Meadow: A Software System for Cooperating Devices

November 9, 2014

Contents

1 Introduction

Programming multiple, autonomous devices to accomplish a task is not easy. Consider programming n devices to perform a task which requires interaction of these devices. Assuming that they depend on *message passing* for interaction, which is the norm in distributed computing, guaranteeing that the system works correctly under any order of interactions is not trivial. Though huge amount of research efforts have been devoted to the study of behavior of concurrent systems, most efforts have been focused on *specification* and *verification* of systems, rather than "programming" such systems.

As more programmable devices are connected, coordinating them to perform tasks is becoming common activities. Many new technologies, such as Internet of Things, robot programming, industrial automation, and smart cars, will only accelerate this trend.

In particular, **the Meadow system** is a software system for programming cooperating devices, where users can program interaction of devices from a single viewpoint, rather than from multiple, different viewpoints (i.e. one per each participating device). Devices are viewed as "programmable objects" in the Meadow system, which provide services to other devices. Using the **Meadowview scripting language**, users can define a process over a set of devices, by composing existing device services to accomplish a task. Later, the defined process can be installed in actual devices and then, executed.

1.1 Benefits

Device choreography When a task involves multiple devices, adding behaviors to devices which collectively perform a single task is not easy. The Meadow system allows to define a process over devices, which will then be synthesized into code fragments which will be deployed to devices. Execution of these devices will perform the task.

Workflows over devices Also, only using pre-deployed functionalities in devices, one can define a workflow over a set of devices. This workflow will be instantiated using

Deployment of reactive code to devices

Dynamically typed language Provides a dynamically-typed language which expedites the development of the system and avoids all hassle of statically-typed languages at the expense of performance.[1].

Dynamic device interface A device interface is defined as the *set of services* provided by the device. It represents the behavior of device, i.e. what it can provide to the rest of the (Meadow) world. Users can dynamically change the services of devices unlike many of the distributed computing frameworks such as CORBA.

This simplifies the deployment of new services in many devices and removes the burden of interface repositories or service discovery. A device requests a service to another device, and when the device does not have the requested service, then we **let it fail** – it's the developer's or system maintainer's responsibility to ensure that the target device actually provides the service at the moment.

[1] Latency in distributed programs are already quite high so we may not need to worry too much about performance at this stage. Also, all the messes of type compatibility between different languages, which we might eventually support through language bindings could be avoided to some degree.

Scalable event processing Complex event processing aims intelligent handling events by filtering and processing countless events into a small set of value-added events. However, processing events at a central server is difficult since the number of events to be processed will increase very quickly. A better way to handle is to let "event generator" to generate filtered, processed, and refined events rather than generating raw, unrefined events. Meadow runtime allows to add "filtering/processing" logic at the site of event geneation.

Runtime type checking Even if there are no static type checking, each value carries a type and whenever an "operation" is executed, type checking on the operands is performed. Also, whenever runtime type error is found, it will be reported to the original service requester so that it can be handled.

Interoperable with multiple standards The Meadow system understands multiple standards, such as *AllJoyn*, *Open Interconnect Consortium standards*, *Apple HomeKit*, *Google API/*, etc. This capability will allow the devices which only conform to AllJoyn standard to use the service of devices which only conform to Apple HomeKit.

Persistent device services Unlike distributed languages (like Scala, Erlang, dRuby), properties, functions, etc. are persistent.[2] When a device shuts down for some reason, rebooting the device (and Meadow runtime) will restore the device to some previous checkpoint.

Both interpreted and compiled Users can add interpreted Meadowview script but, for performance, users can opt to compile the code into native code and deploy the native code.

1.2 Meadow devices

In the Meadow system, *devices* are key elements of the system. A **device** is a uniquely identifiable entity, which can provide and request services. In particular, a unique, persistent name, in the form of dot-separated string[3], is assigned to each device. The *behavior* of a device is characterized by the set of services it provides, which can change dynamically as services are added or removed.

Conceptually, *any entity which can run a **Meadow VM** is a **Meadow device*** . For example, a Raspberry Pi board with a Meadow VM installed can be viewed as a Meadow device. One can attach a LED light to the board and create a service which turns on or off the light. Also, a Linux box can be viewed as a Meadow device, if a Meadow VM is installed on the box. When one can import MqSQL operations into this VM, this device can support database operations.[4]

1.3 Device services

A device can provide services to other devices. A **service** provided by a device is either a *property*, an *event*, or a *function*. A set of services provided by a Meadow device is called **device interface**. *The interface of a device can change dynamically by adding or removing services*.

[2] Should we make persistence as a system service rather than making it a default?

[3] Let's say, for now, we follow Java package naming convention – e.g. `com.sun.java.util`.

[4] Actually, a general Linux box as a Meadow device can be useful by itself in a non-IoT context. Check if this can compete with Hadoop, Pig, etc. For this, design the Meadow language so that MapReduce job can be described.

Properties A device can have properties.[5] A **property** of a device is a named value, which can be read or written locally (by the device which contains the property) or remotely. Properties support *pull-based dataflow model*.

For example, a device can have a property, `remainingBattery`, which is a real number between 0 and 1. Consider a factory which has multiple battery-operated devices. Then, one may want to check if all devices have enough battery life. Users can add a Meadowview script which monitors the battery life of devices periodically (and do some action, such as sending an email, when the battery life is low) or create a program which grabs and displays these properties using the **Meadow API**[6].

Events A device can *publish* and/or *subscribe to* events. An **event** is a named value, which can be read locally or remotely. An event is delivered to its subscribers, whenever an event producer publishes an occurrence of the event. Events support *push-based dataflow model*.

As an example, a device with a GPS receiver can produce an event, `locationChanged`. This event can be "used" by its subscribers by defining "event handlers" for the event. An event handler can be defined in the form of a *reactor* in the Meadow language or a program which uses the Meadow API provided by the Meadow system.

Functions A device can have functions, which can be called locally or remotely. A **function** is a piece of code parametrized by symbols, which can be executed by its local or remote consumers. A function can be invoked in either blocking or nonblocking manner.

Consider a "MySQL Server" device as an example. One can add a Meadow function, `addUser(name, age, address, phone)`, to the device by *importing* a corresponding C function which performs a DB operation over the MySQL server.

Imported vs user-defined services A service of a device can be either *imported* or *defined* by the user.

As an example of an **imported service**, consider an LCD panel device. The manufacture of the panel device can provide a static library compiled for Atmel AVR micro-controller. This library may provide a function, say "`void display_message(char *)`", which allows to write a string to the panel. The Meadow system provides a method for importing such functionality into a device, say as a *Meadow function* so that the functionality can be used locally or remotely.

*Device services through another software frameworks, such as **AllJoyn**, can also be imported into Meadow devices. We could also support **automatic discovery and registration of external services** such as AllJoyn services.*

In addition to imported services, **user-defined services** can be described using the Meadowview language and *installed* (or deployed) to the device. After being installed, the function can be used locally or remotely.

1.4 Coordinating device interaction

In the Meadow system, we are most interested in coordinating interaction between devices to perform tasks. For pull-based services, such as properties or functions, we can directly use such services inside the Meadowview code. To use events, which is a push-based service, we need to

[5] That is, a device is one of possible entities which can "own" properties. Actually, the most common and important class of property owner.

[6] The Meadow system provides a C library (also in C++, Java, Python, etc) which allows to access device services.

add some code which represents the behavior that we want when the given event occurs. For this purpose, the Meadow system allows to define and install *reactors* to devices.

Reactors A **reactor** is a piece of code, which is executed whenever the associated event occurs. It can also be called as a *callback* or an *event handler*.

As an example, let two devices be given: `Switch` and `LedLight`, where the former can generate `changed` event and the latter provides functions, `turnOn` and `turnOff`. We want the toggle switch device to control the LED light device. Consider the following Meadowview code as an example.

```
reactor LedLightControl(Swtich:changed ev) {
  if (ev.value == 1)
    LedLight:turnOn();
  else
    LedLight:turnOff();
}
```

This reactor code can be deployed either to the `Switch` device or the `LedLight` device. Deploying the code to a third device is also possible.

1.5 Meadow processes

The Meadow system allows to describe collective behavior of the devices in a *process description*. A **process** is a collection of properties, events, functions, and reactors. A process can be *instantiated with actual devices*, which amounts to *installation of the process into devices*. After installation, a process can be executed either by explicitly calling some "start" function or by generation of an event to which some reactors of the process is sensitized to.

As an example, let four devices be given. One device, called `detector` which detects any Meadow device which comes nearby. Whenever it detects a device, it generates an event called `detected`, which contains the "detected device"[7], as its payload. A device named `door` is an automatic door, which opens the door. Also, an LCD display, called `lcd`, which can display any string on the panel. Finally, the `logger` device provides the function `logEntry(name, time)`.

```
process WelcomeProc(detector, door, lcd, logger) {
  reactor (detector:detected ev) {
    // open the door
    door:open();

    // display the welcome message on the LCD panel
    name = ev.device:getOwnerName();
    lcd:display("welcome" + name);

    // log the entry; time is a predefined attribute of any event,
    // which contains the time of event occurrence
    logger:logEntry(name, ev.time);
  }
}
```

[7] Technically, the unique id of the device.

2 The Meadowview Language

The Meadowview is a language for describing behavior of individual devices or collective behavior of multiple devices. The language consists of two parts: **the core language** for describing individual device behavior and **the process language** for describing collective behavior of devices.

2.1 Core language

The core language allows to define **properties**, **events**, **functions** and **reactors**. The core language is a *native language of Meadow VM*, meaning that Meadowview code in core language can be directly interpreted by a Meadow VM.

2.2 Process language

The process (definition) language allows to define **processes**. The code written in process definition language *cannot* be directly interpreted by Meadow VMs. Therefore, a process definition must be synthesized into code in the core language before they are deployed to devices.

`WelcomeProc` revisited The `WelcomProc` process definition,

```
process WelcomeProc(detector, door, lcd, logger)
```

discussed in Section 1.5 can be compiled into *object code*, which is a set of $(\langle device \rangle, \langle service \rangle)$ pairs. There are many ways to compile above definition but one possible object code is given below.

```
// code to be deployed to door
event doorOpened;
// pragma deploy(door) -- forces deployment of reactor to door
reactor(detector:detected ev) {
  :open();
  event_publish(EVENT(doorOpened, device => ev.device) /* ctor */);
}

// code to be deployed to lcd
event lcdDisplayed;
reactor(door:doorOpened ev) {
  ownername = ev.device:getOwnerName();
  :display("welcome" + name);
  event_publish(EVENT(lcdDisplayed, name => ownername));
}

// code to be deployed to logger
reactor(lcd:lcdDisplayed ev) {
  logger.logEntry(ev.name, ev.time);
}
```

Upon process instantiation using actual devices, the service code will be instantiated with actuals and deployed to them. Let actual devices be `com.mv.b1.l1.detector1`, `com.mv.b1.l1.door`, `com.mv.b1.l1.lcd`, and `com.mv.logger`. Then, following instantiated code will be installed to above devices.

```
// instantiated code installed on com.mv.b1.l1.door
event doorOpened;
reactor(com.mv.b1.l1.detector:detected ev) {
  :open();
  event_publish(EVENT(doorOpened, device => ev.device) /* request ctor */);
}
// code to be installed on com.mv.bl.l1.lcd
event lcdDisplayed;
reactor(com.mv.bl.l1.door::doorOpened ev) {
  name = ev.device:getOwnerName();
  :display("welcome" + name);
  event_publish(EVENT(lcdDisplayed, name => ownername));
})
// code to be installed on com.mv.logger
reactor(com.mv.bl.l1.lcd::lcdDisplayed ev) {
  logger.logEntry(ev.name, ev.time);
})
```

[8]

3 The Meadow VM

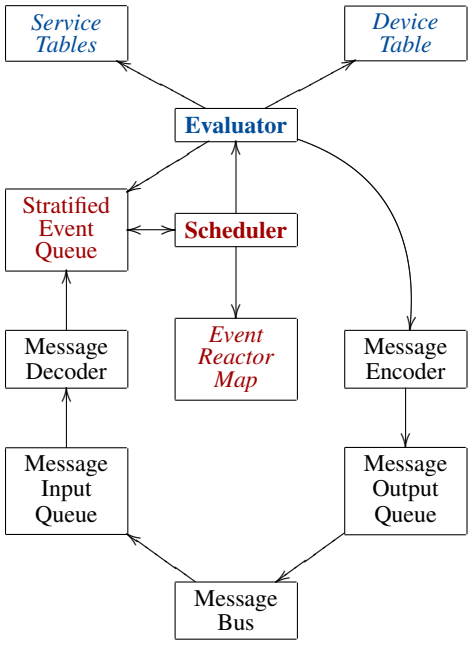
The Meadow system consists of Meadow devices, where each device runs a Meadow VM (or Meadow runtime), which is characterized by the properties, events, functions, and reactors, it contains. The **Meadow VM** is a middleware which is responsible for the following functionalities:

- **Management of device services**: Allows to dynamically add, remove, update *properties*, *events*, *functions* (imported or user-defined), and *reactors*.
- **Device service provider**: allows local or remote service consumers to access properties, events, and functions.
- **Scheduling and evaluation of reactors**: Upon receipt of an event, fetch all reactors that subscribes to the event and evaluates them. As a result of evaluation, new reactors can be scheduled.
- **Bridge to other software frameworks**: Allows to import functions in AllJoyn framework, subscribe to event in Apple Home Kit, etc.
- **Message bus**: All communication between devices are done in the form of messages. For example, a message can encode many different types of information: event occurrence, function call request, function call response, some system information for bus maintenance.

  For now, use existing tools such as ZeroMQ.

[8] Q: What is the **FAILURE SEMANTICS? What happens when a function or reactor fails in the middle of execution?**

3.1 Architecture of Meadow VM



Event reactor map A map from *events* to *event reactors*. Whenever an event is fetched from an event queue, event reactor map is looked up for associated reactors.

3.2 Example flow: Device property query

Assume that there are two devices, `A` and `B`.

1. **Device `A` sends request**
   (a) `A` wants to access property named `batteryLife` of `B`, e.g. when it executes the code `life = B:batteryLife`.
   (b) This assignment causes `evalExpr` function over the `B:batteryLife`. This, in turn, causes evaluation of `propGet("B", "batteryLife")`.
   (c) The body of `propGet` will create a message which encodes the "get property" request destined to `B`, and put the message into outgoing message queue. Technically, the message request amounts to generation of "builtin event", say `propGetRequest`, which every device (VM) automatically supports (i.e. subscribes).[9]
   (d) Message bus will route the message to `B`. We can use existing message bus libraries, such as ZeroMQ.
   (e) Also, two things are created (or instantiate pre-created ones):
   - Event called `propGetResponse`.[10]

[9] Note that an *event provided by a device can be generated either by itself or from other device*.

[10] **How to differentiate two different `propGetReponse`, one called from during evaluation of function `foo`, the other from `bar`? When two responses come back, how can we pick proper continuation out of two?**

- Reactor that listens to this event, i.e. *continuation* from the point when the value is ready.

2. **Device `B` handles request**
   (a) From the incoming message queue, the message requesting the property of B is retrieved, parsed.
   (b) Already scheduled reactor for `propGetRequest` event is executed. Whose body will create a another message with the value of property as its payload. This message is technically an another event called `propGetResponse`.
   (c) Message is sent to `A`.
3. **Device `A` handles response**
   (a) From the incoming message queue, the message containing the response is retrieved, parsed.
   (b) Already scheduled reactor for `propGetResponse` is executed, whose body is essentially a **continuation** from the point for property query.

3.3 Events vs event reactors

One technical issue to handle is that when there are two reactors to the same event. Let's say there are two functions in a device, `foo` and `bar`. While executing these functions, `foo` executed property queries, `dev0.val0` and `dev1.val1`, respectively. Then both functions are actually subscribing `propGetResponse` event, with its own reactors, say `r0` and `r1` respectively.[11]

When a `propGetResponse` event arrives to this device, which reactor should we evaluate? *This actually amounts to how to emulate "session" in stateless message passing system.* There may exist several ways to handle this. For now, we can use a method based on a **token**, which allows to add *causality* between the request and response.

In particular, request carries a **token**, a unique integer, which will be copied to response message when the servicing device responds, so that the original requester can figure out which request the response corresponds to. That is, when reactor (or reactor instance) is added to reactor queue, we check the token to see if that's what the reactor is waiting for.

This suggests that we might need to revise **event reactor map** such that it will be either a map from (*event, tag*) pairs to reactors. The *tag* can be either a wildcard or a unqiue number. Typical user-generated reactors are wildcard-reactors while some internal reactors which require causality are added as a **tagged reactor**.

3.4 Handling exceptions

In distributed systems, **exceptions and failures are very common.** So, either linguistic and system-wide measures should be provided to handle such situations.

4 The Meadow System Services

In addition to Meadow devices and their services, there are system-wide services. Some are essential services but some other can be provided as paid services.

Services can be implemented in different forms depending on the service. For example, device registry service is most likely implemented as functions in a libray. However, logging service can be implemented a Meadow device which uses a predefined Meadowview library.

[11] Or, we could consider two property queries which is executed from side *two invocations of* `foo`

- **Meadow device registry service**: Maintains a global persistent namespace of devices.[12] Also, maintains information on whether the device is connected to the "system" and how one can talk to the device (e.g. IP address and port).
- **Meadow process service**: Maintains a global persistent namespace of processes. Also, supports instantiation of processes.
- **Meadow device group service**: Meadow devices can be grouped so that some per-group behavior (events, properties, functions, reactors) can be defined.[13]
- **Meadow transaction service**:
- **Meadow concurrency service**:
- **Meadow security service**: Supports authentication, authorization, etc.
- **Meadow logging service**: Allows logging of activities. Needed for auditing purposes.

4.1 Meadow device registry service

The device registry service is implemented as functions in the C library. External C programs or web programs can access these functions to add a new device name.

`int devicereg_register(char *device_name)` Introduces a new device name into the Meadow system. The name must be globally unique and this function is supposed to check its unqiueness.

`int devicereg_unregister(char *device_name)` Removes the device name from the Meadow system.

`int devicereg_exists(char *device_name)` Checks if the device name is already registered or not.

4.2 Meadow process service

The Meadow process service provides interface for management of processes. In particular,

- **management of process definition table**
  - mapping from process name to "compilation result of process definition"
- **management of process lifecycle**
  - **instantiation of process**, which amounts to
    - find compilation result of the given process
    - deploy code to bound devices
  - fire "initial events" which will cause chain reaction of device actions
  - **uninstallation** of processes if needed[14]
- **monitoring process activities**

[12] Q: Should events, properties, functions, reactors managed in a global table? Or should each device which contains such services be the "manager" of such tables? Or hybrid?

[13] Q: implemented as P2P overlay network?

[14] Think more. Maybe quite challenging.

```
int proc_register(char *proc_name, char *proc_def)

int proc_unregister(char *proc_name)

PROC_HANDLE_t proc_lookup(char *proc_name)
```

Meadowview process compiler A Meadowview process compiler takes programs written in *Meadowview process definition language* as its input and produces programs in *Meadowview core language*.

Let a process definition involve n devices. Then, the compilation result is n code fragment, each of which will be deployed to a Meadow device. Technically, it will be a set of

$$(\langle device\_class \rangle, \langle core\_item \rangle)$$

pairs, where $\langle device\_class \rangle$ is a "symbol" which denotes a class of devices and $\langle core\_item \rangle$ is one of the following:

- property definition
- event definition
- function definition
- reactor definition

4.3 Meadow device group service

4.4 Meadow security service

Authorization model Basically, authorization is where a given **subject** is allowed to do [operation on an **object**. In the Meadow system, authorization is performed at the following level:

- **subjects**: device
- **objects**: task, function, property
- **operations**: for now, just add/delete/read/write

4.5 Meadow logging service

5 The Meadow API

The Meadow system provides C++ library which can access devices. This allows external programs to access device services and to get information about the devices and the Meadow system.[15]

5.1 Event service API

```
int event_add(std::string device_name, std::string event_name)

int event_delete(std::string device_name, std::string event_name)
```

[15] **TODO: Currently, no security issues are taken into account. For example, each function should succeed or fail based on some security model. Add it later.**

```
int event_exists(std::string device_name, std::string event_name)

int event_subscribe(std::string device_name, std::string event_name)

int event_unsubscribe(std::string device_name, std::string event_name)

int event_generate(std::string device_name, std::string event_name, VALUE_t
event)
```

5.2 Property service API

```
int prop_add(std::string device_name, std::string prop_name)

int prop_delete(std::string device_name, std::string prop_name)

int prop_exists(std::string device_name, std::string prop_name)

int prop_get(std::string device_name, std::string prop_name, VALUE_t value)

int prop_set(std::string device_name, std::string prop_name, VALUE_t value)
```

5.3 Function service API

```
int function_add(std::string device_name, std::string func_def)

int function_delete(std::string device_name)

int func_call(std::string device_name, std::string func_name, VALUE_t **args,
VALUE_t *rets, HANDLER_t fail)
```

5.4 Reactor API

```
int reactor_add(std::string device_name, std::string reactor_def)

int reactor_delete(std::string device_name, std::string reactor_name)
```

5.5 C value binding API

```
TYPE_t value_typeof(VALUE_t value)

VALUE_t value_encode_int(int value)

VALUE_t value_encode_real(float value)
```

6 Meadow Tools

6.1 Meadow shell

Make a shell which each API function as a “shell command”.

6.2 Meadow interface checker

Meadow devices continue to change its interface through dynamic addition and removal of services. We may want to ensure if the device interfaces conform to code at the given time.

Also, function signature checking can be done for “arities” not “datatypes”.