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
`niek.janssen3@ru.nl` `mart@cs.ru.nl` `pieter@cs.ru.nl`

Abstract. Implementing communication between edge devices and their server in IoT systems is often a tedious and bug-prone task, since different programming languages with distinct underlying paradigms need to cooperate. The mTask system prevents this semantic friction by providing a single task-oriented framework for the whole system. Shared data sources provide flexible communication between tasks running on the server and tasks running on edge devices, however the current implementation has some drawbacks. We introduce an improved version of these shares with a clearer semantics. To improve the communication between edge devices and the server, we introduce the possibility to start parametrized server tasks from the edge device.

1 Introduction

Implementing communication between an edge device in an IoT system and the server is typically a tedious and bug-prone task. The server software is written in high-level languages or frameworks providing automated memory management, and other high-level features. Edge devices are still mostly programmed in low-level languages like C, due to their tight resource constraints. As these languages are vastly different and lack tooling that covers the entire system, it is difficult to get the server and edge device to cooperate smoothly. The problems caused by the discrepancy between levels of abstraction in the programming languages are called semantic friction [6].

For the 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 and once for the client-side. These two implementations have to semantically complement each other exactly to provide correct communication. Furthermore, all this duplicated code has to be maintained, every change can break this exact match between the two implementations.

To prevent those problems, we use a tierless approach to program IoT systems; the server and edge code is generated from a single high level source [11]. The mTask system brings task-oriented programming (TOP) to edge devices [8]. It integrates directly with iTask, a TOP system for web servers [18]. Both are implemented as shallowly embedded DSL's in the host language Clean [15, 17].

With iTask and mTask, a programmer writes a single program of which a part is executed on the server, and other parts are executed on the edge devices. The existing mTask system clones an iTask share to an edge device. These copies of the share are automatically synchronized when one of them is changed. This is convenient, but race conditions can cause incorrect updates. Moreover, the automatic synchronization can invoke superfluous communication. This paper introduces an improved design where shares are hosted either on the server, or on an edge device. The new system has simple and safe ways to update remote shares by executing a remote task. The ability to execute a task from the server on the edge device was part of the core of mTask. The possibility to start a task on the server from an edge device is new.

In section 2, we discuss the current state of task-oriented programming and shared data sources, focussing on communication. In section 3, we improve on the implementation of shared data sources, using friction points identified in section 2. We discuss cross-platform access of shares, moving shares from task-level to device-level so that they are accessible from multiple tasks, and synchronization between edge device and server.

We then informally discuss some properties of the semantics of shared data sources in section 4. We start with the existing properties of iTasks shared data source semantics, and see that these semantics hold for cross-platform access. We also see that one-way synchronization is semantically safe, whereas two-way synchronization is not.

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. Finally, in section 6 we discuss a bigger example and see how our new shared data sources perform in practice, in terms of possibilities and ease of use.

1.1 Research Contributions

- We introduce a new design for shared data sources on edge devices (section 3) that prevents race conditions.
- The functionality of the new shares is a strict subset of shared data sources available on servers (section 2.3). We sketch the semantic equality of these types of shares in (section 4).
- We introduce a way to invoke parametrized server tasks from the edge devices (sections 5.1 and 5.2). We use this general mechanism to update shares on the server from the edge devices.

2 Task-Oriented Programming for the IoT

In this section, we discuss the current state of task-oriented programming for the IoT. We briefly explore TOP in iTask and mTask. We then implement a simple thermostat program as a vessel to discuss problems and limitations in the current version of mTask. We first focus on task parameters/results and later shared data sources to implement communication.

2.1 Tasks, iTask and mTask

At the core of TOP are tasks. Tasks are an abstract representation of work. 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 are constructed from task primitives or by combining smaller tasks. For many tasks requiring user interaction, a user interface can be generated.

In its simplest form, tasks are combined 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 of a time limit.

In this paper, we use two different implementations of task-oriented programming: iTask and mTask. Both are implemented as a shallow embedded DSL [15] in the host language Clean [17]. For programmers familiar to Haskell, a document comparing the two is available [10]. Whereas iTask is used to create multi-user distributed web applications [18, 19], mTask is used to create applications for edge devices [8]. Due to the tighter resources on edge devices, the language of mTask is more restricted. As iTask sends mTask programs to embedded devices, and both languages are written as embedded DSL's in Clean, a full system containing iTask and mTask code can be written in a single source file.

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

The implementations of mTask and iTask are fully separated. As they each have their own combinator implementation, we cannot use the same combinator symbols to implement for example the sequential and parallel combinators. To distinguish them and avoid naming conflicts, mTask combinators usually contain a period character (.), whereas iTask combinators do not.

2.2 A Simple Thermostat

Using iTask and mTask, we create an example implementation of a simple thermostat. We assume an mTask device exists, which is equipped with a temperature sensor and a heater. In all our examples, a `TCPSettings` object is provided, containing all necessary information to create a connection with the embedded device. This device is controlled from a web server, which sets the target temperature. This example is used to introduce iTask and mTask gradually, so no shared data sources are used yet. Both the iTask and mTask code are implemented in the same source code file; iTask sends the mTask bytecode to the device over the network.

```
thermostat1 :: TCPSettings → Task Bool
thermostat1 deviceInfo =
    updateInformation [] 20.0 >>? λtargetTemp →
```

```
4     withDevice deviceInfo λdevice →
5     liftmTask (onDevice targetTemp) device
6  where
7     onDevice :: Real → Main (MTask v Bool) | mtask,dht v
8     onDevice targetTemp =
9         dhThermometer λthermometer →
10        declarePin D4 PMOutput λheater →
11        { main = rpeatEvery (BeforeSec $ lit 30) (
12            temperature thermometer ≫∼. λcurrentTemp →
13            writeD heater (currentTemp <. lit targetTemp)
14        )}
```

Listing 1. In the design of thermostat v1, the target temperature is set once via iTask, and then maintained using the thermometer and heater on the mTask device.

In Listing 1, we create a function `thermostat1`, hosting the iTask code. The function `updateInformation` on line 3 generates a user interface on the server, where the user can edit the target temperature that the thermostat should strive towards. We also create an `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. The difference between iTask and mTask is visible in their types: whereas iTask tasks have type `Task a` (as visible in the type of `thermostat1`), mTask tasks have type `Main (MTask v a)` (as visible in the type of `onDevice`).

In Listing 1, some basic tasks are clearly visible. For example, on line 3 the `updateInformation` in iTask, displays a web form to the user. In mTask, the `temperature` task on line 12 uses the `thermometer` to read the current temperature.

On line 3, we also see a sequential combinator. As soon as the user clicks the submit button on the form, the sequential combinator executes the rest of the program. Listing 1 does not have parallel combinators, but we see some later in the paper.

The function `withDevice` on line 4 sets up a connection between the server running iTask and an edge device running the mTask runtime system. The `liftmTask` function on line 5 compiles an mTask program and starts it on a connected edge device. This way, iTask controls the placement of any mTask device [11].

An Interactive Thermostat The example in Listing 1 is nice, but we can only set the temperature once. The thermostat then executes the loop to maintain this temperature indefinitely. If we want a thermostat where the user can change the target temperature at any time, and also see the current temperature, we end up with a relatively convoluted and cumbersome implementation. The idea is that we keep restarting the task running on the edge device using a new target temperature, and use the task value to communicate back the current temperature. The iTask main body necessary to accomplish this is shown in Listing 2

```
1  /* Part of thermostat v2 */
2     mainLoop :: MTDevice (Real, Real) → Task ()
```

```
mainLoop device (targetTemp, currentTemp) =
    (   liftmTask (onDevice targetTemp) device >>- λcurrentTemp →
        waitForTimer False 30 >-| return (targetTemp, currentTemp))
    -||-
    ((updateInformation [] targetTemp -|| viewInformation [] currentTemp)
        >>- λtargetTemp → return (targetTemp, currentTemp))
    >>- λtemperatures → mainLoop device temperatures
```

Listing 2. Where thermostat 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.

Communication and Limitations To understand why this implementation is so cumbersome, we first look at the communication present in the previous example thermostat 1 in Listing 1. In this example, we only communicate the target temperature to the device, by embedding it into the mTask program. We see on line 13 that `lit targetTemp` is used to lift the target temperature into the mTask domain. The mTask language is implemented as a shallow embedded DSL. When this DSL is executed at runtime, the mTask code is compiled into bytecode and sent to the mTask device. The target temperature is embedded into this bytecode as a constant. Once the task ends, the result of the mTask task becomes the result of the `liftmTask` combinator on line 5 in Listing 1. This mechanism could be leveraged to communicate a value back to the server.

One major limitation of this approach is the lack of modularity. As we can see in the main loop in Listing 2, program logic is interleaved with communication logic and the creation of user interfaces. Another limitation of this approach is the continuous recompilation and re-uploading of the mTask task.

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 that run on batteries, this accelerates 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.

2.3 Shared Data Sources

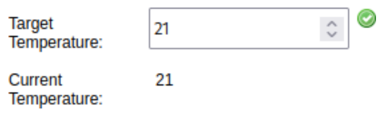
When communicating using task parameters and results, parallel running tasks only communicate by being restarted. Shared data sources allow for communication between parallel running tasks without the requirement of being restarted. In iTask, shared data sources are only a programmer interface containing a read and a write function. Since their first implementation, they have been extended with some extra functionality, like parametric lenses and a notification system [4]. Shared data sources in mTask are a strict subset of shared data sources in iTask. Everything that is not supported by mTask, is outside the scope of this paper.

In mTask, only simple shared data sources are supported that act as type-safe data storages. Neither parametric lenses nor the notification system are available. Furthermore, mTask shared data sources are limited to fixed-size data structures.

In iTask, such a simple shared data source acting like a variable is 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.

```
shares :: Task Real
shares = withShared 20.0 λsds →
        updateSharedInformation [] sds -||
        viewSharedInformation [] sds
```

Listing 3. This small iTask program uses shared data sources.


Target Temperature: 21

Current Temperature: 21

Fig. 1. This interface is being generated by the program on the left.

For example, in Listing 3, we have a shared data source used by two parallel tasks. The tasks generate an interface containing an input field and a numerical 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 shared data source changes. Whenever the form value is changed, the value below changes with it. This process continues indefinitely, 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 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 4.

```
thermostat3 :: TCPSettings → Task Real
thermostat3 deviceInfo =
    withShared 20.0 λiTargetTempShare →
    withShared 20.0 λiCurrentTempShare →
    updateSharedInformation [] iTargetTempShare -||
    viewSharedInformation [] iCurrentTempShare -||
    withDevice deviceInfo λdevice →
    mainLoop device iTargetTempShare iCurrentTempShare
where
    mainLoop device iTargetTempShare iCurrentTempShare =
        (forever $ get iTargetTempShare >>- λtargetTemp →
        liftmTask (onDevice targetTemp) device >>- λcurrentTemp →
        set currentTemp iCurrentTempShare >-|
        waitForTimer False 30)

    onDevice :: Real → Main (MTask v Real) | mtask,dht v
```

```
17        // Identical to onDevice in Listing 2
```

Listing 4. Whereas thermostat 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 lines 5 and 6, we see the forms being created and linked to their respective shared data sources. All input/output of these form fields are handled directly via shared data sources. The form is now created outside the thermostat loop.

On lines 3 and 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 see on lines 11 and 13 that the values are retrieved from and updated to the shared data sources.

3 Improving Shared Data Sources

Up to here, we have implemented our thermostat solely using techniques already available in iTask and mTask. However, we still have a relatively monolithic implementation, with a big main loop (the `forever` on line 11) starting both the iTask and mTask parts of the code repeatedly. We would like our implementation to be more modular, similar to how `updateSharedInformation` detaches the form logic from the edge device logic. In this section, we introduce a new design for shared data sources and use it to implement communication between the server and the device.

3.1 Cross-platform shared data source Access

To accomplish this detachment of iTask and mTask logic, we introduce interfacing to access shared data sources cross-platform. We provide the functions `iGet` and `iSet` to access iTask shares from within mTask. To access mTask shares from within iTask, we provide `mGet` and `mSet`. Overloading the existing `get` and `set` methods is not possible, as `mGet` and `mSet` need the extra device argument. 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 no longer restart the thermostat task every 30 seconds. Instead, we provide the device with a task that uses the shared data source to update the target temperature. This way, we overcome the limitations posed in section 2.2. We use our newly introduced cross platform access functions where necessary. As a small optimization, we compare the current value to the previous one, and only update it once the value changes.

```
thermostat4 :: TCPSettings → Task Bool
thermostat4 deviceInfo =
    withShared 20.0 λiTargetTempShare →
    withShared 20.0 λiCurrentTempShare →
```

```
5     updateSharedInformation [] iTargetTempShare ||-
6     viewSharedInformation [] iCurrentTempShare ||-
7     withDevice deviceInfo λdevice →
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        { main = rpeatEvery (BeforeSec $ lit 30) (
18            temperature thermometer >>~. λcurrentTemp →
19            getSds oldCurrentShare >>~. λoldCurrent→
20            If (currentTemp ==. oldCurrent) (rtrn currentTemp)
21                (iSet iCurrentTempShare currentTemp >>|.
22                setSds oldCurrentShare currentTemp) >>|.
23            iGet iTargetTempShare >>~. λtargetTemp →
24            writeD heater (currentTemp <. targetTemp)
25        ) }
```

Listing 5. Where thermostat v3 only used shared data sources on the server, v4 uses shared data sources for all communication.

The function `thermostat4` in Listing 5 is very similar to `thermostat3` in Listing 4. The only difference is that on line 8 of Listing 5, the call to the main loop is replaced by `liftmTask`.

Then, in the mTask code, `withmTaskShared` is used to create a new shared data source on line 16. We start the device loop using `rpeatEvery` on line 17, and in the loop on line 18 to line 24 we implement our thermostat logic. When we get or set the current or target temperatures on lines 21 and 23, 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 have saved a lot of network traffic (and battery life) by checking whether the value is actually new. We can see on line 19 to line 22 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 shared data source is fully handled using `updateSharedInformation`. We discuss this approach in more detail in section 3.3.

Finally, one of the goals we described at the end of section 2.2, was to have a less monolithic implementation. However, if we try to split our code into separate tasks for the thermometer and heater control, we encounter a problem. As we create a shared data source from within a task, the scope of that shared data source is limited to that task only. In another task, we can create a separate shared data source, but all communication between them has to go via the server, even though they exist on the same device. This approach causes a lot of network traffic, which is also something we want to avoid.

3.2 Task-independent Shares

To get a grasp of why exactly it is not possible to fully separate communication logic, we consider fig. 2. On the left hand side, we see the current situation, where shared data sources are created by and belong to a certain mTask task. This creates a scope, containing a task along with all the shares it needs. It also illustrates the inability for any other task to access these shares. As stated before, this situation is undesirable.

We now lift the creation of shared data sources out of the task creation, to the iTask domain. Once such a shared data source is created, any task can access these shares, as illustrated by the right hand side of fig. 2.

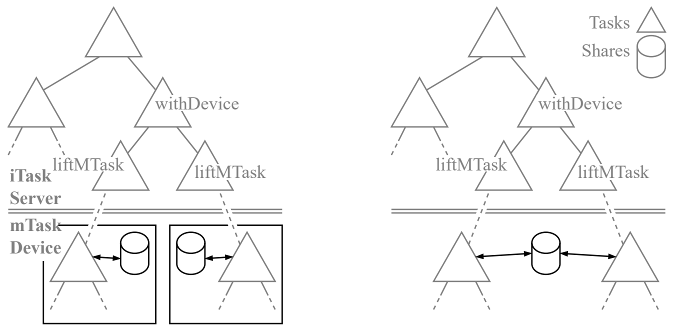


Fig. 2. This diagram visualizes the communication between the server and a device. The left hand side displays the old variant with a limited scope for shared data sources. The right hand side displays the new design, which is demonstrated in Listing 6.

To come back to our thermostat example, this change now allows us to create separate tasks for the heater and temperature controllers. This reduces the monolithicness of our implementation, and improves maintainability and the separation of concerns.

```
thermostat5 :: TCPSettings → Task Bool
thermostat5 deviceInfo =
    withShared 20.0 λiTargetTempShare →
    withShared 20.0 λiCurrentTempShare →
    updateSharedInformation [] iTargetTempShare ||-
```

```
6     viewSharedInformation [] iCurrentTempShare ||-
7     withDevice deviceInfo λdevice →
8     withmTaskShared device 20.0 λmCurrentTempShare →
9     liftmTask (sensor mCurrentTempShare iCurrentTempShare) device ||-
10    liftmTask (heater mCurrentTempShare iTargetTempShare) device
11 where
12    sensor mCurrentTempShare iCurrentTempShare =
13        dhThermometer λthermometer →             /* No share creation here */
14        { main = rpeatEvery (BeforeSec $ lit 30) (
15            temperature thermometer ≫=. λtemp →
16            getSds mCurrentTempShare ≫~. λoldCurrent →
17            If (currentTemp ==. oldCurrent) (rtrn temp)
18                (iSet iCurrentTempShare temp ≫|.
19                setSds mCurrentTempShare temp)}
20
21    heater mCurrentTempShare iTargetTempShare =
22        declarePin A4 PMOutput λheater →         /* No share creation here */
23        { main = rpeatEvery (BeforeSec $ lit 30) (
24            getSds mCurrentTempShare ≫~. λcurrent →
25            iGet iTargetTempShare ≫~. λgoal →
26            writeD heater (current <. goal))}
```

Listing 6. Whereas thermostat v4 had to be implemented as a single mTask task, it is now possible to separate tasks for the heater and temperature controllers due to the possibility of accessing a shared data source from multiple tasks.

The thermostat version of Listing 6 contains a few updates from the previous version in Listing 5. Firstly, the `onDevice` task has been split into a `sensor` task on line 12, which is responsible for reading the thermometer and updating the current temperature, and a `heater` task on line 21, which is responsible for controlling the heater from the target and current temperatures. Secondly, the creation of the shared data source has been taken out of the individual mTask tasks, and has moved to the iTask code on line 8.

Communication and Limitations The current version allows us to separate the concerns of the heater control and the temperature sensor. However, as we see on lines 18 and 19, we still need to update both the mTask and iTask version of the share. On line 25, we see that every iteration of the temperature check still requires a network call to the iTask shared data source, even though the temperature probably only changes sporadically. The communication logic is still interleaved with the two controllers.

3.3 Synchronizing Shares

We now would like to go one step further, and fully separate the thermostat logic from the communication logic. We provide the synchronization tasks `syncItoM` and `syncMtoI`. These functions take an iTask and an mTask shared data source, and synchronize the value from one side to the other whenever the value changes. Their implementation is further discussed in section 5.3. These are simply functions we can insert alongside the temperature, heater and form tasks, in the same manner as we use `updateSharedInformation` in section 2.3 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 7, 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 lines 10 and 11. Note that on line 19, we do not update the iTask shared data source anymore. Data synchronization does come with a few caveats. 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 informal semantics of shared data sources and how they relate to cross platform access and synchronization, as described in section 3. We define some properties on these semantics, and show that they hold for cross-platform access, but not for all cases of synchronized shared data sources. The full formal semantics are outside the scope of this paper.

```
 1 thermostat6 :: TCPSettings → Task Bool
 2 thermostat6 deviceInfo =
 3     withShared 20.0 λiTargetTempShare →
 4     withShared 20.0 λiCurrentTempShare →
 5     updateSharedInformation [] iTargetTempShare ||-
 6     viewSharedInformation [] iCurrentTempShare ||-
 7     withDevice deviceInfo λdevice →
 8     withmTaskShared device 20.0 λmTargetTempShare →
 9     withmTaskShared device 20.0 λmCurrentTempShare →
10     syncIToM device iTargetTempShare mTargetTempShare ||-
11     syncMtoI device mCurrentTempShare iCurrentTempShare ||-
12     liftmTask (sensor mCurrentTempShare) device ||-
13     liftmTask (heater mCurrentTempShare mTargetTempShare) device
14 where
15     sensor mCurrentTempShare =
16         dhThermometer λthermometer →
17         { main = rpeatEvery (BeforeSec $ lit 30) (
18             temperature thermometer ≫=. λtemp →
19             setSds mCurrentTempShare temp)}
20
21     heater mCurrentTempShare mTargetTempShare =
22         declarePin A4 PMOutput λheater →
23         { main = rpeatEvery (BeforeSec $ lit 30) (
24             getSds mCurrentTempShare ≫∼. λcurrent →
25             getSds mTargetTempShare ≫∼. λgoal →
26             writeD heater (current <. goal))}
```

Listing 7. Whereas thermostat v5 had to do manual synchronization every time we write to a shared data source, v6 have background tasks on lines 10 and 11 to automate these tasks.

4.1 Semantic properties on Shared Data Sources

We now look at the informal semantics of the operations `get`, `watch`, `set` and `upd` [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. We introduce the full set of operations on shared data sources first.

- `get` takes the shared data source and returns its value.
- `watch` continuously reads the shared data source, and updates the task result with the value. Opposed to all other operations on shared data sources, this task does not terminate immediately, but only when it is discarded.
- `set` takes a value and a shared data source. It writes this value to the shared data source once, and returns it as well.
- `upd` takes a transformation function of type `a → a` and a shared data source of type `a`. It applies the transformation function to 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.

The type signatures of operations on shared data sources in iTask, and the mTask counterpart, are found in Listing 8. Note that `get` is a special case of `watch`. In iTask, the choice has been made to specify both operations, where `get` is an optimized version that cannot be used to perform a `watch`. In mTask, the `get` is used for both.

```
// Given a shared data source of type (Shared a):
/* iTask signature */                         /* mTask variant */
  get :: (Shared a) → Task a                  // getSds
  watch :: (Shared a) → Task a                // getSds
  set :: a (Shared a) → Task a                // setSds
  upd :: (a → a) (Shared a) → Task a          // updSds
```

Listing 8. We define the type signatures of the operations on shared data sources.

We define five properties on the shared data sources as implemented in iTask:

- *Atomicity*: The operations are guaranteed to be *atomic*. While reading, writing, or updating, it is guaranteed that no other task can access the share. This is especially important for the `upd` operation.
- *Incompleteness*: It is *not* guaranteed that all tasks observing the shared data source see all changes. If two updates happen before a task gets a chance to observe the share, the first update is missed.
- *Ordering*: It is guaranteed that all tasks observing the shared data source see changes in the same order.
- *Convergence*: After a finite amount of time without any updates, all tasks observing the shared data source see the same value.

[1] The full iTask shared data source implementation supports several features that mTask shared data sources do not. For the sake of simplicity, we 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 Böhm [2], Domoszlai et al. [4].

To give a bit more intuition for the properties *Incompleteness*, *Ordering* and *Convergence*, let us look at the example of Listing 9. In this example, we run tasks `a` and `b` in parallel. Each of them sets the value of the shared data source, and then forever reads the shared data source 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 shared data source changes twice between two observations, the task misses the first value. When it is important that no updates are missed, the programmer can employ a queue in the shared data source.

The *Ordering* property guarantees 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.

```
both = withShared 0 λshare →
    a share -&&- b share

a sds = set 12 sds >-|                          b sds = set 42 sds >-|
    watch sds ≫* /* ... */                          watch sds ≫* /* ... */
```

Listing 9. 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 the `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 [13]. Implementing it with the atomic `upd` operator instead, we get `upd ((+ ) 1) share`. This version performs the whole operation in a single step, avoiding race conditions.

4.2 Semantics of Cross-Platform Access

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

However, as iTask is run on a server and mTask is run on an embedded device, a system employing both is inherently a multiprocessor system. If we want to implement this combination in some kind of distributed manner, we run into all kinds of synchronization problems. The old implementation of shared data sources for mTask has exactly this problem. A more elegant approach is to handle updates of the shared data source on the runtime system where the shared data source exists. This is the approach we used in section 3.1. It has also been used in distributed iTasks [14, Section 4.2]. However, as mTask shares differ from iTask shares, we cannot simply overload the shared data source 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 hand 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. As any kind of parallelism is implemented single-threadedly by the iTask combinators, the `upd` function can, in this case, control exactly what gets executed before other tasks get execution time again. The programmer gives it a transformation function, and the `upd` operation performs the get, function and set in one single rewrite step. The result of the function is also used as the task result.

Figure 3 on the right hand 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 provided by the programmer. 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 with 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.

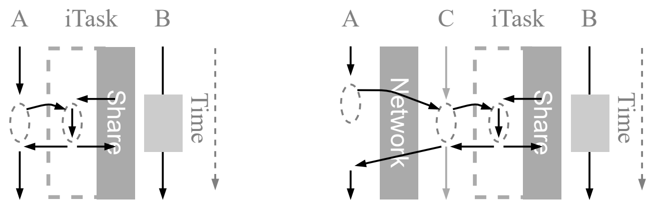


Fig. 3. These sequence diagrams show how an `upd` operation is performed. The left hand side has tasks A and B running on the same device, while the right hand side performs the `upd` over the network.

Using this approach, we guarantee that all semantics introduced in section 4.1 hold. All operations on shares are performed on the single-threaded system where the shared data source lives, as if it were a single-core system. There is, of course, a networking delay in reading or writing a shared data source, but our semantics do not disallow such delay. The *Atomicity* property is guaranteed directly in this way. However, for *Ordering* we do require that messages do not get reordered in the network connection, and for *Convergence* we require that messages always eventually arrive. These requirements are easily fulfilled by using a communication protocol that gives us such guarantees, such as TCP or MQTT. The mTask implementation currently supports either of those.

4.3 Semantics of One-Way Synchronization

In section 3.3, we introduced synchronization functions `syncItoM` and `syncMtoI` to synchronize two shares existing in iTask and mTask. This synchronization is useful only if reading from the synchronized shared data source is semantically equivalent to reading from the original share. In this section, we show that this equivalence holds for one-way synchronization. For two-way synchronization, it is much harder and much more costly to ensure these semantics. These synchronization issues are well documented and out of the scope of this paper [13]. As we use one-way synchronization only, writing has to be done to the *original* shared data source. For convenience, we define the original shared data source as side A, and the synchronized share as side B.

We now check our synchronization from side A to B against the semantic guarantees defined in section 4.1. Writing to the shared data source is only possible on side A. As side A is a shared data source, all semantics for shared data sources hold on side A automatically. For side B, only the semantics for read operations have to hold. *Atomicity* holds, as reading from side B is still atomic. *Ordering* holds as well, as the semantics of cross platform shared data source access ensure this property. The same holds for *Convergence.*

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 the shared data source on side A, so that the semantics are preserved.

5 Implementation

In this section, we discuss the implementation of the techniques above. We discuss that the implementation of cross-platform shared data source access only requires one extra combinator `loweriTask`. We 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, the safest and simplest way to do this while maintaining all guarantees and no code duplication is to insert a task actually performing the `get`, `watch`, `set` or `upd` on the side where the shared data source 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.

```
mGet :: MTDevice (BCInterpret (Sds a)) → Task a | type a
mGet dev sds = liftmTask {main = getSds sds ≫~. rtrn} dev

mSet :: MTDevice (BCInterpret (Sds a)) a → Task a | type a
mSet dev sds a = liftmTask {main = setSds sds (lit a)} dev

mUpd :: MTDevice ((BCInterpret a) → BCInterpret a) (BCInterpret (Sds a))
    → Task a | type a
mUpd dev fndef sds = liftmTask (
  fun λfn=fndef In
  {main=updSds sds fn}) dev
```

Listing 10. Using `liftmTask`, we implement iTask access to mTask shares.

Listing 10 shows the implementations for `get`, `set` and `upd`. The `BCInterpret` type is the mTask compilation monad. For simplicity, we omit the `watch`, as it is similar to `get`. 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, which needs to be provided by the programmer. Due to how functions definitions are implemented in the shallow embedding of mTask, this function needs to be defined in the same mTask program as it is used, so the programmer only needs to give the function body. The exact semantics of mTask functions are described by Lubbers [9, Section 5.3.2].

mTask Access to iTask Shares To provide mTask access to iTask shares, we use an approach that is inverse of the approach of providing iTask access to mTask shares. For this, we need an operator inverse of `liftmTask`, which we define in section 5.2. Listing 11 how we use this combinator to implement iTask shared data source access from mTask.

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

5.2 loweriTask

As mentioned in section 5.1, we need a `loweriTask` combinator to implement mTask access to iTask shares. 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 12.

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

Listing 12. We define the type signature of `loweriTask`.

The function to be executed in iTask (the first argument) 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. If we were to implement functions to execute on the server using the mTask DSL, these restrictions would apply. To circumvent this, we give `loweriTask` a task implemented in iTask instead, 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 hand 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.

We extend this communication infrastructure with the necessary messages for `loweriTask`, as visualized on the right hand 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. Note that the iTask/mTask programmer does not see any of this; All communication is abstracted away.

5.3 Synchronization

In this section, we provide an implementation of the one-way synchronization as discussed in section 3.3. We rely heavily on the cross-platform shared data

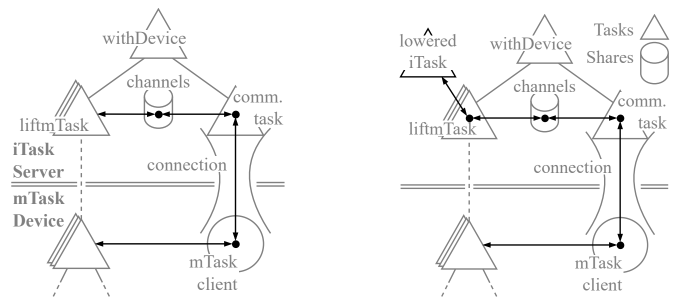


Fig. 4. We show that the existing communication channels can be reused. The left hand side displays the communication channels used for `liftmTask`. The right hand side reuses these channels to implement `loweriTask`.

source access for the implementation. A one-way synchronization algorithm is relatively straightforward. We watch the shared data source for updates. As an optimization, we check if the updated value is actually different from the previous one. If it is, we update the share on the other side to this new value. The algorithm to synchronize an iTask share with an mTask shared data source is given in Listing 13.

```
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 → Task ()
8      | RWShared sds & iTask a & type a
9    detectLoop dev isds msds oldValue =
10     watch isds ≫* [OnValue $ ifValue ((=!=) oldValue) $ \newValue →
11     mSet dev msds newValue >-|
12     detectLoop dev isds msds newValue]
```

Listing 13. `syncIToM` synchronizes the value of an iTask shared data sourceto an mTask shared data source using the techniques shown in section 3.1.

We see a `watch` being used on line 10 to detect updates of the shared data source on the iTask side. This is followed by a ≫* step combinator. The ≫* is a generalized step combinator, which allows choosing for different task continuations based on certain conditions. We use it here to step only if the value of the shared data source is actually different from the old one. If it is, we continue execution and update the shared data source on the mTask side on line 11. Finally we recursively start the detection algorithm using a recursive call on line 12.

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 line 4.

Again, to synchronize an mTask shared data source to an iTask shared data source, the implementation is inverse of the synchronization of an iTask shared data source to an mTask shared data source, where the iTask combinators are exchanged for mTask combinators.

6 Case study

In this section, we implement a system monitoring air quality as a case study. The example makes use of a few different methods of working with shared data sources across iTask and mTask. The mTask device is equipped with a DHT sensor and an air quality sensor. The DHT sensor measures temperature and humidity. The air quality sensor measures CO2, but needs to be calibrated using the temperature and humidity.

We create two tasks `tempAndHum` (line 25) and `co2sensor` (line 38), that respectively read the temperature, humidity and co2 level from their respective sensors. The `co2sensor` task uses the temperature and humidity values to calibrate the co2 sensor.

Any time the temperature or co2 level strays too far from their optimal values, an alarm level is increased. An interface to set these optimal values, as well as view the alarm level and co2 level, is implemented in the function `interface` (line 17). The functions `updateInformation` (lines 19 and 20) and `viewInformation` (lines 22 and 23) provide this functionality.

The shared data sources necessary to communicate between tasks are defined in the main function `start` (line 1). They consist of:

- `iOptTempSds`, `mOptTempSds`, `iMaxCO2Sds` and `mMaxCO2Sds` (lines 4 and 5) form two pairs of synchronized shares, created by `withDoublyShared`. Because of the `ItoM`, the value of the iTask share is synchronized to the mTask share.
- `iAlarmSds` (line 7) counts the alarms, and only has a copy on the server. The device accesses this share directly to set the alarm (lines 36 and 46).
- `iAirQualitySds` (line 8) holds the co2 value, and only has a copy on the client. The server accesses this share directly to display the value (line 21).
- `mCurTempSds` and `mCurHumSds` (lines 9 and 10) are only accessed on the device to communicate the current temperature and humidity to the co2 sensor task.

The implementation is available in Listing 14.

```
1 start :: TCPSettings → Task ()
2 start deviceInfo =
3   withDevice deviceInfo λdevice →
4   withDoublyShared device ItoM 20.0 λiOptTempSds mOptTempSds →
5   withDoublyShared device ItoM 1200 λiMaxCO2Sds mMaxCO2Sds →
6
7   withShared 0 λiAlarmSds →
```

```
  withMTaskShared device 0   λmAirqualitySds →
  withMTaskShared device 0.0 λmCurTempSds →
  withMTaskShared device 0.0 λmCurHumSds →

  interface device iOptTempSds mAirqualitySds iMaxCO2Sds iAlarmSds -||
  liftmTask (tempAndHum mCurTempSds mCurHumSds mOptTempSds iAlarmSds) device -||
  liftmTask (co2monitor mCurTempSds mCurHumSds mAirqualitySds mMaxCO2Sds iAlarmSds)
      device

interface ::  MTDevice (ISDS Real) (MSDS Int) (ISDS Int ) (ISDS Int) → Task ()
interface device iOptTempSds mAirqualitySds iMaxCO2Sds iAlarmSds =
  (updateSharedInformation [] iOptTempSds ≪@ Label "Optimal temperature (C)") ||-
  (updateSharedInformation [] iMaxCO2Sds ≪@ Label "Maximum co2 (ppm)") ||-
  mWatch device mAirqualitySds >&> λiAirQualitySds →
  (viewSharedInformation [] iAirQualitySds ≪@ Label "Current air quality") ||-
  (viewSharedInformation [] iAlarmSds ≪@ Label "Alarm level") @! ()

tempAndHum :: (MSDS Real) (MSDS Real) (MSDS Real) (ISDS Int) → MTaskMain ()
tempAndHum mCurTempSds mCurHumSds mOptTempSds iAlarmSds =
  dht (DHT_DHT (DigitalPin D2) DHT22) λdht → main (

  rpeatEvery (BeforeSec $ lit 30) (
    temperature dht ≫~. λtemp →
    humidity dht ≫~. λhum →
    setSds mCurTempSds temp ≫|.
    setSds mCurHumSds hum ≫|.
    getSds mOptTempSds ≫~. λoptTemp →
    Iff (mAbs (optTemp -. temp) >. lit 2.0)
        (iUpd iAlarmSds (λ_ i → i+1) (lit ()))))

co2monitor :: (MSDS Real) (MSDS Real) (MSDS Int) (MSDS Int) (ISDS Int)→MTaskMain()
co2monitor mCurTempSds mCurHumSds mAirqualitySds mMaxCO2Sds iAlarmSds =
  airqualitySensor AQS_SGP30 λsensor → main (

  rpeatEvery (BeforeSec $ lit 30) ( // Check CO2 level every 30 seconds
    co2 sensor ≫~. λco2level →
    setSds mAirqualitySds co2level ≫|.
    getSds mMaxCO2Sds ≫~. λmaxCO2 →
    Iff (co2level >. maxCO2) (iUpd iAlarmSds (λ_ i → i+1) (lit ()))
  ) .||.
  rpeatEvery (BeforeSec $ lit 600) ( // Recalibrate sensor every 10 minutes
    getSds mCurTempSds ≫~. λtemp →
    getSds mCurHumSds ≫~. λhum →
    setEnvironmentalData sensor temp hum
  ))
```

Listing 14. The source code for the air quality monitoring system.

In this example we see several utilisations of shared data sources that were not possible in the old implementation. There are four major usages of these new shared data sources that were not possible in the old situation:

In the new implementation. . .	**In the old implementation . . .**
The shared data sources for the optimal temperature and max CO2 values (lines 4 and 5) are created, and synchronized only from the server to the client.	The old implementation only has bi-directionally linked shared data sources.
The shared data sources for current temperature and current humidity (lines 9 and 10) only exist on the device.	Each mTask task needs to have its own shared data source, both connected to the same shared data source on the server. Everything is synchronized automatically, and all communication has to go via the server.
The shared data source for air quality (line 8) is directly accessible from the server. If we only occasionally needed the value for something, we only occasionally have to retrieve the value.	The server has its own shared data source that is linked, always keeping it up to date. Communication would take place whenever the client has a new value.
The shared data source counting the alarms (line 7) is safely updated from both the `tempAndHum` task (line 25) and the `airQuality` task (line 38) by directly performing the +1 operation on the server.	Both the `tempAndHum` task (line 25) and the `airQuality` task (line 38) have their own copy. The +1 operation is performed on the copy, and afterwards synchronized to the rest of the linked shared data sources, leading to race conditions.

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 TCP or UDP connections directly [20]. On top of this, high-level communication protocols like HTTP or web sockets can be used. Alternatively, a message broker like MQTT or AMQP can be used. These options are explored and compared in a paper by Naik [12]. The mTask system supports both TCP and MQTT for communication. The programmer, however, never needs to send TCP/MQTT messages directly. This communication is all fully implied by starting/ending tasks, or writing to shared data sources cross platform.

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 to 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. [16] 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 state, 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 [5]. A (remote) observable in Potato is somewhat similar to a SDS in task-oriented programming. The main difference is that an observable produces a stream of values, while a SDS only has a single current value that changes over time.

Our implementation of `loweriTask` uses tagged functions which are then remotely triggered to be executed. This is similar to remote function calls in other programming languages or frameworks. Because of our tierless setup [11], the mTask programmer does not need to define these tags and/or interfaces themselves. Instead, these tasks are generated from the single source file, and automatically inserted into the compiled mTask program.

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.

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 updates or reads 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 shared data source 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 shown in this paper is available online [7].

9 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, all types in mTask are required to be of fixed size, including values for shared data sources. 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 same 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.

Bibliography

[1] Joe Armstrong, editor. *Concurrent Programming in ERLANG*. Prentice Hall, London ; New York, 2nd ed edition, 1996. ISBN 978-0-13-508301-7.

[2] Haye Böhm. Asynchronous Actions in a Synchronous World. Master's thesis, Radboud University, Nijmegen, Netherlands, January 2019.

[3] Christophe De Troyer, Jens Nicolay, and Wolfgang De Meuter. Building IoT Systems Using Distributed First-Class Reactive Programming. In *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 185–192, Nicosia, December 2018. IEEE. ISBN 978-1-5386-7899-2. https://doi.org/10.1109/CloudCom2018.2018.00045.

[4] László Domoszlai, Bas Lijnse, and Rinus Plasmeijer. Parametric lenses: Change notification for bidirectional lenses. In *Proceedings of the 26nd 2014 International Symposium on Implementation and Application of Functional Languages*, pages 1–11, Boston MA USA, October 2014. ACM. ISBN 978-1-4503-3284-2. https://doi.org/10.1145/2746325.2746333.

[5] Paul Hudak. Functional reactive programming. In S. Doaitse Swierstra, editor, *Programming Languages and Systems*, pages 1–1, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. ISBN 978-3-540-49099-9.

[6] Christopher Ireland, David Bowers, Michael Newton, and Kevin Waugh. A Classification of Object-Relational Impedance Mismatch. In *First International Conference on Advances in Databases, Knowledge, and Data Applications*, pages 36–43, Cancun, Mexico, 2009. IEEE. ISBN 978-0-7695-3550-0. https://doi.org/10.1109/DBKDA.2009.11.

[7] Niek Janssen, Mart Lubbers, and Pieter Koopman. Source code for paper Distributed Data in Task-Oriented Programming on edge devices. 2024. https://doi.org/10.5281/zenodo.14236133.

[8] Pieter Koopman, Mart Lubbers, and Rinus Plasmeijer. A Task-Based DSL for Microcomputers. In *Proceedings of the Real World Domain Specific Languages Workshop 2018*, pages 1–11, Vienna Austria, February 2018. ACM. ISBN 978-1-4503-6355-6. https://doi.org/10.1145/3183895.3183902.

[9] Mart Lubbers. *Orchestrating the Internet of Things with Task-Oriented Programming*. Radboud University Press, 1 edition, October 2023. ISBN 978-94-93296-11-4. https://doi.org/10.54195/9789493296114.

[10] Mart Lubbers and Peter Achten. Clean for haskell programmers, 2024. URL https://arxiv.org/abs/2411.00037.

[11] Mart Lubbers, Pieter Koopman, Adrian Ramsingh, Jeremy Singer, and Phil Trinder. Could Tierless Languages Reduce IoT Development Grief? *ACM Transactions on Internet of Things*, 4(1):1–35, February 2023. ISSN 2691-1914, 2577-6207. https://doi.org/10.1145/3572901.

[12] Nitin Naik. Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP. In *2017 IEEE International Systems Engineering Symposium (ISSE)*, pages 1–7, Vienna, Austria, October 2017. IEEE. ISBN 978-1-5386-3403-5. https://doi.org/10.1109/SysEng.2017.8088251.

[13] Robert H. B. Netzer and Barton P. Miller. What are race conditions?: Some issues and formalizations. *ACM Letters on Programming Languages and Systems*, 1(1):74–88, March 1992. ISSN 1057-4514, 1557-7384. https://doi.org/10.1145/130616.130623.

[14] Arjan Oortgiese, John Van Groningen, Peter Achten, and Rinus Plasmeijer. A Distributed Dynamic Architecture for Task Oriented Programming. In *Proceedings of the 29th Symposium on the Implementation and Application of Functional Programming Languages*, pages 1–12, Bristol United Kingdom, August 2017. ACM. ISBN 978-1-4503-6343-3. https://doi.org/10.1145/3205368.3205375.

[15] F. Pfenning and C. Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pages 199–208, Atlanta Georgia USA, June 1988. ACM. ISBN 978-0-89791-269-3. https://doi.org/10.1145/53990.54010.

[16] Laure Philips, Coen De Roover, Tom Van Cutsem, and Wolfgang De Meuter. 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*, pages 69–81, Portland Oregon USA, October 2014. ACM. ISBN 978-1-4503-3210-1. https://doi.org/10.1145/2661136.2661146.

[17] Rinus Plasmeijer and Marko Van Eekelen. Keep it clean: A unique approach to functional programming. *ACM SIGPLAN Notices*, 34(6):23–31, June 1999. ISSN 0362-1340, 1558-1160. https://doi.org/10.1145/606666.606670.

[18] Rinus Plasmeijer, Peter Achten, and Pieter Koopman. iTasks: Executable specifications of interactive work flow systems for the web. *ACM SIGPLAN Notices*, 42(9):141–152, October 2007. ISSN 0362-1340, 1558-1160. https://doi.org/10.1145/1291220.1291174.

[19] Rinus Plasmeijer, Bas Lijnse, Steffen Michels, Peter Achten, and Pieter Koopman. Task-Oriented Programming in a Pure Functional Language. In *Proceedings of the 14th Symposium on Principles and Practice of Declarative Programming*, PPDP '12, pages 195–206, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1522-7. https://doi.org/10.1145/2370776.2370801.

[20] W. Richard Stevens and Kevin W. Fall. *TCP/IP Illustrated. Volume 1, The Protocols*. Addison-Wesley, Boston, MA, 2nd ed edition, 1994. ISBN 978-0-13-280820-0.