Draft: Communication for Task-Oriented Systems with Edge Devices

Niek Janssen[0009−0003−7348−7788], Mart Lubbers[0000−0002−4015−4878], and Pieter Koopman[0000−0002−3688−0957]

Institute for Computing and Information Sciences,
Radboud University, Nijmegen, The Netherlands
mart@cs.ru.nl pieter@cs.ru.nl niek.janssen3@ru.nl

Abstract. Implementing communication between edge devices in IoT systems and their server is often a tedious and bug-prone task, since different programming languages with distinct underlying paradigms have to cooperate. The mTask system prevents this semantic friction by providing a single task-oriented framework for the whole system. Communication between tasks on the server and the edge device is the focus of this paper. Shared data sources provide flexible communication between running tasks on the server and the edge device. We introduce an improved version of these shares with a clearer and easier to use semantics. The shares on the edge device offer a strict subset of the functionality of shares on the server. To improve the communication between edge devices and the server, we introduce the possibility to start parameterized server tasks from the edge device.

1 Introduction

Implementing communication between an edge device in an IoT system and a web server is often a tedious and bug-prone task. While most PC or server software is written in higher level languages or frameworks, edge devices are still mostly programmed in a low-level language like C. In this paper, we aim to automate this communication within a task-oriented framework by first using task parameters and results, and later shared data sources, shares for short.

One of the challenges with code involving different types of devices (in our case, an edge device and a web server) is the semantic friction between the different devices [5]. This semantic friction is caused by the different kinds of programming languages and paradigms that are used on web servers and edge devices. Web servers are often programmed in high-level languages and frameworks providing automated memory management, a functional programming paradigm, and other higher-level features. Edge devices, however, are usually programmed in low-level languages such as C. As these languages are vastly different and often have a different underlying paradigm, it is difficult to create code that is semantically exactly the same on each side.

Edge devices are nowadays often connected to the internet, i.e. part of the Internet of Things. These connected devices need to communicate with a web

server, which then provides a user interface for the end user. For the implementation of this communication code, the semantic friction is even more apparent than in the rest of the code base. Firstly, all communication code has to be written twice: once for the server-side paradigm and language, and once for the client-side paradigm and language. These two implementations have to semantically match each other exactly to provide correct communication, while type checking across the two languages is usually not possible. This makes the implementation of communication code prone to bugs. Furthermore, all this duplicate has to be maintained, so implementing bug-free communication does not guarantee that it will stay that way.

The mTask system is a framework that removes this semantic friction by bringing top to edge devices [7]. It integrates directly with iTask, a top system for web servers [16]. With iTask and mTask, a programmer writes a single program of which part is executed on the web server, and part is executed on the edge device. All communication between iTask and mTask is fully automated, although some limitations still have to be overcome. In this paper, we will refer to edge devices running mTask as mTask devices, or just devices.

In section 2, we briefly discuss what top is and how it works. We discuss what iTask and mTask are and how they are used. We create an example implementation of a thermostat as a case study and discuss how communication between the web server and mTask devices can be instigated using task parameters and results.

In section 3, we introduce shares to handle communication. We discuss how these can be used on the server, as well as on the device. We discuss the disadvantage that it is currently not possible to access shares cross-platform between iTask and mTask, and propose changes to make this possible. We do the same for the impossibility to share a shared data source between multiple mTask tasks. Finally, we touch on synchronization and how it can be used to further increase the modularity of a program.

In section 4, we discuss the semantics of shared data sources. We will discuss the already existing semantics for iTask and mTask shares, and see that these semantics hold for cross-platform access. We also see that, in some case, two synchronized shares can semantically be considered the same share.

In section 5, we look at how we implement the features used in section section 3. We conclude that with only one added combinator, it is possible to implement all features discussed in this paper.

1.1 Research contributions

- We provide a systematic motivation for using shared data sources in task-oriented programming.
- We introduce a new design for shared data sources on mTask devices. Their semantics is a strict subset of shared data sources available on iTask servers.
- We introduce a way to invoke parameterized server tasks from mTask devices.

– We show that the semantics of the previous design is easily implementable in the new design.

2 Task-oriented programming in IoT

In this section, we briefly explore top in iTask and mTask. We then implement a simple thermostat program, where we only use task parameters and results for communication. We discuss the problems and limitations we encounter. Shares are introduced in section 3.

2.1 Tasks, iTask and mTask

At the core of top are tasks. Tasks are an abstract representation of work, or in other words, tasks that have to be done. This can be a wide variety of things. For example, in iTask, a task might be: fill in this form using the web interface. An example of an mTask task is: read the temperature using a certain sensor.

Tasks can be combined to make compound tasks. In its simplest form, this is either sequential or parallel. Sequential combinators are also called step combinators. An example use of a step combinator is: let a user fill in this form, *then* display the filled in form values on the screen for confirmation. An example use of a parallel combinator is: let the user fill in this form, and *at the same time* show a countdown for a time limit.

An implementation of top for multi-user distributed web applications is iTask [16, 18]. It is an embedded domain-specific language (DSL) written in the host language Clean [17]. The mTask system is a top implementation for mTask devices, also written as an embedded DSL in Clean. The two systems are integrated using the function `withDevice`. This sets up a connection between an mTask device and the iTask server. Once a connection to a device has been established, the iTask function `liftmTask` is used to execute an mTask task on a connected device. This function compiles the DSL code to bytecode and handles all communication [10]. The mTask device only has to be programmed once with the mTask RTS, a domain-specific operating system. This way, iTask controls what mTask tasks are run on which device, and when. This is called placement [11].

2.2 A simple thermostat

Using iTask and mTask, we create an example implementation of a simple thermostat. We assume we have connection information of an mTask device, which is equipped with a temperature sensor and a heater. This device is controlled from a web server, which sets the target temperature.

In listing 1.1, we create a function `thermostat1`, hosting the iTask code. The user interface is handled completely by iTask, where the user can enter both the connection details for the device and the target temperature that the thermostat should strive towards. We also create a `onDevice` function, hosting the mTask

code. This code takes the temperature set within iTask, and uses a temperature sensor and heater to keep the temperature stable around the target temperature.

```
1  // Initiate the thermometer on the edge device.
2  // Definition is used to keep technical clutter out of the examples
3  dhThermometer = dht (DHT_DHT (DigitalPin D2) DHT22)
4
5  // Create a form field to insert the target temperature
6  temperatureForm temperature =
7      updateInformation [] temperature <<@ Label "Target temperature"
8
9  // All of our examples are implemented as a 30-second loop
10 repeat30 task = rpeatEvery (BeforeSec $ lit 30) task
11
12 thermostat1 :: TCPSettings → Task Bool
13 thermostat1 deviceInfo =
14     temperatureForm 20.0 >>? λtargetTemp →
15     withDevice deviceInfo λdevice →
16     liftmTask (onDevice targetTemp) device
17 where
18     onDevice :: Real → Main (MTask v Bool) | mtask,dht v
19     onDevice targetTemp =
20         dhThermometer λthermometer →
21         declarePin D4 PMOutput λheater →
22
23         { main = repeat30 (
24             temperature thermometer >>~. λcurrentTemp →
25             writeD heater (currentTemp <. lit targetTemp)
26         )}
```

Listing 1.1. Thermostat v1. In this design, the target temperature is set once via iTask, and then maintained using the thermometer and heater on the mTask device.

In the code in listing 1.1, some basic tasks are clearly visible. For example, `temperatureForm`, on listing 1.1 within iTask, displays a form in the user interface. In mTask, the `temperature` task on listing 1.1 uses the `thermometer` to read the current temperature.

On listing 1.1, we also see a step combinator. As soon as the user clicks the submit button on the form, the step combinator executes the rest of the program. Listing 1.1 does not have parallel combinators, but we will see some in listing 1.2.

mTask does not use the same combinators as iTask, but has its own distinct set of combinators. Even though mTask and iTask are integrated well, the implementations of iTask and mTask are separate. Using the same combinators would result in name conflicts. Generally, mTask combinators contain a dot character (`.`) whereas iTask combinators do not. This makes it easy to recognize which is which.

Finally, we use the integration functions. On listing 1.1 we use `withDevice` to make a connection with the device using the connection details, in this case supplied by the function argument. On listing 1.1 employ the `liftmTask` function to create an iTask task from a mTask device and an mTask task.

Communication and limitations The example in listing 1.1 is fairly limited. It is only possible to set the target temperature once, and then the mTask thermostat heats to that temperature forever. During the execution, the target temperature cannot be changed. There is only one form of communication which is going from the server to the device. On listing 1.1 we can see that the variable `targetTemp` is used in the mTask program.

To understand exactly how this form of communication works, we have to understand `liftmTask` and how it runs tasks on an mTask device. In our example in listing 1.1, we see that the mTask program is created in the function `onDevice`. The target temperature is inserted into this representation by calling `lit` on it on listing 1.1. This means that the value of `targetTemperature` is hardcoded, the host language is used as a macro language to construct mTask programs, linguistic reuse [8]. As mTask is a shallowly embedded DSL, the view on the DSL is a bytecode compiler. This is then compiled at runtime into bytecode and sent to the device.

2.3 An interactive thermostat

In the example in listing 1.1, it is only possible to set the target temperature once. We now extend the example so that the user can set the target temperature whenever he wants, and we display the current temperature in the interface. To set the target temperature on the device, we use a function argument again. To retrieve the current temperature from the device, we use the task result.

```
1  // Definition to create a temperature viewing field
2  temperatureShow temperature =
3      viewInformation [] temperature <<@ Label "Current temperature"
4
5  thermostat2 :: TCPSettings → Task ()
6  thermostat2 deviceInfo =
7      withDevice deviceInfo λdevice →
8      mainLoop device (20.0, 0.0)
9  where
10     mainLoop :: MTDevice (Real, Real) → Task ()
11     mainLoop device (targetTemp, currentTemp) =
12         (   liftmTask (onDevice targetTemp) device >>- λcurrentTemp →
13             waitForTimer False 30 >-| return (targetTemp, currentTemp))
14         -||-
15         ((temperatureForm targetTemp -|| temperatureShow currentTemp)
16             >>- λtargetTemp → return (targetTemp, currentTemp))
17         >>- λtemperatures → mainLoop device temperatures
18
19     onDevice :: Real → Main (MTask v Real) | mtask,dht v
20     onDevice targetTemp =
21         dhThermometer λthermometer →
22         declarePin D4 PMOutput λheater →
23
24         { main =
25         temperature thermometer >>~. λcurrentTemp →
26         writeD heater (currentTemp <. lit targetTemp) >>|.
```

```
27          rtrn currentTemp }
```

Listing 1.2. Thermostat v2. Where v1 only allows the user to set the target temperature once, v2 continuously show the interface to set the target temperature and view the current temperature. Every 30 seconds everything refreshes.

In listing 1.2, we have taken the loop out of the mTask program and put it in the iTask program. This means that, instead of looping every 30 seconds, the mTask program is restarted every 30 seconds with a new target temperature. The current temperature is then returned as the task result, and is used in iTask again to be displayed. Aside from using shares, which we introduce in section 3, we have no other way to refresh the interface with the new current temperature then to refresh it every 30 seconds.

The iTask main loop can be seen in listing 1.2 on listings 1.2 to 1.2. We use the `-||-` and `-||` on listing 1.2 to run some parts of the program in parallel. The semantics of the two combinators are slightly different, however.

The `-||-` results in the value of the task that ends first. This means that if the user presses submit on the form, the right task ends and restart both tasks with the new target temperature. If the user does not press submit on the form, the timer expires and the left task ends. Both tasks restart again, but this time with the updated current temperature.

The `-||` on listing 1.2 is a variant, which is only interested in the task result on the left-hand side. This makes sense in our example, as we display the current temperature side by side with the target temperature, but we are only interested in the result of the form. The right-hand side task runs until the left-hand side ends.

As iTask and mTask are single-threaded applications, parallel tasks cannot run truly concurrently. Instead, the semantics of iTask and the parallel combinators run the two tasks interleaved, using small step rewrites.

And finally, there is `liftmTask` followed by a 30-second timer, on listing 1.2. The `liftmTask` function executes a step of the heater and updates the current temperature. The timer makes sure the mTask program is executed at least every 30 seconds. At the same time, on listing 1.2, a temperature form for the target temperature is shown, together with a field displaying the current temperature.

Communication and limitations In the example of listing 1.2, we communicate with the mTask device by continuously stopping and restarting the task on the device, as well as the tasks generating the user interface. This way, we use the task parameters and results to keep the target temperature and current temperature up-to-date in all parts of the program.

In section 2.2, we discussed how mTask programs are constructed and uploaded to the mTask device. We concluded that the target temperature value is hardcoded into the mTask program that is sent to the device. Because mTask is implemented as a shallowly embedded DSL, the type checking is done at compile time by the host programming language Clean. However, the actual compilation is done at runtime. This means, in listing 1.2, every time the main loop gets executed and `liftmTask` is called, the program is recompiled and reuploaded to the

device, with the most up-to-date target temperature hardcoded in the compiled task code. This is problematic for several reasons.

First of all, this is a burden for the programmer. In this case, we needed a relatively complex loop structure with several parallel tasks to perform a relatively simple task. This code needs to be written and maintained by the programmers. Secondly, it is a waste of resources. Every 30 seconds, the entire mTask program is recompiled and sent over the network. Especially on mTask devices, which usually run on batteries, this can accelerate battery depletion. Moreover, some devices are connected by relatively low-bandwidth connections, which has to be shared with other devices. Currently, we are only looking at a basic thermostat implementation which restarts every 30 seconds, but it is easy to imagine systems with more volatile variables, generating many more recompilation cycles.

3 Communication for running tasks

In section 2, we implemented a simple thermostat system using a web server programmed in iTask, and an edge device programmed in mTask. We implemented the communication using task parameters and results, and we concluded that task parameters and results are not sufficient to implement communication between a device and a web server. In this section, we look at another mechanism frequently used in task-oriented programming: shared data sources.

3.1 Shared data sources

When communicating using task parameters and results, parallel running tasks can only communicate by being restarted. Shared data sources allow for communication between parallel running tasks without the requirement of being restarted. In their core principle, shared data sources are only a small interface containing a read and a write function. Shared data sources in iTask have been extended with some extra functionality, like parametric lenses and a notification system [4].

A simple shared data source acting like a variable can be created using the function `withShared`. It takes an initial value for the shared data source, and a callback function with the shared data source as an argument. This shared data source can then be used in child-tasks.

```
1 // New temperature form fields using shares
2 temperatureFormShared sds = updateSharedInformation [] sds
3     <<@ Label "Target Temperature"
4 temperatureShowShared sds = viewSharedInformation [] sds
5     <<@ Label "Current Temperature"
6
7 shares :: Task Real
8 shares = withShared 20.0 λsds →
9         temperatureFormShared sds
```

```
10          -|| temperatureShowShared sds
```

Listing 1.3. This example program shows parallel usage of shares. The `temperatureShowShared` field is continuously updated with the value set using `temperatureFormShared`.

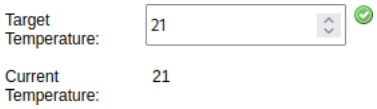

Fig. 1. The interface generated by listing 1.3 is shown here. The user updates the field on top, which is continuously synchronized with the read-only field at the bottom.

For example, in listing 1.3, we have a shared data source used by two parallel child tasks. The child tasks generate an interface containing an input field and a textual written value, as can be seen in fig. 1. The form task updates the value of the shared data source when the form value changes, and the display task updates the displayed value when the value of the share changes. Whenever the form value is changed, the value just below changes with it. This process is continued infinitely, or until it is stopped by a parent task.

These shared data sources are used to improve the modularity and separation of concerns in source code. Take for example our thermostat implementation in listing 1.2. There, we have a single main loop, handling the user interface and mTask device in a big ball of spaghetti. Using a shared data source, we separate these concerns in listing 1.4.

```
1  thermostat3 :: TCPSettings → Task Real
2  thermostat3 deviceInfo =
3      withShared 20.0 λiTargetTempShare →
4      withShared 20.0 λiCurrentTempShare →
5      withDevice deviceInfo λdevice →
6      temperatureFormShared iTargetTempShare -||
7      temperatureShowShared iCurrentTempShare -||
8      mainLoop device iTargetTempShare iCurrentTempShare
9  where
10     mainLoop device iTargetTempShare iCurrentTempShare =
11         (forever $ get iTargetTempShare >>- λtargetTemp →
12         liftmTask (onDevice targetTemp) device >>- λcurrentTemp →
13         set currentTemp iCurrentTempShare >-|
14         waitForTimer False 30)
15
16     onDevice :: Real → Main (MTask v Real) | mtask,dht v
17         // Identical to onDevice in listing 1.2
```

Listing 1.4. Thermostat v3. Where v2 has to restart the interface every 30 seconds to update it, v3 uses shared data sources to automate this task. The mTask device code still has to be restarted every 30 seconds to synchronize the target and current temperatures.

On listing 1.3, we see two new form field macros. These variants handle their input/output via shares instead of retrieving and returning the variables directly, like the ones in listing 1.1. These are now called on listing 1.4, outside the thermostat loop.

On listing 1.4, we create the shares to be used in the temperature form fields and the main loop. In the main loop itself, we only have to regularly run the thermostat using the values present in the shared data sources. We can see on listing 1.4 that the values are retrieved from and updated to the shared data sources.

3.2 Share access from the device

Up to here, we have implemented our thermostat solely using techniques already available in stable versions of iTask and mTask. However, we still have a relatively monolithic implementation, with a big main loop (the `forever` on listing 1.4) starting both the iTask and mTask parts of the code repeatedly. We would like our implementation to be more modular, similar to how `temperatureFormShared` is a completely detached module, whilst still communicating via the shared data source.

To increase modularity, we provide interfacing to access shared data sources cross-platform. Tasks in mTask can now access iTask shared data sources via the functions `iGet` and `iSet`, while iTask can access mTask shared data sources via the functions `mGet` and `mSet`.

This further allows us to separate the communication from the control structure. Communication between the iTask server and the mTask device is no longer dependent on starting or restarting the mTask program.

In the next version of our thermostat, we move the main loop back to the device. This means that the thermostat task on the device only has to be started once, which overcomes the limitations as posed in section 2.3. Whenever we need to access data from the shared data sources, we use `iGet` and `iSet` to do so. As a small optimization, we compare the current value to the previous one, and only update it once the value changes.

```
1  thermostat4 :: TCPSettings → Task Bool
2  thermostat4 deviceInfo =
3      withShared 20.0 λiTargetTempShare →
4      withShared 20.0 λiCurrentTempShare →
5      withDevice deviceInfo λdevice →
6      temperatureFormShared iTargetTempShare ||-
7      temperatureShowShared iCurrentTempShare ||-
8      liftmTask (onDevice iTargetTempShare iCurrentTempShare) device
9  where
10     onDevice :: (Shared sds Real) (Shared sds Real)
11         → Main (MTask v Bool)
12         | RWShared sds & mtask,dht,lowerSds v
13     onDevice iTargetTempShare iCurrentTempShare =
14         dhThermometer λthermometer →
15         declarePin D4 PMOutput λheater →
16         withmTaskShared 20.0 λoldCurrentShare →
```

```
17
18      { main = rpeatEvery (BeforeSec $ lit 30) (
19          temperature thermometer >>~. λcurrentTemp →
20          getSds oldCurrentShare >>~. λoldCurrent→
21          If (currentTemp ==. oldCurrent) (rtrn currentTemp)
22              (iSet iCurrentTempShare currentTemp >-|
23              setSds oldCurrentShare currentTemp) >-|
24          iGet iTargetTempShare >>~. λtargetTemp →
25          writeD heater (currentTemp <. targetTemp)
26      ) }
```

Listing 1.5. Thermostat v4. Where v3 only used shared data sources on the server, we now use shared data sources for all communication.

The function `thermostat4` in listing 1.5 is very similar to the function `thermostat3` in listing 1.4. The only difference is that on listing 1.5 of listing 1.5, the call to the main loop is replaced by `liftmTask`.

Then, in the mTask code, we create a new share using `withmTaskShared` on listing 1.5. We start the device loop using `repeat30` on listing 1.5, and in the loop on listing 1.5 to listing 1.5 we implement our thermostat logic. When we get or set the current or target temperatures on listing 1.5, the data is directly retrieved from or written to the corresponding shared data source on the server.

Communication and limitations In this example, all communication is fully handled by reading to or writing from shared data sources. The shared data sources can be directly used from the server. However, as we do not expect the current temperature to change every 30 seconds, we can save a lot of network traffic (and battery life) by checking whether the value is actually new. We can see on listing 1.5 to listing 1.5 that some scaffolding is necessary to implement this behaviour.

To implement the same behaviour on the server side for the target temperature, we have to take a different approach. On the client side, we have our own `iGet` every time the value changes. However, on the server side, the updating of the share is fully handled using `temperatureFormShared`. We discuss this approach in more detail in section 3.4.

Finally, two of our goals with this approach were to be less monolithic and to use less data over the network. However, if we want to split our task into separate tasks for the thermometer and the heater control, we encounter a problem. As we call `withmTaskShared` from within a task, we cannot access the same shared data source from two different tasks. We are forced to use the iTask share, which operates fully over the network. In section 3.3 we change the implementation of `withmTaskShared` to overcome this limitation.

3.3 Task-independent shares

To overcome the limitations discussed in section 3.2, we need to separate the connections to the shared data sources from tasks. We can see an example of what exactly is the problem in listing 1.6. In this example, we tried to keep the

data local on the device, by using mTask shared data sources. However, each task now has its own shared data source, which means the data is not linked. This is visualized in section 3.3. On the left side we see the situation as in listing 1.6. Each task lifted by `liftmTask` lives in its own little box. All shared data sources it needs are included in this box, and tasks cannot access shares in other tasks' boxes. The right side shows the desired situation, where the connections to shares are made independent of tasks uploaded by `liftmTask`.

```
thermostat5a :: TCPSettings → Task Bool
thermostat5a deviceInfo =
    withShared 20.0 λiTargetTempShare →
    withShared 20.0 λiCurrentTempShare →
    withDevice deviceInfo λdevice →
    temperatureFormShared iTargetTempShare ||-
    temperatureShowShared iCurrentTempShare ||-
    liftmTask (sensor iCurrentTempShare) device ||-
    liftmTask (heater iTargetTempShare) device
where
    sensor iCurrentTempShare =
        dhThermometer λthermometer →
        withmTaskShared 20.0 λmCurrentTempShare →

        { main = repeat30 (
            temperature thermometer >>=. λtemp →
            setSds mCurrentTempShare temp >-|
            iSet iCurrentTempShare temp)}

    heater iTargetTempShare =
        declarePin A4 PMOutput λheater →
        withmTaskShared 20.0 λmCurrentTempShare →

        { main = repeat30 (
            getSds mCurrentTempShare >>~. λcurrent →
            iGet iTargetTempShare >>~. λgoal →
            writeD heater (current <. goal))}
```

Listing 1.6. Thermostat v5a. We attempt to split the code for the thermometer and heater into two different tasks. This implementation fails, as the `mCurrentTempShare` in the heater task does not communicate or synchronize with the one in the thermometer task.

To accomplish this, we have to take the creation of shares out of the mTask code, and create shares in the iTask code instead. Listing 1.7 shows how this is implemented in our example thermostat application.

```
thermostat5b :: TCPSettings → Task Bool
thermostat5b deviceInfo =
    withShared 20.0 λiTargetTempShare →
    withShared 20.0 λiCurrentTempShare →
    withDevice deviceInfo λdevice →
    withmTaskShared device 20.0 λmCurrentTempShare →
    temperatureFormShared iTargetTempShare ||-
    temperatureShowShared iCurrentTempShare ||-
    liftmTask (sensor mCurrentTempShare iCurrentTempShare) device ||-
    liftmTask (heater mCurrentTempShare iTargetTempShare) device
```

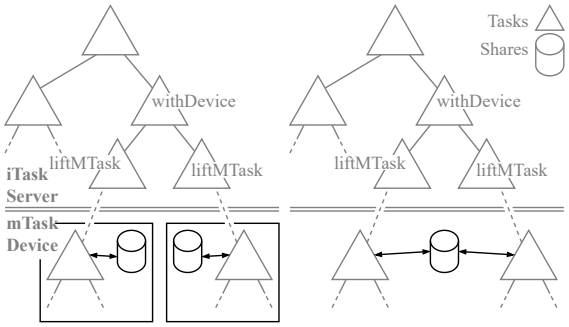

Fig. 2. This diagram shows the setup and communication between iTask on the server and mTask on a device. The left side shows the situation as in listing 1.6, with the situation of listing 1.7 on the right.

```
11 where
12     sensor mCurrentTempShare iCurrentTempShare =
13         dhThermometer λthermometer →
14         /* No share creation here */
15
16         { main = repeat30 (
17             temperature thermometer >>=. λtemp →
18             setSds mCurrentTempShare temp >-|
19             iSet iCurrentTempShare temp)}
20
21     heater mCurrentTempShare iTargetTempShare =
22         declarePin A4 PMOutput λheater →
23         /* No share creation here */
24
25         { main = repeat30 (
26             getSds mCurrentTempShare >>~. λcurrent →
27             iGet iTargetTempShare >>~. λgoal →
28             writeD heater (current <. goal))}
```

Listing 1.7. Thermostat v5b. Where v5a does not have working communication between the two tasks, v5b does.

The thermostat version of listing 1.7 contains a few updates from the previous version in listing 1.5. Firstly, the `onDevice` task has been split into a `sensor` task on listing 1.7, which is responsible for reading the thermometer and updating the current temperature, and a `heater` task on listing 1.7, which is responsible for controlling the heater from the target and current temperatures. Secondly, the creation of the share has been taken out of the individual mTask tasks, and has moved to the iTask code on listing 1.7. Finally, we drop the check to see if the current temperature has changed from the previous one on listing 1.5 to listing 1.5 of listing 1.5, as we discuss this in more detail in section 3.4.

In the thermostat version of listing 1.7, the communication between the device and the web server is implemented as described on the right side of sec-

tion 3.3. Separate tasks on the device can now directly communicate with each other, as well as with the web server.

Communication and limitations This version eliminates the network traffic for communication between running mTask tasks. However, as we see on listing 1.7, we still need to update both the mTask and iTask version of the share. On listing 1.7, we see that every iteration of the temperature check still requires a network call to the iTask share, even though the temperature probably only changes sporadically.

3.4 Synchronizing shares

```
thermostat6 :: TCPSettings → Task Bool
thermostat6 deviceInfo =
    withShared 20.0 λiTargetTempShare →
    withShared 20.0 λiCurrentTempShare →
    withDevice deviceInfo λdevice →
    withmTaskShared device 20.0 λmTargetTempShare →
    withmTaskShared device 20.0 λmCurrentTempShare →
    syncIToM device iTargetTempShare mTargetTempShare ||-
    syncMtoI device mCurrentTempShare iCurrentTempShare ||-
    temperatureFormShared iTargetTempShare ||-
    temperatureShowShared iCurrentTempShare ||-
    liftmTask (sensor mCurrentTempShare) device ||-
    liftmTask (heater mCurrentTempShare mTargetTempShare) device
where
    sensor mCurrentTempShare =
        dhThermometer λthermometer →

        { main = repeat30 (
            temperature thermometer >>=. λtemp →
            setSds mCurrentTempShare temp)}

    heater mCurrentTempShare mTargetTempShare =
        declarePin A4 PMOutput λheater →

        { main = repeat30 (
            getSds mCurrentTempShare >>~. λcurrent →
            getSds mTargetTempShare >>~. λgoal →
            writeD heater (current <. goal))}
```

Listing 1.8. Thermostat v6. Where v5 has to do manual synchronization every time we write to a share, we now have background tasks to automate these tasks.

We now would like to go one step further, and fully separate the thermostat logic from the communication logic. We would also like to optimize our communication logic, so it only requires bandwidth whenever the value actually changes. To accomplish this, we make use of separate data-synchronization tasks `syncItoM` and `syncMtoI`. They take an iTask and an mTask share, and synchronize the value from one side to the other whenever the value actually changes. Their implementation is further discussed in section 5.3. These are simply functions we

can insert alongside the `liftmask`'s and temperature form, in the same manner as we use `temperatureFormShared` in section 3.1 to offload the handling of the user interface. In mTask, we can now focus solely on implementing our device code, knowing the communication is fully handled elsewhere.

In this final version of our thermostat in listing 1.8, we outsource the synchronization of shares between server side and client side to the new tasks `syncItoM` and `syncMtoI`.

We run the synchronization tasks parallel to the rest in listing 1.8. Note that on listing 1.8, we do not need to update the iTask share any more.

Data synchronization does come with a few footnotes. We discuss the exact semantics and limitations of synchronization in section 4.3.

4 Semantics of shared data sources

In this section, we discuss the semantics of shared data sources, cross-platform access to shared data sources, and synchronization of shared data sources as described in section 3. We note that the original semantics for shared data sources hold for cross-platform access, but not for all cases of synchronized shared data sources.

4.1 Semantics of shared data sources

In this section, we look at the informal semantics of the operations `get`, `watch`, `set`, `upd` and `amend`[1]. In the thermostat examples of section 3, we only used `get` and `set`, together with their mTask (`getSds`, `setSds`) and cross-platform (`iGet`, `iSet`, `mGet`, `mSet`) variants. The `upd` and `amend` operations are also available in the mTask and cross-platform variants, using the same naming conventions. Finally, in iTask we have a `watch` operation, this is implemented as a different combinator to `get` as an optimization. The watch is used to observe a shared data source over a period of time. In mTask, the `getSds` includes the functionality of the `watch`. A comparison table of mTask and iTask operations is given in table 1. The exact definitions of the five base operations in iTask are given in listing 1.9.

```
1  // Get the value of a shared data source once
2  // as a stable value
3  get :: (Shared a) → Task a
4
5  // Watch the value of a shared data source continuously
6  // as an unstable value
7  watch :: (Shared a) → Task a
8
9  // Set the value of a shared data source
10 set :: a (Shared a) → Task a
11
```

[1] Note: The full iTask shared data source implementation supports several features that mTask shared data sources do not. For the sake of simplicity, we will only discuss features of iTask shared data sources that mTask supports as well. More info about full iTask shared data sources can be found in papers by Lubbers and Böhm [9] and Domoszlai et al. [4].

```
// Update the value of a shared data source
upd :: (a → a) (Shared a) → Task a

// Update the value of a shared data source, return a custom value
amend :: (a → (b, Maybe a)) (Shared a) → Task b
```

Listing 1.9. We define the type signatures of the operations on shared data sources.

iTask	mTask
`get`	`getSds >>=. rtrn`
`watch`	`getSds`
`set`	`setSds`
`upd`	`updSds`
`amend`	`amendSds`

Table 1. We compare the iTask operations on shared data sources with their variants in mTask.

Given a shared data source of type `Shared a`:

- `get` takes the shared data source and immediately returns the value of the share as a stable value.
- `watch` continuously reads the shared data source, and returns its value as an unstable value.
- `set` takes a value and a shared data source. It writes this value to the shared data source, and returns it as well.
- `upd` takes a transformation function of type `a → a` and a shared data source. It applies the transformation function on the value of the shared data source. The resulting value is both written back to the shared data source, and returned by the `upd` function.
- `amend` is a more general version of the `upd`. It separates the value written into the share and the value returned by the `amend` task utilizing a tuple. The full type of the transformation function is `a → (b, Maybe a)`. The left part of the tuple is returned by the `amend`, whilst the right side is written as a new value to the share. Furthermore, the right side is a `Maybe`. It is possible to omit writing back to the share by returning `Nothing` here.

Furthermore, the following properties hold on all four operations:

- *Atomicity*: The operations are guaranteed to be *atomic*.
  For `get` and `set` this is natural and easy to achieve. However, for `upd` and `amend`, this is an important property which requires a careful implementation. If we were to transform the value of a share using only `get` and `set`, a parallel running task could write to the same shared data source, introducing a race condition.
- *Incompleteness*: It is *not* guaranteed that all tasks observing the shared data source see all changes.

- *Ordering*: It is guaranteed that all tasks observing the shared data source see changes in the same order.
- *Convergence*: After a certain amount of time without any updates, all tasks observing the shared data source see the same value.

To give a bit more intuition for the properties *Incompleteness*, *Ordering* and *Convergence*, let us look at the example of listing 1.10. In this example, we run tasks `a` and `b` in parallel. Each of them sets the value of the share, and then forever reads the share to do something with it.

The *Incompleteness* non-guarantee tells us that it is undefined whether any of these tasks sees both values. It is possible that either, or both, only see the value that is written last. This simply has to do with the fact that there are defined moments on which the task looks at the value of the share. If the value of the share changes twice between two observations, the task misses the first value. When it is imperative that no updates are missed, the programmer can employ a queue in the shared data source.

The *Ordering* property guarantees us that the values that are seen by the tasks are always seen in the same order. It is not possible that task `a` first sees value 12 and then 42, while task `b` first sees 42 and then 12, or vice versa.

The *Convergence* property guarantees that, eventually, both tasks will observe the last written task value.

```
1 both = withShared 0 λshare →
2     a share -&&- b share
3
4 a sds = set 12 sds >-|
5     watch sds >>* // ...
6
7 b sds = set 42 sds >-|
8     watch sds >>* // ...
```

Listing 1.10. We define two iTask tasks to illustrate that both tasks see values in the same order (*Ordering*), but not necessarily all values (*Incompleteness*).

The *Atomicity* property demonstrates the need for an `upd` operation nicely. As an example, we consider a program where multiple parallel running tasks increase a counter by one throughout the code. Were this `+1` implemented by a `get share >>- λ v → set (v+1) share`, we would have a race condition in our code. Implementing it with an `upd` instead, we get `upd ((+) 1) share`. This version performs the whole operation in a single step, avoiding race conditions.

To illustrate the need for `amend` over `upd`, we look at the function `dequeueInShare` in listing 1.11. This example would not be possible to implement using an `upd`, as we need to return the updated `queue` to the share, but we need to return the `dequeuedItem` to the program. This is precisely the functionality the `amend` provides.

```
1 dequeueInShare :: (Shared (Queue a)) → Maybe a
2 dequeueInShare share = amend (λqueue →
3     let (dequeuedItem, updatedQueue) = dequeue queue in
4     (dequeuedItem, Just updatedQueue)
```

```
5 ) share
```

Listing 1.11. The dequeue operation shows the necessity of the `amend` operation.

The `upd` is also not sufficient. We would not be able to write the updated queue back to the shared data source, while returning the dequeued item to the function caller. This illustrates the necessity of the `amend` on top of the `upd`. It is, however, possible to implement the `upd` using the `amend` (see listing 1.12).

```
1 upd :: (a → a) (Shared a) → a
2 upd fn sds = amend (λvalue →
3     let value‘ = fn value in
4     (value‘, Just value‘)
5 ) sds
```

Listing 1.12. It is possible to implement the `upd` operation using the `amend`.

4.2 Semantics of cross-platform access

In the previous section, we discussed the semantics of iTasks and mTasks shared data sources. The properties of *Atomicity*, *Ordering* and *Convergence* are easily guaranteed on these systems, as the host/server is the sole controller. The iTask and mTask runtime systems can tightly control how operations on shared data sources are performed, and ensure these properties.

However, as we combine iTask and mTask, we have a multiprocessor system. If we want to implement this combination in some kind of distributed manner, we run into all kinds of synchronization problems. A more elegant approach is to handle share updates on the runtime system where the share exists. This approach has also been used in distributed iTasks [14, Section 4.2]. However, as mTask shares differ from iTask shares, we cannot simply overload the share access operations like they do.

Let us look at this approach more closely. As an example, we take the `upd` operation. In fig. 3, on the left side, we see a visualization of the `upd` operation using a single runtime (either iTask or mTask) setting. Time is represented on the y-axis. On the x-axis, we have space for two tasks A and B, and a shared data source they can use. At some point, task A wants to perform an `upd` operation. To make the `upd` atomic, the transformation function is taken out of the task, and executed directly on the value of the shared data source. This value is then directly written back into the share, as well as given back to the calling task. Any task B is blocked from accessing the shared data source for as long as the transformation function needs to update the value.

Figure 3 on the left side shows the same process, but in a cross-platform setting. The platforms are separated by a network layer in the middle. In this case, we again take the transformation function out of the task. This time we send it over the network to give it to our own task C running on the side containing the shared data source. This task C then runs the `upd` operation using the transformation function, which is executed like in the single-platform situation (fig. 3, right). Finally, task C sends the task result back to task A, so it can resume execution as well.

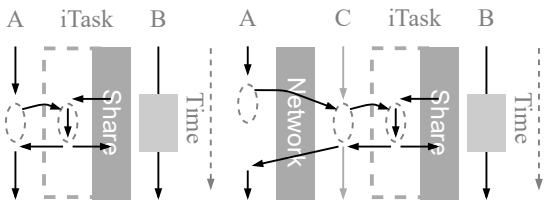

Fig. 3. These sequence diagrams show how an `upd` operation is performed, both on the same device and over the network. On the left side the update is single-platform, while on the right side it is cross-platform.

Let us check whether the semantics indeed hold using this method. The *Atomicity* property holds, as the transformation function is eventually executed as a normal single-platform `upd` operation. The *Ordering* property holds as well, as all changes to the status of the share originate from the single iTask system. They arrive to different parts of the system in the same order, albeit a bit later on the other platform due to networking delays. The same holds for *Convergence*. As our semantics do not restrict timing delays in any shape or form, all semantics still hold.

4.3 Semantics of synchronization

In section 3.4, we introduced synchronization functions `syncItoM` and `syncMtoI` to synchronize two shares existing in iTask and mTask. In this section, we discuss how this synchronization compares to the semantics defined in section 4.1. Can we see these synchronized shares as two different interfaces to semantically one single share? When should caution be used?

One-way synchronization In `thermostat6` as defined in listing 1.8, we see two sets of shared data sources: one for the target temperature and one for the current temperature. For both of these shared data sources, *one way synchronization* is applied. In the example of the target temperature for example, the target temperature is read and written from the server side (by `temperatureFormShared`), and only read from by the heater task on the client side. As such, the value is only synchronized from the server to the client. To generalize: we have read-write side A, read-only side B, and one way synchronization from A to B.

In this setup, the semantics hold and both shares can be considered one single shared data source, as long as side B *only* reads from it. As there is no synchronization from B to A, side A never updates if side B updates. This means we only have to discuss the semantics of `get`, as `upd` and `amend` write to the share like `set` does. The *Atomicity* property holds. As the synchronization function sends updates in the same order as they come into the share on side A, we also preserve the order of updates. The *Ordering* property holds as well. If the update function skips an update, it is not a problem because of the *Incompleteness*

property. Finally, as the *Convergence* property does not restrict timing delays, we only need to ensure that the task value *eventually* ends up on the other side. As this is the exact job description for the synchronization function, this property holds as well.

Another way of looking at this setup is by considering B as a read-only cache for the shared data source. In this case, the updates are pushed proactively to side B by the synchronization function, instead of retrieved from side A on-demand by the cross-platform `get` function.

If a write from side B is still necessary, we can always use a cross-platform write to maintain the semantics.

Two-way synchronization When introducing two-way synchronization, all three of the properties *Atomicity*, *Ordering* and *Convergence* fall apart. For the *Atomicity* property, we can easily create a parallel `upd` on either side of the network, breaking atomicity completely. For the *Ordering* property, any kind of crossing communication from updates on either side of the network breaks the semantics. Each side sees its own update first, followed by the others. When two updates happen on either side on exactly the same time, they will send a message to the other side to synchronize it. They both inherit each other's value, breaking *Convergence* as well.

In this case, we are dealing with *distributed memory* [**?**], with all its race conditions and other synchronization issues. That does not mean, however, that two-way synchronization cannot be useful, as there are still properties we can define on synchronized shares:

- *Propagation* Any update on either side will eventually reach the other side. How long exactly depends on the network connection between the client and server.

This property can be used to implement different synchronization protocols, made to fit the exact situation. For example, we can change the synchronization method slightly to get back the *Convergence* property:

On one side of the synchronization, we change the implementation. Every time a value update comes in which is different from the value already in the shared data source, we update it and send it back to the original side as well.

Synchronization or cross-platform access? Multiple different trade-offs between synchronization and cross-platform access exist. There is no one solution to fit all use cases. In this section, we briefly discuss the differences.

- *Data safety:* The semantic properties hold using cross-platform access or one-way synchronization, but not when using two-way synchronization.
- *Network/battery usage:* When reads are plentiful but writes are sparse, synchronized data uses less network and bandwidth. Even if two-way synchronization is not possible due to data-safety concerns, one-way synchronization can still be used.

In the case where writes are plentiful, but reads are sparse, cross-platform access generally performs better. One might even develop a small protocol where two completely separate shared data sources collect data on either side, only to be combined when the data is read.
- *Read/write speed:* Access to a share on the same side is orders of magnitude faster than cross-platform access over a network. If read-speed is an issue, synchronized data performs better. Write speed is not an issue, as the task continues execution whilie the write is performed.

5 Implementation

In this section, we will discuss the implementation of the techniques above. We will see that the implementation of cross-platform share access only requires one extra combinator `loweriTask`. We will look at `loweritask` and discuss what constructs we require for its implementation. Finally, we show that synchronization in its simplest form also only requires constructs previously defined in this paper.

5.1 Cross-platform access

To perform cross-platform communication safely, we need to execute the read and/or write on the other side of the network. As we discussed in section 4.2 near fig. 3, the safest way to do this while maintaining all guarantees and no code duplication is to insert a task actually performing the `get`, `watch`, `set`, `upd` or `amend` on the side where the share in question lives.

iTask access to mTask shares For iTask to work with mTask shares we already have the `liftmTask` combinator to perform any mTask operation on the device. We can utilize this to transfer our operation to mTask.

```
1  mGet :: MTDevice (BCInterpret (Sds a)) → Task a
2    | type a
3  mGet dev sds = liftmTask {main = getSds sds >>~. rtrn} dev
4
5  mSet :: MTDevice (BCInterpret (Sds a)) a → Task a
6    | type a
7  mSet dev sds a = liftmTask {main = setSds sds (lit a)} dev
8
9  mUpd :: MTDevice ((BCInterpret a) → BCInterpret a)
10     (BCInterpret (Sds a)) → Task a
11   | type a
12 mUpd dev fndef sds = liftmTask (
13   fun λfn=fndef In
14   {main=updSds sds fn}) dev
```

Listing 1.13. Using `liftmTask`, we implement iTask access to mTask shares.

Listing 1.13 shows the implementations for `get`, `set` and `upd`, where the `BCInterpret` type is the mTask compilation monad. For simplicity, we omit the `watch` and `amend`, as they are similar to `get` and `upd` respectively. For `get` and

`set`, the implementation is trivial. We simply wrap the mTask version of the operation in a `liftmTask`. The `upd`, however, needs a function. Because of how functions in mTask are implemented, this adds some complications. To see why, let us look at an example function in listing 1.14.

```
1 fun λfn=(λarg → // Function body
2 ) In // Function fn is callable here
```

Listing 1.14. A function in mTask is defined using the `fun ... In ...` operator pair.

In this example, we see two different notions of the function. Firstly, we have the function definition (`λ arg →` *//function body*). This is a lambda function in the host language, which gets an argument and delivers a function body. It has type `(BCInterpret a) → (BCInterpret b)`: it receives an mTask argument and results in an mTask return value. If we used this directly as a function in mTask code, the full function body would be expanded directly into the calling code every time we call the function.

Secondly, we have `fn`. This notion of function can be considered just the label of the function: it tells the mTask system which function body has to be called when we execute this function. To make sure this function is callable by mTask code, it has type `(BCInterpret a) → (BCInterpret b)`. The observant reader has already seen that this is the same type as the function definition.

Finally, we have the `In` operator. The `fun` and `In` operators together make sure that the function body is compiled, labelled using `fn`, and usable on the right side of the `In`. The current implementation of mTask functions is introduced in an earlier paper on mTask [10, Sections 3.1 & 4.2].

We now create our `mUpd` with the above knowledge. To make sure we are on the right side of the `In`, where we are allowed to use the function, we ask the function definition `fndef` from the mTask programmer and define it as a function ourselves as function `fn`. On the right side of the `In`, we can now use this function in the `updSds`. However, the function definition and the function label both have the same type, which means we cannot use the type system to enforce that the mTask programmer provides the correct one out of those two. This is a limitation of our current approach.

It is possible to avoid these overlapping types by adding a function application combinator to the eDSL language. This addition allows us to change the type of the function label (`fn`) to a boxed variant. The trade-off is that adding a function application combinator pushes the eDSL further away from the host language, causing semantic friction. In our eDSL implementation, we chose to accept this type overlap in favour of not having an application combinator.

mTask access to iTask shares To provide mTask access to iTask shares, we can use a similar approach. We define a `loweriTask` combinator, which is the inverse of `liftmTask`. In section 5.2 we will discuss how such combinator is implemented. We now use this combinator to implement the cross-platform access. As the function we give the `upd` is simply a host language function, we do not need any workaround to make this work as we needed for `mUpd`. The definitions are given in listing 1.15.

```
iGet :: (Shared a) → BCInterpret (TaskValue a)
  | Readable sds & TC a & type a
iGet sds = loweriTask (λ_ → get sds) (lit ())

iSet :: (Shared a) (BCInterpret a) → (BCInterpret (TaskValue a))
  | Writeable sds &  TC a & type a
iSet sds value = loweriTask (λa → set a sds) value

iUpd :: (Shared a) (b a → a) (BCInterpret b)
    → (BCInterpret (TaskValue a))
  | RWShared sds & TC a & type a & type b
iUpd sds fn value = loweriTask (λb → (upd (fn b) sds)) value
```

Listing 1.15. Using `loweriTask`, we implement mTask access to iTask shares.

5.2 loweriTask

To implement cross-platform access as described in section 5.1, we need a `loweriTask` combinator. Such a combinator contacts the server, executes a task there, and synchronizes the task value generated by the iTask task with the mTask task. The signature of loweriTask is given in listing 1.16.

```
loweriTask :: (a → Task b) (BCInterpret a) → BCInterpret (TaskValue b)
```

Listing 1.16. We define the type signature of `loweriTask`.

As we can see, the function expects exactly one argument to be passed to the task. Zero or more than one arguments can be passed by using a unit type or a tuple. This value is sent to the server, along with the request to start the task. Every time the task value changes, it is sent back to the mTask device, and set to be the task result of the `loweriTask` combinator.

To allow for this execution to happen, we need two components: (1) We need to be able to create an iTask task from the mTask system, and (2) we need to communicate with the server to be able to start the task

Creating an iTask task from mTask We want to be able to execute any arbitrary Clean / iTask code from our `loweriTask` combinator. However, mTask is an embedded DSL, which is restricted in its features compared to the host language. We do not want these restrictions on the iTask tasks we want to execute. Instead, we give `loweriTask` a native iTask task, which is then labelled and stored on the server. The mTask program then uses this label to instruct the server on which stored task to execute.

Communicating with the server The `loweriTask` function sends a message to the server to start the iTask task. The iTask server already has an established communication channel with the mTask device to start mTask tasks and obtain task values when completed, which is visualized in the left side of fig. 4. When a connection with a device running the mTask client is made, a communication task is started. This communication task handles the connection with the mTask

client. The device handle then contains a shared data source which is used as a communication channel between the communication task and any other task who needs to communicate with the mTask client. For example, `liftmTask` uses these channels to upload a new mTask program to the mTask device.

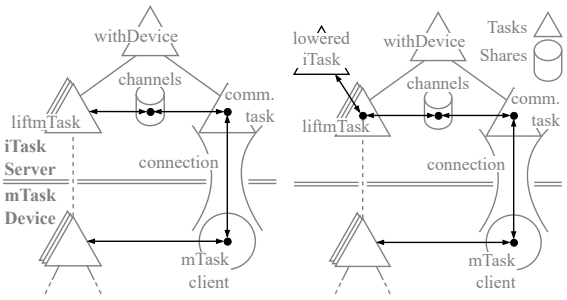

Fig. 4. We show the communication channels used to communicate between iTask and mTask tasks. On the left side, the channels are used to start an mTask task from iTask, while on the right side an iTask task is started from mTask.

We extend this communication infrastructure with the necessary messages for `loweriTask`, as visualized on the right side of fig. 4. As the `liftmTask` function holds all labelled iTask tasks, it listens to task starting requests in the channel's shared data source. Once this task is started, it watches task value updates, and sends them back to the mTask client.

5.3 Synchronization

To synchronize a share on side A to a share on side B, we use the following four-step algorithm:

- *Watching*: Detect when a share on side A is updated
- *Checking*: Test if the value is different from the old one
- *Updating*: Update the share on side B if it is
- *Repeating*: Go back to *watching* and repeat

Let us discuss these steps according to the implementation of this algorithm given in listing 1.17. This implementation synchronizes an iTask share to an mTask share.

The first step of our algorithm is *watching*. We see a `watch` being used on listing 1.17. The watch task has the value of the shared data source as its task result. This watch is followed by the `>>*` step combinator. This combinator rewrites the task on the left-hand side (the `watch`), and checks if a condition on its task result holds. If it does not, it tries again for the next task result. If it does, it will continue with the next task. Such a pair containing a condition and a task is called a

task continuation. As the `>>*` combinator checks for multiple task continuations, it accepts a list of them.

The second step of our algorithm, the *checking*, is embedded in the use of this task combinator. As we check for inequality to the old task value, this task combinator only continues with the next steps of our algorithm when this inequality holds.

The third step of our algorithm, the *updating*, happens on the next line (12) using the `mSet` task to write directly to the mTask share. The *repeating* step on follows listing 1.17, using a recursive call.

Finally, to start our algorithm, we have to retrieve the value of the shared data source, to provide an initial "old" value to the algorithm. This bootstrapping is done on listing 1.17.

```
1  syncIToM :: MTDevice (Shared a) (BCInterpret (Sds a)) → Task ()
2    | RWShared sds & iTask a & type a
3  syncIToM dev isds msds =
4    get isds >>- λv →
5    detectLoop dev isds msds v
6  where
7    detectLoop :: MTDevice (Shared a) (BCInterpret (Sds a)) a
8          → Task ()
9      | RWShared sds & iTask a & type a
10   detectLoop dev isds msds oldValue =
11     watch isds >>* [OnValue $ ifValue ((=!=) oldValue) $ \newValue →
12     mSet dev msds newValue >-|
13     detectLoop dev isds msds newValue]
```

Listing 1.17. `syncIToM` synchronizes the value of an iTask share to an mTask share using the techniques shown in section 3.2.

The implementation to synchronize an mTask share to an iTask share in listing 1.17 is symmetrical to the iTask-to-mTask synchronization. However, this synchronization is implemented in mTask, there are some differences. First of all, as we saw in table 1 in section 4.1, the `getSds` task implements the functionality of both the iTask `get` and `watch`. We see that on `lst:smi:watch`, the `getSds` is used to watch the shared data source. The other differences are merely the usages of mTask combinators instead of iTask combinators. We see the *watching* and *checking* steps on listing 1.18, the *updating* on listing 1.18, and the *repeating* on listing 1.18.

```
1  syncMToI :: MTDevice (BCInterpret (Sds a)) (Shared a) → Task ()
2    | RWShared sds & iTask a & type a & Eq a & basicType a
3  syncMToI dev msds isds =
4    liftmTask (detectLoop msds isds) dev
5  where
6    detectLoop :: (BCInterpret (Sds a)) (Shared a)
7          → Main (BCInterpret (TaskValue ()))
8        | RWShared sds & iTask a & type a & Eq a & basicType a
9    detectLoop msds isds =
10     fun λfn=(λoldValue →
11           getSds msds >>*. [(IfValue $ (!=.) oldValue) \newValue →
12       iSet isds newValue >>|.
13       fn newValue]
```

```
14      ) In
15      { main = getSds msds >>~. λv → (fn v) }
```

Listing 1.18. `syncMToI` synchronizes the value of an mTask share to an iTask share using the techniques shown in section 3.2.

To extend this to two-way synchronization, we simply call both `syncItoM` and `syncMtoI` on the pair of shares, as in listing 1.19.

```
1 syncIandM :: MTDevice (BCInterpret (Sds a)) (Shared a) → Task ()
2     | RWShared sds & iTask a & type a & Eq a & basicType a
3 syncIandM dev msds isds =
4   syncIToM dev isds msds -|| syncMToI dev msds isds
```

Listing 1.19. `syncIandM` synchronizes an mTask share and an iTask share bi-directionally by starting single-directional synchronization tasks in both directions.

6 Future work

In this paper, we have shown how shares can be used to greatly improve communication between an mTask device and a web server. However, there are still some open questions.

Currently, shares are required to be of fixed size. This means that recursive abstract data types like lists/queues/trees are not supported. It is an open problem to see how we can add support for those data types on mTask devices, given the fact that the amount of memory is very limited, and no memory virtualization exists. This could be very useful when we want to implement for example a message queue. This message queue could use an actual queue data structure inside a shared data source.

Secondly, we only discussed one-on-one communication for shares, between a server and a device. Communication between two mTask devices is forced to take a detour via the server. When two devices are on the other end of a low-bandwidth connection, direct device to device communication is preferable. This could also be used for swarm behaviour, or mesh networks of mTask devices.

Finally, some shares may contain big data types which are only partially updated. We can save on a lot of network traffic if we only send deltas of sds updates. This way, queues and records containing a lot of data can be updated way more efficiently.

7 Related work

On smaller IoT devices using microcontrollers, the industry standard for writing applications is the programming language C. The simplest, most bare-bones option for the implementation of communication, is to use HTTP requests or web sockets. Alternatively, a message broker like MQTT or AMQP can be used. These options are explored and compared in a paper by Naik [12]. Compared to the communication methods discussed in this paper, HTTP requests/sockets

and message brokers are as low level and bare-bones as C is compared to higher order languages like Clean.

On bigger IoT devices running a full operating system, any solution that also runs on normal computers and web servers can be employed. While this paper focusses on smaller IoT devices using microcontrollers, the same principles can be applied for bigger IoT devices. Distributed iTask implements proxy access to shared data sources similar to the interface of this paper [14, Section 4.2]. Another possibility is to run a full-fledged distributed memory system like Erlang [1].

mTask provides fine-grained control over what code gets executed on what device, using the `liftmTask` and `loweriTask` combinators. Other systems determine automatically what code runs on the server, and what runs on the client. In contrast to our solution, these systems assume that all nodes are powerful enough to execute any code fragment. A tierless JavaScript project created by Philips et al. [15] uses static code analysis, and inserts remote calls automatically into the code where necessary. JavaScript is an object-oriented language with extensive access to program status, so shared data sources and/or synchronization are not implemented. Potato is a reactive programming solution for IoT problems in the Elixir/Erlang world, using a similar approach [3]. Potato is a specific version of functional reactive programming [**?**]. A (remote) observable in Potato is somewhat similar to a SDS in TOP. The main difference is that an observable produces a stream of values, while a SDS only has a single current value that can change over time.

8 Conclusion

In this paper, we improve our single source solution for communication between edge devices in IoT systems with their server. In the existing solution, the server could spawn tasks on the edge device. The tasks on the server and edge device can communicate via shared data sources during their execution. The two-way communication via a single shared data source has a fuzzy semantics due to the unavoidable communication delays.

In this paper, we introduce separate shares on the server and the device. The interface to these shares on the server and the edge device is very similar. The semantics of the shares on the edge device is a proper subset of the server-based shares.

The server can update or read a shared data source on the device by spawning an appropriate task. This requires the edge device-wide shared data sources introduced here. In the previous system, every task on the edge device had its own copy of the shared data source on the server.

To facilitate easy and efficient communication from edge device tasks to server tasks, the device tasks can invoke a parameterized task on the server. Our examples show that this yields a convenient abstraction level for safe communication.

Unidirectional synchronization from server to device, or vice versa, has still a well-defined semantics. This is easily expressed as a general task in our new

abstraction level. The remote share will reflect any value that lasts long enough with some delay.

All code shown here is implemented in the existing iTask system for the server and mTask system for task-oriented programming. All code show in this paper is available [6].

Bibliography

[1] Joe Armstrong (Ed.). 1996. *Concurrent Programming in ERLANG* (2nd ed ed.). Prentice Hall, London ; New York. https://dl.acm.org/doi/abs/10.5555/229883

[2] James Cheney and Ralf Hinze. 2002. A Lightweight Implementation of Generics and Dynamics. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*. ACM, 90–104. https://doi.org/10.1145/581690.581698

[3] Christophe De Troyer, Jens Nicolay, and Wolfgang De Meuter. 2018. Building IoT Systems Using Distributed First-Class Reactive Programming. In *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, Nicosia, 185–192. https://doi.org/10.1109/CloudCom2018.2018.00045

[4] László Domoszlai, Bas Lijnse, and Rinus Plasmeijer. 2014. Parametric Lenses: Change Notification for Bidirectional Lenses. In *Proceedings of the 26nd 2014 International Symposium on Implementation and Application of Functional Languages*. ACM, Boston MA USA, 1–11. https://doi.org/10.1145/2746325.2746333

[5] Christopher Ireland, David Bowers, Michael Newton, and Kevin Waugh. 2009. A Classification of Object-Relational Impedance Mismatch. In *First International Conference on Advances in Databases, Knowledge, and Data Applications*. IEEE, Cancun, Mexico, 36–43. https://doi.org/10.1109/DBKDA.2009.11

[6] Niek Janssen, Mart Lubbers, and Pieter Koopman. 2024. Source Code for Paper Distributed Data in Task-Oriented Programming on Edge Devices. https://doi.org/10.5281/zenodo.14236133

[7] Pieter Koopman, Mart Lubbers, and Rinus Plasmeijer. 2018. A Task-Based DSL for Microcomputers. In *Proceedings of the Real World Domain Specific Languages Workshop 2018*. ACM, Vienna Austria, 1–11. https://doi.org/10.1145/3183895.3183902

[8] Shriram Krishnamurthi. 2001. *Linguistic Reuse*. Ph. D. Dissertation. Rice University, Houston, USA.

[9] Mart Lubbers and Haye Böhm. 2017. Asynchronous Shared Data Sources. (2017).

[10] Mart Lubbers, Pieter Koopman, and Rinus Plasmeijer. 2021. Interpreting Task Oriented Programs on Tiny Computers. In *Proceedings of the 31st Symposium on Implementation and Application of Functional Languages (IFL '19)*, Jurriën Stutterheim and Wei Ngan Chin (Eds.). ACM, New York, NY, USA, 12. https://doi.org/10.1145/3412932.3412936

[11] Mart Lubbers, Pieter Koopman, Adrian Ramsingh, Jeremy Singer, and Phil Trinder. 2023. Could Tierless Languages Reduce IoT Development Grief? *ACM Transactions on Internet of Things* 4, 1 (Feb. 2023), 1–35. https://doi.org/10.1145/3572901

[12] Nitin Naik. 2017. Choice of Effective Messaging Protocols for IoT Systems: MQTT, CoAP, AMQP and HTTP. In *2017 IEEE International Systems Engineering Symposium (ISSE)*. IEEE, Vienna, Austria, 1–7. https://doi.org/10.1109/SysEng.2017.8088251

[13] Citation Needs. [n. d.]. If You See This, There Is Still a Missing Citation.

[14] Arjan Oortgiese, John Van Groningen, Peter Achten, and Rinus Plasmeijer. 2017. A Distributed Dynamic Architecture for Task Oriented Programming. In *Proceedings of the 29th Symposium on the Implementation and Application of Functional Programming Languages*. ACM, Bristol United Kingdom, 1–12. https://doi.org/10.1145/3205368.3205375

[15] Laure Philips, Coen De Roover, Tom Van Cutsem, and Wolfgang De Meuter. 2014. Towards Tierless Web Development without Tierless Languages. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. ACM, Portland Oregon USA, 69–81. https://doi.org/10.1145/2661136.2661146

[16] Rinus Plasmeijer, Peter Achten, and Pieter Koopman. 2007. iTasks: Executable Specifications of Interactive Work Flow Systems for the Web. *ACM SIGPLAN Notices* 42, 9 (Oct. 2007), 141–152. https://doi.org/10.1145/1291220.1291174

[17] Rinus Plasmeijer and Marko Van Eekelen. 1999. Keep It Clean: A Unique Approach to Functional Programming. *ACM SIGPLAN Notices* 34, 6 (June 1999), 23–31. https://doi.org/10.1145/606666.606670

[18] Rinus Plasmeijer, Bas Lijnse, Steffen Michels, Peter Achten, and Pieter Koopman. 2012. Task-Oriented Programming in a Pure Functional Language. In *Proceedings of the 14th Symposium on Principles and Practice of Declarative Programming (PPDP '12)*. ACM, New York, NY, USA, 195–206. https://doi.org/10.1145/2370776.2370801