

Sec. 1 justified the need to have migration supported as a core part of a programming language, so that different programmers can write out different migration processes based on the need of their specific applications. A basic example to show the use of our migration language construct is as follows:

```plaintext
// x is a pre-existing cell reference
migrate x to NodeID_1;
```

First we need to make clear the migration process needs to be atomic, i.e. it either succeeds without interruption, or it fails. In the following semantics description, readers need to keep atomicity as an implicit requirement.

When the above expression is evaluated, cell referred to by \( x \) will inform all the cells currently connecting to it, about the migration and the destination it intends to move to (in the example, \( \text{NodeID}_1 \)). All the informed cells will hence update their states accordingly to record the change of the connections. When all are successfully done, the migrating cell serializes all its cell-level states and connection states, unload itself from the memory of the current node, sends the serialized form to the new location, and at this new location loads the same code in, deserializes the state. The process also involves the backup and recovery of execution point.

All the internal states of the application, including the states of all its living connections, need to be preserved and transferred. The complexity and atomicity of this process justify why we treat migration as a first-class language construct: a user-defined process would be error-prone, if possible at all.

From another perspective cell migration shows the long-livedness of connections. Even when one of the connecting parties migrates to new physical nodes, original connections associated with the migrating cell are still kept alive. Connections are cut off only if they are explicitly disconnected or the cell is explicitly unloaded from memory.

3 Formal Syntax

The abstract syntax of the language is shown in Fig. 3. Before elaborating on details, we first reinstate the difference of two terms we use: `module` and `cell`. When we use the term module, we emphasize the code property of the program fragment, i.e., the static code piece that can be composed and loaded to memory; on the other hand, when we use the term cell, we emphasize the runtime property of the program, which can be taken as module after being
loaded to memory and module runtime possessing states. Understanding the difference between the two terms is important for understanding our language.

A module $M$ can be formed in three ways. It is either an atomic module, or a composed module or a module with interface renaming. $\mu$ is the name of the module:

- Atomic modules are formed with a list of static linkers ($S$), a list of connectors ($C$), a list of cell fields ($\epsilon$), and initialization code for bootstrapping the module when they are first loaded to memory ($e$). The overline denotes a repetition of items; for instance $X$ represents a list of $X$s. Each static linker has a name $n$, a list of functions to be imported ($\text{import } \iota$), and a list of functions to be exported ($\text{export } \iota$). Different from static linkers, it can also own a list of connector fields ($\epsilon$), which are used for stateful connections.

---

**Figure 3: Abstract Syntax**

$M ::= \text{module } \mu \{ \overline{\tau C S \epsilon} \}$

- atomic module

$\text{module } \mu = \mu_1 + \mu_2$

- composed module

$\text{module } \mu = \mu_0 [n_1 \mapsto n_2]$

- module with interface renaming

$C ::= \text{connector } n \{ \overline{\tau \text{import } \iota \text{export } \tau} \}$

- connector

$S ::= \text{sinker } n \{ \overline{\text{import } \iota \text{export } \tau} \}$

- static linker

$\iota ::= \text{fn } \tau \{ e \}$

- function

$e ::= \text{instn } \tau$

- state

$e ::= x | \text{est}$

- variable, constant

$\vert e.\text{instn} | e.\text{instn}\!::\!e$

- cell reference

$\vert e.\text{fn} | e.\text{fn} | e(e)$

- instance and its operations

$\vert \text{load } \mu | \text{unload } e$

- function and its operations

$\vert \text{connect } e \text{ at } n | \text{disconnect } e$

- load/unload

$\vert \text{connect any at } n$

- connect/disconnect

$\vert \text{migrate } e \text{ to } \nu$

- location-carefree connection

$\vert e; e$

- migrate

$\vert e; e$

- continuation

$T_m ::= \text{module } \{ \overline{C S} \}$

- module signature

$T_c ::= \text{cell } \{ \overline{C} \}$

- cell signature

$\tau ::= \text{int} | \tau \rightarrow \tau | T_c | C$

- type

$\nu ::= \text{node ID}$

$\mu ::= \text{module name}$

$n ::= \text{interface name}$

$\text{fn}$

- function name

$\text{instn}$

- instance name

$\text{cid}$

- cell ID
• Composed modules are simply formed by a “+” operator. $\mu_1 + \mu_2$ means module with name $\mu_1$ and module with name $\mu_2$ are to be composed together.

• Modules with interface renaming have a syntax in the form of $\mu_0 [n_1 \mapsto n_2]$. It means the resulting module is the same as module with name $\mu_0$, but interface with name $n_1$ is renamed to $n_2$. If $n_2$ is $\phi$, it means interface with name $n_1$ is removed. Here interface applies to both static linkers and connectors.

Expressions ($e$) in the language include:

• variables ($x$) and constants ($cst$).

• cell reference constants, which include cell IDs ($cid$) or thiscell denoting the cell reference to the current running cell itself.

• connection constants thisconn, denoting the current connection.

• state-related expressions. Depending on whether the state is a cell field or a connector field, these expressions are also in different forms. thiscell.instrn is used to get the value of a cell field, $e.n$ is used to get the value of a connector field, where of course here $e$ is a variable of connection type. $e := e$ is the expression to set state to new values.

• function-related expressions. $n :: fn$ is used to refer to a function imported or exported in static linker with name $n$; $e.fn$ is used to refer to a function associated with a connection where $e$ has a connection type; When $fn$ shows up, it should be understood as thisconn.$fn$. Finally, $e(e)$ is standard function application.

• expressions for loading modules and unloading cells. load $\mu$ is used to load a module into memory where $\mu$ is the name of the module to be loaded. This expression returns a cell reference; unload $e$ on the other hand unloads a cell from memory, where $e$ is of cell reference type. Here $e$ can either be a CID constant, or thiscell, a reference obtained from load, or a reference passed in as a function parameter.

• expressions for connecting and disconnecting cells. connect $e$ at $n$ is used to connect to cell(s) referred to by cell reference $e$; the connector to be connected is of name $n$. The expression returns a variable of connection type. disconnect $e$ disconnects the connection represented by $e$ ($e$ here is of connection type). connect any at $n$ is used to initiate location-carefree connections.

• expression for migrating cells. migrate $e$ to $\nu$ is used to migrate cell referred to by cell reference $e$ to a new node identified by $\nu$.

• $e_1; e_2$ is sequencing of expressions. It means evaluating $e_1$ first, and then $e_2$. 

16
Types of the language include a module signature and a cell signature. As we can see, module signature includes the connectors and static linkers of the module, and yet cell signature only includes the runtime interfaces of the module, which are connectors. Besides, we also support basic types like integers (int), function types ($\tau \rightarrow \tau$) and connection type, which is the same as declaration of a connector.

4 Cell Runtime Design

Starting from this section, this report demonstrates how a language we just defined can be mapped to an implementation. As discussed in Sec. 2.1, the approach we take is intermediate: we will show a possible design that is low-level enough to be directly implemented, and yet does not limit itself to just one specific implementation to a specific operating system. In this section, we first give a description of cell runtime representation. In Section 5 and 6, we give a specification of our virtual machine. In Section 7, a description of language semantics is given.

At post-compilation time, modules can take different forms depending on implementation. They could be in regular binary code form, or in bytecode form similar to Java bytecode, or intermediate code capsules created by Mate. A close analogy of post-compilation modules in our design would be class files in Java. Since typical sensors do not have permanent storage associated with them, we here assume post-compilation modules are also kept in memory, with type ModuleType. Note this kind of memory habitants are different from cells; the latter would be analogously thought of as objects in Java.

Cell Runtime Memory Layout When a module is loaded, a cell is created using the module as code template. In this section, the focus of our attention is on the runtime memory layout of cells; the loading process itself will be detailed later in Sec. 6.5.

The runtime memory representation of a cell can be demonstrated by the following type:

```c
#define MAX_CELL_FIELD 100
#define MAX_CONN 100
type CellType = struct {
    ConnType ct[MAX_CONN];
    State ft[MAX_CELL_FIELD];
    string µ;
};
```

It indicates each cell runtime is composed of: 1) $ct$, a list of states recording the status of live connections the cell currently keeps. We sometimes call it connection table of a cell; 2) $ft$, a list of states recording the values of cell fields; 3) $µ$, the name of the codebase module of the cell. Type $State$ and $ConnType$ are defined below:
```go
#define MAX_CONN_FIELD 100

type State = struct {
    string label;
    VAL value;
};

type ConnType = struct {
    ConnID connid;
    string connName;
    CID connectTo;
    State st[MAX_CONN_FIELD];
};
```

According to the definition above, a connection is recorded with its connection ID (`connid`), the name of the connector involved in the connection (`connName`), the CID of the party being connected (`connectTo`), and a list of per-connection states (`st`).

**Some Housekeeping Operations** Several local housekeeping operations are defined on data structures introduced above. These operations are mostly getters and setters of related data structures; we simply list them here, instead of giving full definitions. They will be used in later sections when we define more complicated operations:

- **updateConnT** (`cid, connid, connName, connectTo, st`) updates cell `cid`'s connection table, in which the entry representing connection with ID `connid` is modified with the new tuple `(connid, connName, connectTo, st)`.
- **newConnT** (`cid, connid, connName, connectTo`) adds to cell `cid`'s connection table one new entry, `(connid, connName, connectTo, st)`, where `st` is the initial per-connection state of connector `connName` of cell `cid`.
- **lookupConnT** (`cid, connid`) returns the entry indexed by `connid` of cell `cid`'s connection table.
- **removeConnT** (`cid, connid`) removes the entry indexed by `connid` from cell `cid`'s connection table.
- **updateCellSt** (`cid, label, value`) updates cell `cid`'s field, where the value labeled by `label` is changed to `value`.
- **lookupCellSt** (`cid, label`) returns the value of cell field indexed by `label` of cell `cid`.
- **initCellSt** (`mh`) returns the initial cell-level state after module referenced by `mh` is loaded.
- **updatePerConnSt** (`cid, connid, label, value`) updates cell `cid`'s field in connection `connid`, and the label of the field is `label`. After updating, the field is set to `value`. 

18
• lookupPerConnSt(cid, connid, label) returns the value of cid's field in connection connid, and the label of the field is label.

5 Virtual Machine Design: Data Structures

Each sensor has a virtual machine running on top of it. Multiple cells can run on the same virtual machine. Each virtual machine owns some global data, and provides several built-in services that will be used by programs written in our language. In this section, we specify virtual machine level data structures. Virtual machine built-in services will be specified in Sec. 6.

5.1 Standard Data Structures

Same as traditional models, our virtual machine also holds the following standard data structures:

• a call stack, with each frame owning a local variable area and an operand stack. The difference for our language is, whenever a function call is fired, two parameters will be put onto operand stack automatically, the cell ID the function belongs to, and the connection ID the function invocation is currently engaged in. Details on this subject will be discussed in Sec. 6.1.

• a program counter, the execution pointer indicative of where the current execution runs for the cell. Since in our virtual machine design, all cells share one thread, the program counter will need to consist of two parts: the CID of the cell currently in execution, and the pointer to point to the next instruction in the code area of the cell.

5.2 Module Location Table

Each virtual machine holds a table, module location table (shorthanded as MLT), which keeps record of the modules sitting on a particular sensor network node. It adopts the following form, defined as mlt:

```c
#define MAX_MLT 100

type ModuleLocRecord = struct {
    string µ;
    ModuleType * mh;
};

ModuleLocRecord mlt[MAX_MLT];
```

Notation (*) is used here (and several definitions in later sections) to denote reference to a data structure. Readers should not confuse this level of representation with our source language.

Two local operations are defined on MLT, which will be used in later sections:
• updateMLT(\(\mu, mh\)) updates the virtual machine’s MLT, in which the entry representing module with name \(name\) is modified with new new location pointer \(mh\). If the entry for \(name\) does not exist, the operation append a new entry to the table.

• lookupMLT(\(\mu\)) returns the entry indexed by \(\mu\) of the virtual machine’s MLT.

5.3 Cell Location Table

Each virtual machine holds a table, cell location table (shorthanded as CLT), which keeps record of the location information of cells. It adopts the following form, defined as \(clt\):

```c
#define MAX_CLT 100
type Status = enum {Active, Remote};
type LocType = union {
    CellType* ch;
    NetAddr ν;
};
type CellLocRecord = struct {
    CID cid;
    Status sts;
    LocType loc;
};
CellLocRecord clt[MAX_CLT];
```

Each entry of CLT (with type \(CellLocRecord\)) indicates the location to find the cell with ID \(cid\). There is a status flag (\(sts\)) in the data structure, which can represent two different cases, and the type of \(loc\) depends on the value of the status flag:

• The cell is currently running on the same virtual machine. In this case, the \(loc\) field will keep record of the cell handle, which has type \(CellType*\). The flag is set to \(Active\).

• The cell is currently on a different sensor node. In this case, the \(loc\) field will keep record of the sensor node information. \(NetAddr\) is a built-in type. Depending on implementation, it could be the MAC address associated with sensor networks, or any identifier which can uniquely find a specific virtual machine on a remote site.

The reason why CLT not just has entries about cells running on top of the CLT-owning virtual machine, but also has entries about cells on other virtual machines is realistically each sensor node is supposed to know its neighbor information. Sensor network at low level is broadcast network in nature. It is these neighbor sensors a particular sensor is able to communicate with. It is therefore important for a virtual machine, at any give time, to keep an active record of
what are the neighbors of its lodging sensor. The maintenance of neighbor table is a task for the virtual machine. Its protocol varies depending on many factors such as efficiency and energy preservation requirements. It is initialized when sensor networks are laid out, and can be refreshed periodically. This helps for location-carefree operation.

In addition, since we allow cell migration, it is possible that requests are sent to a sensor after the cell has moved to another sensor. We need a forwarding mechanism.

Several local operations are defined on CLT, which will be used in later sections:

- `updateCLT(cid, sts, loc)` updates the virtual machine’s CLT, in which the entry representing cell `cid` is modified with new status `sts` and new location `loc`. If the entry for cell `cid` does not exist, the operation append a new entry to the table.

- `lookupCLT(cid)` returns the entry indexed by `cid` of the virtual machine’s CLT.

- `randomSelectNeighbor(n)` randomly selects a `cid` from the list of cell IDs appearing in the virtual machine’s neighbor table. The selected one must have a connector `n`.

6 Virtual Machine Design: Built-in Services

Here we specify the built-in services each virtual machine is expected to have. A program written in our language will depend on the correct running of the virtual machines. Some of our language constructs will be compiled to a form using these primitives; built-in services of virtual machines therefore also form an indispensable part of our language core. A summary of the built-in services is as follows:

6.1 call: Function Invocation Service

Function invocation is a basic service available to essentially every virtual machine design. In our virtual machine, the service has the following interface:

```c
call(cid, connid, fn, ifn, v);
```

which denotes invoking a function named `fn` with parameter `v`, defined in cell `cid`. There are two cases: 1) if the function is defined in a connector, `connid` is the ID of the current connection. In this case, `ifn` is set to `NULL_STR`; 2) if the function is defined in a static linker, `ifn` is the name of the static linker. In this case, `connid` is set to `NULL_CONNID`. Here the difference originates from the fact that if the function is in a static linker, static linker name and function name combined is enough for all information to invoke the function; however if the function is in a connector, connector name and function name combined is
not enough, since the function needs to know what the current connection is, to
decide on which copy of the connection fields it operates on.

The service first locate the code entry for function fn. This is not a difficult
task because cell runtime representation already has its codebase information
(See Sec. 4). The point that deserves attention is that when the function call
happens, not just v is pushed to operand stack; two more values are also put
to stack: cid, the CID of the current cell, and connid, the connection ID of the
current connection. Thus, inside the function, code can refer to these two values
using thiscell and thisconn. This handling is very similar to that of the this
pointer of object-oriented languages like C++ and Java.

6.2 Serialization/Deserialization Services

Serialization is the crucial process to preserve cell states when cells migrate
from one sensor to another; we expect the states of cells are kept and restored
when they arrive at the new virtual machine. Deserialization is the dual of
serialization, which deals with restoration of serialized data. The two services
have the following signatures:

\[
\text{SerializedForm serialize}(\text{CID } \ast \text{cid});
\]
\[
\text{CID deserialize}(\text{SerializedForm } s\mathcal{f});
\]

A serialized form of cells has type \text{SerializedForm}, which is defined as follows:

\[
\text{type SerializedState } = \text{struct} \{
\text{string label;}
\text{SerializedVAL value;}
\};
\]

\[
\text{type SerializedConnType } = \text{struct} \{
\text{ConnID connid;}
\text{string conname;}
\text{CID connectTo;}
\text{SerializedState sst[\text{MAX_CONN\_STATE}];}
\};
\]

\[
\text{type SerializedForm } = \text{struct} \{
\text{CID cid;}
\text{ModuleType } m;
\text{SerializedConnType sct[\text{MAX_CONN}];}
\text{SerializedState sft[\text{MAX\_CELL\_STATE}];}
\};
\]

The definition indicates a serialized cell is composed of: 1) cell ID; 2) codebase
of the serialized cell, of type \text{ModuleType}; 3) serialized connection fields recording
live connection information, of type \text{SerializedConnType}; 4) serialized cell
fields of type \text{SerializedState}.

Serialization and deserialization of data types (type \text{SerializedVAL}) are not
trivial issues. Although the current definition of our language does not support
many data types like pointers and objects since the pick of these data types are
almost orthogonal to the issues we are interested, but in a realistic language,
references and objects should always be supported. In a C like language, if a
pointer is serialized, the data that is pointed to are also expected to be serial-
ized. In a Java like language, if an object is serialized, any field of the object
also needs to be serialized. Since a pointer might contain another pointer inside,
or an object has another object as a field, the serialization process will operate
hierarchically on the data to be serialized. When recursion happens, the pro-
cess also needs to make sure cycles are taken care of to avoid non-termination.
The process is largely standard; interested readers can refer to [LY99] for a
description of Java’s serialization process.

Deserialization inevitably involves a process of constructing cell runtime
memory layout. It differs from load in the sense that load only initialize cell
fields and connection fields, and yet deserialize recovers states from serialized
data. At the end of the deserialization process, updateCLT(cid, Active, ch) is
invoked, where cid is the ID of the just deserialized cell (this information is
inside serialized form), and ch is the pointer to the cell runtime memory area.
This virtual machine thus acknowledges the existence of cell cid, active on top
of it.

6.3 send: Virtual Machine Message Sending Service
Built-in service send sends a request to a cell potentially located on a different
virtual machine and sensor. It signature is as follows:

send(NetAddr ν, CID source, CID dest, Request req);

which denotes a request req is sent to location ν. The receiving cell has a CID
dest, locating on top of a virtual machine at location ν. The request is sent by
cell source. Because of migration, a request might need to be forwarded if a cell
has already moved to another virtual machine. It is therefore not always true
that source is the party invoking send.

6.4 receive: Virtual Machine Message Listening Service
Built-in service receive is a message handler of the virtual machine. It is
automatically triggered when requests sent via send are received. From a low-
level view, any inter-cell communications are mediated by virtual machines,
which implies, any invocations sent to a specific cell will first be received by
the virtual machine of the cell, and then the message listener receive of the
virtual machine will decide on how to handle it. The service has the following
signature:

receive(CID source, CID dest, Request req);

which denotes a request req is received by the current virtual machine, and it
is sent from cell source, and it is sent to cell dest sitting on the current virtual
machine. Either source or dest can also be set to be VM, which denotes the request is sent from the current virtual machine, or sent to the current virtual machine, instead of a specific cell sitting on top of it.

We omit the formal definition of Request type here, but it is indeed a straightforward union type of various record types, each of which represents a request format. Each record type has its first field set to the flag representing the kind of request, followed by a list of parameters needed by the specific kind of request. Request flags can be to Deserialize, NewConn, DisConn, CallConn and UpdateLoc, as we will see shortly. We define receive mathematically using case analysis on request format, as follows:

\[
\begin{align*}
\text{receive}(s, VM, \langle \text{Deserialize}, sf \rangle) & \overset{\text{def}}{=} \text{deserialize}(sf); \\
\text{receive}(s, d, \langle \text{NewConn}, connid, n \rangle) & \overset{\text{def}}{=} \text{newConn}(d, connid, n, s); \\
& \quad \text{if } \langle d, \text{Active}, ch \rangle = \text{lookupCLT}(d) \\
\text{receive}(s, d, \langle \text{NewConn}, connid, n \rangle) & \overset{\text{def}}{=} \text{send}(\nu, s, d, \langle \text{NewConn}, connid, n \rangle); \\
& \quad \text{if } \langle d, \text{Remote}, \nu \rangle = \text{lookupCLT}(d) \\
\text{receive}(s, d, \langle \text{DisConn}, connid \rangle) & \overset{\text{def}}{=} \text{removeConn}(d, connid); \\
\text{receive}(s, d, \langle \text{CallConn}, connid, fn, v \rangle) & \overset{\text{def}}{=} \text{call}(d, connid, fn, NONE, v); \\
\text{receive}(s, VM, \langle \text{UpdateLoc}, \nu \rangle) & \overset{\text{def}}{=} \text{updateCLT}(s, Remote, \nu);
\end{align*}
\]

The request handler definition is largely self-explaining. In the first case, when a deserialization request is received, the virtual machine deserializes the serialized data sf. In the second and third cases, a request to set up a new connection with cell d is fired. If d is located on the receiving virtual machine, connection table is updated (the second case); if d is located on a remote virtual machine, the current virtual machine simply forwards the request. This is possible because of cell migration. In the third case, a request for disconnection means the removal of a connection table entry. In the fourth case, a request to invoke a function in a connector mostly hinges on the invocation of system service, the call service. In the fifth case, a request to update location information of a cell, which is useful for cell migration, indicates an update on the neighbor table of the virtual machine.

6.5 Loading/Unloading Services

Service load provides the functionality of preparing a cell runtime memory layout from its codebase module; in contrast, service unload provides the functionality of releasing the memory area a cell occupies, such as its cell states, connection states, local variable pool and call stack. Loading and unloading services are important for sensor programming; with them, modules can be flexibly loaded into memory when they are needed, and memory can be freed when the functionalities provided by the cell are not useful any more. The
signature for these services are:

\[
\text{CID load}(\text{string } \mu);
\]

\[
\text{string unload}(\text{CID } \text{cid});
\]

First, service \textit{load} can be defined as follows:

\[
\text{#define NULL_CONNT null}
\]

\[
\text{load}(\mu) \defeq \begin{cases} 
\text{cid} = \text{newCID}(); \\
\text{ch} = \text{alloc}(\text{NULL_CONNT}, \text{initCellSt}(\mu), \mu); \\
\text{updateCLT}(\text{cid}, \text{Active}, \text{ch}); \\
\text{cid} 
\end{cases}
\]

In this definition, \textit{cid} is the ID of the newly created cell, whose codebase is set to the module with name \textit{\mu}. The cell’s initial connection table is empty (\text{NULL_CONNT}) and its initial cell fields are initialized. \textit{load} service then updates \textit{CLT} to reflect the addition of cell \textit{cid} to the virtual machine. Finally, \textit{load} service returns the cell ID of the newly created cell.

Correspondingly, \textit{unload} service is defined as below. All cells keeping an active connection with the soon-to-be-unloaded cell will be sent a disconnection message, and the memory area allocated for the cell is freed:

\[
\text{unload}(\text{cid}) \defeq \langle \text{cid}, \text{Active}, \text{ch} \rangle = \text{lookupCLT}(\text{cid}); \\
\langle \text{ct}, \text{ft}, \mu \rangle = \ast \text{ch}; \\
\text{foreach } \langle \text{connid}, n, \text{cid}', \text{st} \rangle \text{ in } \text{ct} \\
\langle \text{cid}', \text{Remote}, \nu' \rangle = \text{lookupCLT}(\text{cid}'); \\
\text{req} = \langle \text{DisConn}, \text{connid} \rangle; \\
\text{send}(\nu', \text{cid}, \text{cid}', \text{req}); \\
\text{free}(\text{ch}); \\
\mu; 
\]

Note that since we take the simple threading model for our virtual machine design, loading process does not create new thread. Consequently, we do not allow cells to execute at the same time. The initialization code of the loaded cell is not executed. They can be accessed by the loading cell via expressions like \textit{connect}, \textit{etc}. We have achieved multiple cells running on the same virtual machine without using multithreading.

7 Semi-formal Language Semantics

In this section, we present the semantics of our language. Instead of taking the usual formal approach, we intentionally define each language expression as a semi-formal program. This is because, mathematical formalizations can hide some details that still make a difference during implementation. As a system project in nature, we believe a semi-formal specification is more appropriate, as it is more precise specification of implementation.
Here we define the meaning function $\llbracket . \rrbracket$, whose domain is the set of expressions belonging to our language, and whose range is the set of programs whose control flow can be directly mapped to low-level virtual machine operations. $\llbracket e \rrbracket$ denotes the meaning of expression $e$. Readers are advised that these definitions should not be read as macros. For each expression defined below, the correctness of semantics is hinged on atomicity of the expression.

7.1 Cell Bootstrapping

For any realistic language, there is always this first question to ask: how could the first runtime be established? We here define it below. Notice that it is not part of source language expressions; instead, readers should set analogy between this and typing in `java HelloWorld` in the command line of Java virtual machine:

\[
\llbracket \text{bootstrap } \mu \rrbracket \triangleq \begin{align*}
\text{cid} &= \text{load}(\mu); \\
\text{init}(\text{cid});
\end{align*}
\]

Here $\mu$ is the name of the module to be loaded. The virtual machine first load the cell based on its code base module with name $\mu$, followed by `init(cid)`, a simple function to execute the initialization code of the cell, which is defined as part of module syntax (recall Sec. 3 for the grammar). This is the point execution of code starts.

According to the syntax of our language, we have two other ways to define modules: composed modules and modules with renamed interfaces. Bootstrapping these two kinds of modules is exactly the same as what we just described above. In fact, these two are only concerned with transformations of code, and does not have runtime effect. At compile time, the composed modules will be merged; modules with renamed interfaces will have them renamed. When it bears the `ModuleType` in our system, the code will look exactly like an atomic module. We leave out the part of code transformations of the two cases, since they are fairly intuitive.

7.2 Cell Dynamic Loading/Unloading

We next define the semantics of dynamic loading and unloading. Dynamic loading process is almost the same as bootstrapping process, except that the loaded cell does not run its initialization code. Dynamic unloading directly calls on `unload` service.

\[
\begin{align*}
\llbracket \text{load } \mu \rrbracket & \triangleq \text{load}(\mu) \\
\llbracket \text{unload } e \rrbracket & \triangleq \text{unload}(\llbracket e \rrbracket)
\end{align*}
\]

7.3 Cell Connection Establishment

Cell connection establishment involves two parties, the initiating party and initiated party. For the initiating party, all needs to be done is to add one more
entry to the cell’s connection table, recording the newly established connection.
For the initiated party, there are two cases: 1) if the initiated party sits on a remote node, a request needs to be sent, which is the NewConn request as we demonstrate below; 2) if the initiated party sits on the same node, a direct change of connection table is issued. Whenever a new connection is established, a new connection ID is set up via newConnID function.

\[
\text{connect } e \text{ at } n \quad \text{def} \quad \begin{cases} 
\text{cid} = \llbracket e \rrbracket; \\
\text{connid} = \text{newConnID}(); \\
\text{case lookupCLT}(\text{cid}) \text{ of} \\
\langle \text{cid}, \text{Remote}, \nu \rangle : \quad \text{req} = (\text{NewConn}, \text{connid}, n); \quad \text{send}(\nu, \text{thiscell}, \text{cid}, \text{req}); \\
\langle \text{cid}, \text{Active}, ch \rangle : \quad \text{newConnT}(\text{cid}, \text{connid}, n, \text{thiscell}); \quad \text{newConnT}(\text{thiscell}, \text{connid}, n, \text{cid}); \quad \text{connid}; 
\end{cases}
\]

For location-carefree connections, it is basically a random selection of neighbor cells first, and then connect to the cell:

\[
\text{connect any at } n \quad \text{def} \quad \begin{cases} 
\text{cid} = \text{randomSelectNeighbor}(n); \\
\llbracket \text{connect } \text{cid} \text{ at } n \rrbracket 
\end{cases}
\]

### 7.4 Cell Disconnection

Just as cell connection establishment, cell disconnection also involves two parties, the initiating party and initiated party. For the initiating party, all needs to be done is to remove the entry from the cell’s connection table, the one recording the connection. For the initiated party, similarly two cases are possible:

\[
\text{disconnect } e \quad \text{def} \quad \begin{cases} 
\text{connid} = \llbracket e \rrbracket; \\
\langle \text{connid}, n, \text{cid}, st \rangle = \text{lookupConnT}(\text{thiscell}, \text{connid}); \\
\text{case lookupCLT}(\text{cid}) \text{ of} \\
\langle \text{cid}, \text{Remote}, \nu \rangle : \quad \text{req} = (\text{DisConn}, \text{connid}); \quad \text{send}(\nu, \text{thiscell}, \text{cid}, \text{req}); \\
\langle \text{cid}, \text{Active}, ch \rangle : \quad \text{removeConnT}(\text{cid}, \text{connid}); 
\end{cases}
\]

### 7.5 Cell Field Access

The following two define the semantics for cell field access related expressions:

\[
\llbracket \text{thiscell}.\text{instn} \rrbracket \quad \text{def} \quad \text{lookupCellSt}(\text{thiscell}, \text{instn}) \\
\llbracket \text{thiscell}.\text{instn} ::= e \rrbracket \quad \text{def} \quad \text{updateCellSt}(\text{thiscell}, \text{instn}, \llbracket e \rrbracket)
\]
7.6 Connector Field Access

The following two define the semantics for cell field access related expressions:

\[
\begin{align*}
\llbracket e.\text{instn} \rrbracket & \equiv \text{lookupPerConnSt}(\text{thiscell}, \llbracket e \rrbracket, \text{instn}); \\
\llbracket e.\text{instn} := e' \rrbracket & \equiv \langle \text{connid}, n, \text{cid}, st \rangle = \text{lookupConnT}(\text{thiscell}, \llbracket e \rrbracket); \\
& \quad \text{updatePerConnSt}(\text{thiscell}, \llbracket e \rrbracket, \text{instn}, \llbracket e' \rrbracket);
\end{align*}
\]

7.7 Function Invocation

There are three cases involving function invocation. The first two cases are both to invoke a function defined in a connector. However, since the function can either be defined locally as \textit{export}, or defined remotely as \textit{import}. The semantic behavior of these two cases are actually different, demonstrated by \llbracket e.fn(e') \rrbracket_{\text{local}} and \llbracket e.fn(e') \rrbracket_{\text{remote}} respectively. The last case involves invocations to a function defined in a static linker. Since static linkers are always satisfied at compile time, it does not have the two cases of invocations to functions defined in a connector.

\[
\begin{align*}
\llbracket e.fn(e') \rrbracket_{\text{local}} & \equiv \text{connid} = \llbracket e \rrbracket; \\
& \quad v = \llbracket e' \rrbracket; \\
& \quad \text{call}(\text{thiscell}, \text{connid}, \text{fn}, \text{NULL_STR}, v); \\
\llbracket e.fn(e') \rrbracket_{\text{remote}} & \equiv \text{connid} = \llbracket e \rrbracket; \\
& \quad \langle \text{connid}, n, \text{cid}, st \rangle = \text{lookupConnT}(\text{thiscell}, \text{connid}); \\
& \quad \text{req} = \langle \text{CallConn}, \text{connid}, \text{fn} \rangle; \\
& \quad \text{send}(v, \text{thiscell}, \text{cid}, \text{req}); \\
\llbracket n.fn(e) \rrbracket & \equiv \text{call}(\text{thiscell}, \text{NULL_CONNID}, \text{fn}, n, \llbracket e \rrbracket);
\end{align*}
\]

7.8 Cell Migration

The meaning definition of \textit{migrate e to v} is given below. Cell migration starts with a round of announcement the migrating cell sends to each of the other cells currently connected to it (see the \texttt{UpdateLoc} message below). The virtual machine then serialize the migrating cell, unload it from memory, and send it to the destination via \texttt{Deserialize} request.
\[
\text{\textbf{[migrate e to } \nu]} \overset{\text{def}}{=} \quad \text{cid} = [e]; \\
\langle \text{cid}, \text{Active}, \text{ch} \rangle = \text{lookupCLT}(	ext{cid}); \\
\langle \text{ct, ft, mn} \rangle = *\text{ch}; \\
\text{foreach} \langle \text{connid, n, cid}', \text{st} \rangle \text{ in ct} \\
\quad \langle \text{cid}', \text{Remote}, \nu' \rangle = \text{lookupCLT}(	ext{cid}'); \\
\quad \text{req} = \langle \text{UpdateLoc, } \nu \rangle; \\
\quad \text{send}\langle \nu', \text{cid, cid}', \text{req} \rangle; \\
\text{sf} = \text{serialize}(	ext{cid}); \\
\text{unload}(	ext{cid}); \\
\text{send}\langle \nu, \text{cid, VM, (Deserialize, } \text{sf}) \rangle.
\]

8 Conclusion

In this project, we have designed a new programming language to support sensor network programming, which is well-suited to meet memory and energy requirement of sensor networks, and at the same time in better support of sensor communication abstraction. Applications in our language can be written into well-encapsulated modules and be combined together seamlessly; At run time, applications can also explicitly set up a connection with other applications, dynamically load other applications, or migrate to a different sensor network node with states preserved.

We have also designed a virtual machine with a small number of core services, and showed how programs written in our language can be compiled to an intermediate code form interpretable to the virtual machine.

References


