Master Thesis Computer Science

Reacting to concurrent changes in shared data sources

Radboud Universiteit Nijmegen

Author:
Bob van der Linden

Supervisor:
M.J. Plasmeijer

Assisted by:
Steffen Michels

December 1, 2012
Thesisnumber: 656

Contents

Chapter 1

Introduction

Large software systems often use huge amounts of data. This data is retrieved from and stored to many different sources. Frequently, these sources are also used by many different systems. Many of these systems are web applications. Web applications used to be strictly request-response mechanisms, where the user requests a page and the server responds with the page. Nowadays web applications tend to become more 'real-time'. A page is requested, the server responds with the page and afterwards the page is being kept up-to-date with the latest data the server has.

This approach is great for interactivity, but it can often be hard to implement. The problem of communication between server and browser has been solved by new web standards. The WebSocket API [1] allows for two way communication next to the traditional request-response mechanism. The EventSource API [2] allows a server to send events to a browser. Both of these APIs are implemented in most modern browsers. Now, next to the browsers being able to send data to the server at any time without these APIs, it is possible for servers to send data to the browsers at any time too, using one of these standards. This makes it possible to efficiently keep a webpage up-to-date with a server.

This is quite different between the server and its data. Whereas we have now a standard for communicating with browsers, there are many different ways to store and retrieve data. Some data is stored as text files to be easily accessible, some data is in memory to be accessed efficiently and some is on a central database to be accessible from different locations. The different ways of storage all use their own interface to store and retrieve the data. Data which is accessed concurrently also use different methods to synchronize, such

as locks or transactions. Lastly, for a real-time system to work, there need to be ways for the server to know when data has changed. The technique to do this also varies greatly between the different types of data storages. For files the operating system needs to tell your program when a file has changed. For databases it depends greatly on the type of database that is used. For memory one has to implement such functionality for themselves.

Existing real-time systems have shown their ability to handle real-time changes made by users quite efficiently. One example is Etherpad [3]. This web-application is an online text-editor that lets multiple users edit the same text-document in real-time. In this application the system interacts with a database that contains the text-documents. All changes happen through the same application, which makes it possible for the application to know of all changes that are being performed. In this thesis we focus on ways to know of changes of multiple data-sources, which could be in different locations and are changed by different systems.

The concept of shared data sources [4] tries to solve the problem of storing and retrieving data from different sources in a reliable way from within a functional language, so that operations on the different data keep all data consistent all the time. It provides a concrete type of the actual data so that it is easy to swap out one data source for another, while still using the same interface.

This still leaves open the problem of being notified of changed data. In this thesis we extend the concept of shared data sources with a way to be kept notified of changes. We show the different methods of doing so and conclude with a method that should work both reliably and efficiently.

iTasks [5] is a system that relies on shared data sources. It is a web application to manage tasks which can be handled by different users and systems. The system has a concurrent nature. Different tasks can be handled at any time where the tasks rely on different internal and external data sources. iTasks makes use of shared data sources to access data that is needed from different tasks. However, the concurrent nature of iTasks asks for real-time up-to-date data that the users can interact with.

Since iTasks uses shared data sources to access its data, we want to have some way for iTasks to know when data from a shared data source has changed. This requires an extension to the existing concept of shared data sources which provides a way of notifying user-code when a shared data source has changed.

In this thesis we look for different ways to define and ways to implement

such an extension. We show what requirements we have for the extension to be usable in iTasks, but the extension is still separate from iTasks. This means shared data sources and the extension should both be usable in other applications.

We also show how we used the change notifications in iTasks to explore and show the problems that arise when actually using the notifications in a task-oriented system. We use a simulation of iTask to simplify our problem so that tests are more easily made and problems become clear more quickly. The actual implementation in iTasks itself will not be part of this thesis.

In chapter 3 we show what options we have found in regard to ways to notice and notify changes. In chapter 4 we show how we used the notifications in our iTasks simulation. We show how the simulation is implemented in chapter 5. In chapter 6 we show what work there has been done in the area of concurrent and real-time systems in regard to change notifications. We conclude in chapter 7 with a summary of our findings and recommendations. First however, we will give some background on iTasks and shared data sources in chapter 2.

Chapter 2

Background

In this thesis we refer to a number of existing systems that we make use of. In this section we give a short introduction to what these systems are and how they are used.

2.1 iTasks

iTasks is a system that relies on shared data sources. It is a task oriented programming framework which allows users to generate complete workflow applications using a workflow specification that is embedded in the general purpose functional language Clean [6]. This enables the specification to inherit from the state-of-the-art programming language concepts that the functional language provides.

When workflows are specified and the application is generated, users can use the application via a web-interface, as shown in Figure 2.1. In the web interface users get a list of tasks which they can perform, like filling out a form.

Because workflows are defined within the Clean programming language, it uses different functions that can be combined. These functions include functionality to enter, update and show information. What kind of data can be entered, updated or shown is defined by the type-system. This means lists, tuples, records and algebraic data types can be used to describe what kind of data the user can interact with. iTask generates a graphical user-interface that fits the data-type.

Since the type-system of Clean can infer the types of its expressions for

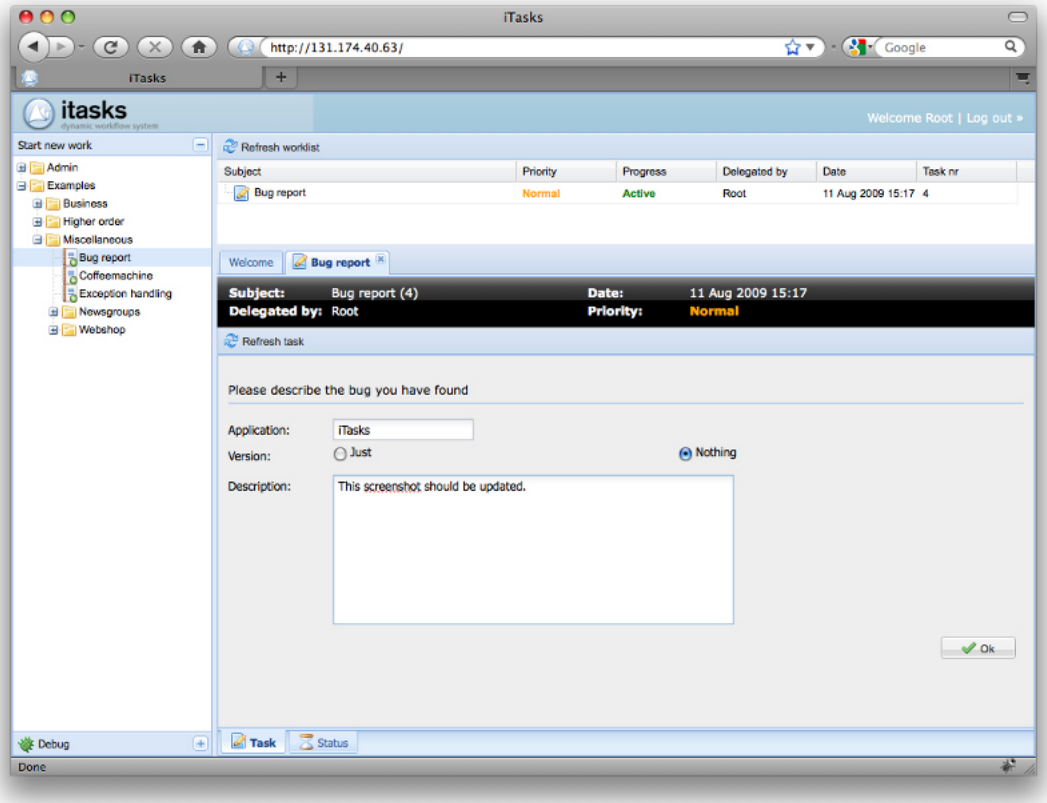

Figure 2.1: The web-interface of iTasks

most cases, it is often not necessary to explicitly annotate the workflows with type information. This can be seen in the later examples.

First we show a very simple workflow that only shows data to the user. We define the workflow using the built-in function `showInformation`, which needs a description, a list of options and the data to be shown. At runtime, the user sees the description, followed by a representation of the data. In this example the description and the data shown are both fixed `String`s:

```
welcome = showInformation "Welcome" [] "This is some data"
```

This example only shows how to report data to the user, but since the data is constant the example is somewhat contrived. A more interesting example would show data that depends on user input.

In the following example we use the function `enterInformation` to ask the

user to fill in their name. This function requires a description and a list of options. It results in data that the user has entered. Thereafter we greet the user, with their entered name, using `showInformation`. To combine these two functions we use the combinator `>>=`, which makes the two tasks run in sequence. It also allows us to retrieve the value that `enterInformation` results in and use that value as an argument of `showInformation`.

```
welcome =
                enterInformation "Please enter your name" []
    >>= λname → showInformation "Welcome" [] ("Hello " +++ name)
```

In this example we can see the use of the combinator `>>=` as well as a lambda function. In Clean a lambda function is declared as λ, followed by argument names, followed by $\rightarrow$, followed by the body of the function. Using `>>=` combined with the lambda function, we name the result of `enterInformation`: `name`. Thereafter we use `name` to make the `String` that is shown to the user by concatenating `name` to `"Hello "`. Because we do such a concatenation, the type-system knows that `name` should be of the type `String`. The type system also infers that therefore the argument `name` is of type `String` and therefore the result of `enterInformation` must, in this case, also be a `String`. This makes iTask generate a user-interface to enter a `String` when `enterInformation` is executed.

Tasks can also rely on data that is shared with other tasks, which are being performed by other users. It does this through shared data sources. To illustrate this, we extend the previous example by adding the name to a shared list of welcomed users:

```
welcomedUsers = sharedStore "Welcomed Users" []

welcome =
                enterInformation "Please enter your name" []
    >>= λname → update (λusers → users ++ [name]) welcomedUsers
    >>|         showInformation "Welcome" [] ("Hello " +++ name)
```

Here `welcomedUsers` is the shared data where we store and retrieve our list of users. `update` is used to atomically add the name to the list of users. After updating we greet the user again like before.

With the example above `welcomedUsers` is only updated and never shown to any user. We can use the following task to show the list of users that we have welcomed:

```
showWelcomedUsers =
        get welcomedUsers
```

```
>>= showInformation "Welcomed users:" []
```

In these examples data is used from a shared data source, whereas previous examples only used local data. Local data is data that is only accessed and changed from the same task. When using shared data sources the data cannot only come from different tasks and users of iTasks itself, but also from other (external) systems through the same interface. The data could be stored in a file or in a database. Other processes can use the that same file or database to access and alter the list of users.

Because some data is shared between tasks, the state of tasks can change when the data, on which the task relies, is changed. This is a very important aspect since it allows users to cooperate and interact, each through their own task, but on the same data. In the above example the data is first retrieved from 'welcomedUsers' and after that the retrieved data is shown to the user.

A more interesting example is an alternative of 'showWelcomedUsers' which retrieves and shows the data from 'welcomedUsers' every time the user visits the task:

```
showWelcomedUsers = showSharedInformation "Welcomed users:" [] welcomedUsers
```

This task retrieves, instead of retrieving the data once, the data every time the information is needed to be shown. The user then always receives up-to-date data from the shared data.

At the time of writing iTasks is not showing changes of its data to the user in an efficient real-time manner yet. When a piece of data (like the list of welcomed users) is changed while a user is watching that data (directly through `showInformation` or via some other method that depends on the data), the user only sees these changes when the user had enabled an option on the web-interface to 'poll' the website for changes. iTasks then uses a short time interval to check whether the server still has changes and refreshes its user-interface once there are any changes pending. This is not very efficient, since the server needs to retrieve all data that is visible to the user. Using this data it needs find out whether it has changed comparing to the previous data it has sent. This can be a bottleneck when a great number of users are using the same iTasks system. It becomes more of a problem when external data sources are used, which all need to be checked for changes every time-interval for each user.

This is where the extension to the concept of shared data sources comes into play. It should enable iTasks to support efficient real-time updates of its tasks and forward those changes in a presentable manner to the user.

2.2 Shared data sources

In the previous sections we touched on the concept of shared data sources and showed how we use it in iTasks. In this section we look more closely how shared data sources can be used. Additionally we show what different characteristics the concept of shared data sources has.

The concept of shared data sources is a solution, implemented in Clean, that allows a uniform and reliable way of accessing data. It has a number of characteristics that makes them reliable to work with under many different conditions.

- A shared data source has its actual storing and retrieving abstracted. This makes it possible to work with data the same way, independent of the physical locations where the data could be stored (such as memory, disk or network).
- A shared data source is always accessed in a type-safe way. The type of the data is predefined and the type-system of Clean can check whether this condition holds.
- The type that is used for writing and reading can differ. This allows for access control so that only parts of the data can be written through the data source. It also allows for hiding information when exposing less for the reading-type compared to the writing-type.
- The data sources can be accessed concurrently while keeping them consistent. We accomplish this behavior using locks, but also using atomic transactions.
- Each data source has a version number. The version can be compared to earlier read versions to determine whether the data has changed without comparing the whole data structure.
- Two data sources can be composed into a new data source. This allows data from different sources to be combined into a data source that can be used just like any other.

With the right implementation of a shared data source, it is possible to reliably read and write data between applications or different machines the same way as we would with data from memory. It is also possible to compose

data from different sources. This is all possible without the fear of accessing the data concurrently and without deadlocks.

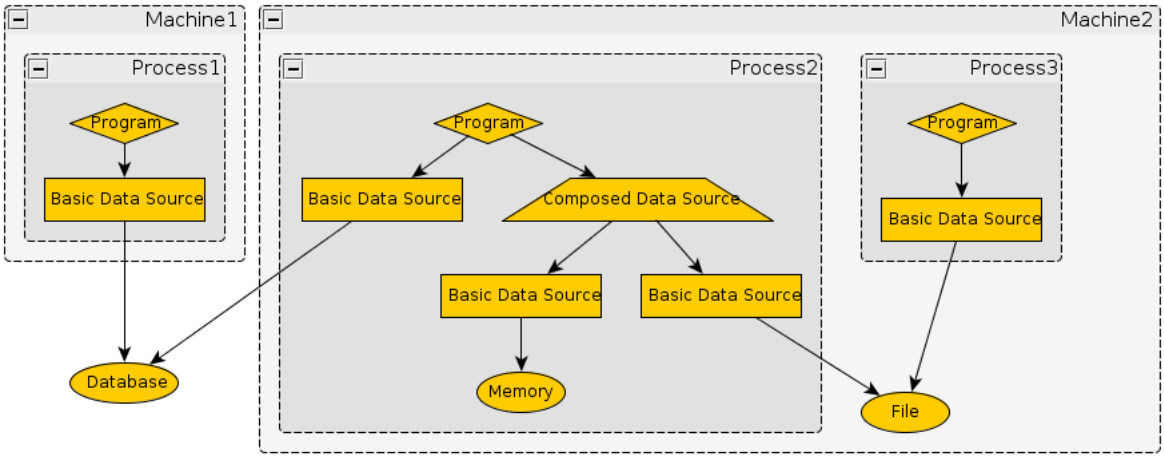

Figure 2.2: Overview of shared data sources being used on different machines

Figure 2.2 gives an overview of the shared data source-system. A program, which runs inside a process, can access data from many different sources. It does this through shared data sources. The data can originate from its own process (memory), its own machine (files) or other machines over the network (databases). These are some examples of such sources, however it is possible to implement more using the same interface that the shared data source-system provides.

By accessing the data through shared data sources it is possible to access multiple sources of data safely. This means that the data can be read and written without concurrency problems or deadlocks. When data is being accessed through a shared data source and the same data is being written through another shared data source, the accessing process waits until the writing process has finished with its operations.

This also applies when multiple shared data sources are composed into one data source. These composed data sources behave just like normal shared data sources. In figure 2.2 a composed data source is shown which combines data from a file and data from memory. Just like normal shared data sources, it is possible to safely access these without concurrency problems. The same sort of mechanism applies where the accessing shared data source waits when another is writing to one or both of the data sources.

Figure 2.2 also shows that a single program can use multiple data sources, such as some basic data sources and some composed data sources.

Basic data sources are the core mechanism that retrieves and writes data from and to physical locations. The basic data source is an abstraction that can have any implementation. Therefore the physical location of the data is determined by its implementation. The implementations that are already planned are memory, file and database. The abstraction allows for reading, writing, locking, versioning and notifying for atomic operations. It has a type for reading (reading-type), a type for writing (writing-type) and a type how the data is stored (storing-type). The storing-type is only used internally within the specific implementation. When writing a value, the value is converted from the writing-type to the storing-type and send/written to the physical location. When reading a value, a value is retrieved from the physical location as the storing-type and is converted from the storing-type to the reading-type.

Composed data sources are a combination of two arbitrary data sources with projections on how the two data sources are combined, one for reading, one for writing. Multiple basic data sources can be combined into one composed data source by using multiple composed data sources, as shown in figure 2.3.

Note that the 'users' in this example are all in the same process as they have direct references to the composed and basic data sources. When other processes access the same physical data through the concept of shared data sources, those processes must also create their own basic and composed sources. Processes that do not use the concept of shared data sources can still access the physical data locations directly. Even though this is possible, for some of the data sources the processes must follow the same protocol as the shared data sources do so that features like concurrent access and versioning will still be intact.

It is also possible that basic sources are used in different parts of the program, either via the use of composed data sources or by using a basic data source directly. An example of such a situation is shown in figure 2.4.

To reliably read from and write to multiple data sources, operations on these data sources are performed in a single atomic operation. The function `readWrite`, which is provided with the implementation of shared data sources, is used to accomplish this:

```
:: RWRes w a state = YieldResult a | Write w a | Redo state
```

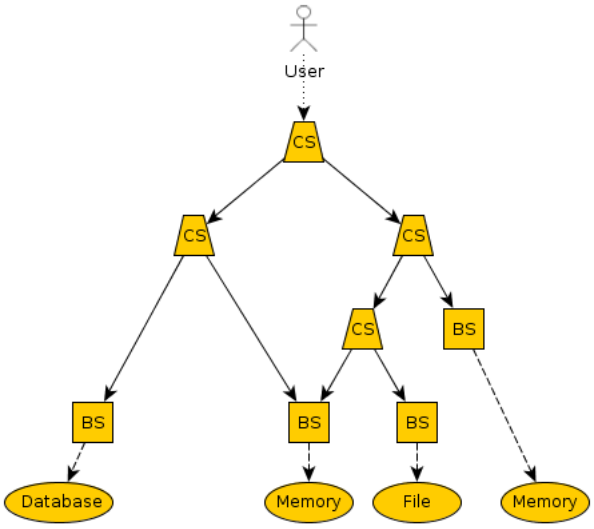

Figure 2.3: Multiple basic data sources (the leafs) are combined to a single combined data source, which is used by the user.

```
readWrite :: !(r Version state → (RWRes w a state)) (RWShared r w *World) state
                *World → (a, *World)
```

In this definition we use `RWShared` to describe the type of a shared data source. Here `r` is the type of data that can be read and `w` is the type of the data that can be written. `state` is the type of data for a value that is held across the transaction of the read or write operation. Finally, `World` is the global state in Clean.

A `RWShared` contains all functionality of a shared data source, specifically for the type of data source. This means that when a file-based shared data source is made, the result is a `RWShared` that contains functionality to read, write, version and lock files.

`readWrite` uses `RWShared` as its second argument to have all of the needed functionality for the specific type of shared data source. The first argument of the `readWrite` function is the atomic function. This function defines what action `readWrite` should take once the atomic function is finished executing. The evaluation of the atomic function and the action that `readWrite` takes all happen without other processes being able to change the data.

When executing `readWrite`, it first locks all basic data sources, then reads

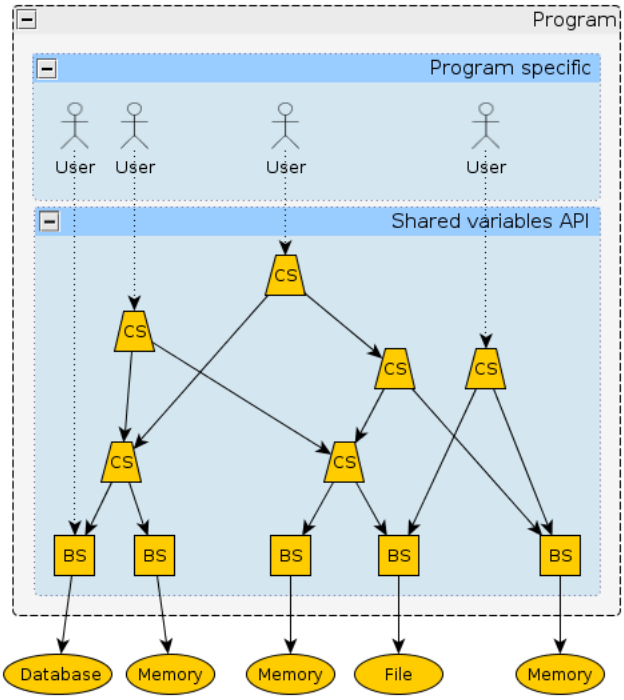

Figure 2.4: Multiple users can access different compositions of basic data sources, or access them directly.

the data and versions of the data from the physical locations. Thereafter it executes the projections of the basic sources and composed sources to get a value of the right type. The resulting value and version are passed into the atomic function, which is then executed. The atomic function is user-defined and results in `YieldResult a`, `Write w a` or `Redo state`. `readWrite` acts on that result: `YieldResult a` unlocks the data and make `readWrite` result in the value that is passed to `YieldResult`. `Write w a` first writes the passed value (`w`) to the data source and unlocks right after that. The value `a` is passed back as the result of `readWrite`, just like `YieldResult` does. `Redo state` makes `readWrite` unlock and waits until the data has changed and reruns the atomic function when that happens. The state that is passed to `Redo` will be passed to the atomic function the next time it is executed.

This behaviour is visualized as an activity diagram in Figure 2.5.

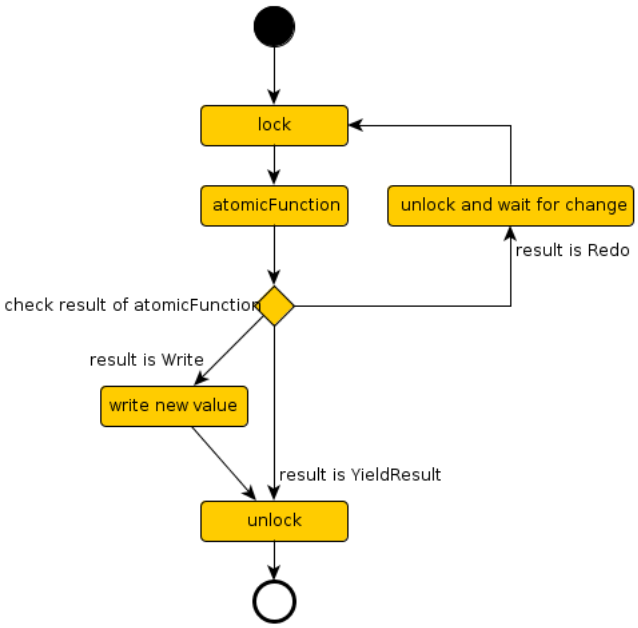

Figure 2.5: Visualized mechanism of readWrite

Chapter 3

Notifying on changes of shared data sources

In this chapter we explore the different possibilities of knowing when a shared data source has changed. We also explore the possibilities of delegating the notification to user-code.

For our purposes we use shared data sources mainly for iTasks. However, since we want our system to be general enough to be used in other systems, we refer to the application using shared data sources as 'user-code'.

We start by describing the properties we want our system to have. This is followed with solutions which we have found during our experiments. For each solution we describe how it works and summarize which of the described properties hold for the particular solution. We finally conclude by summarizing the main points of interest with our system of choice.

3.1 Requirements of notifications

Our system will be an extension to the existing concept of shared data sources. In this section we describe a number of characteristics which we think should apply to our extended system. In the later sections we refer back to these characteristics so we can determine what solution has most of our defined properties.

First we want our extended system to still be compatible with the properties as described for shared data sources in section 2.2. To summarize these properties:

- Data sources can be read
- Data sources can be written
- Basic data sources can read and write independent of the physical location of the data
- Data sources can be composed into new data sources through projections
- Data sources have a version, which is increased each time data is written
- Data sources can be accessed concurrently
- Data sources always access their underlying data consistently: changes to multiple pieces of data can be performed atomically

For our extension to work, we need some way for the user-code to be notified when the data in a data source has changed. A solution without this property is not really a solution, so all solutions in the later sections have this property.

Next we need some way for the user-code to know what shared data source has changed when the user-code is notified. Without knowing what shared data source has changed, it is hard for user-code to efficiently know what exactly it should do. In the case of iTasks it would not know what tasks it should evaluate.

Keeping track of *all* data-sources that exist in the system is not very beneficial for the user. The user is often only interested in changes of specific data sources. Letting the user know about all changes in the system would only cause the user to do extra work by filtering the changes, which only causes overhead. Therefore we need some way for the user-code to specify what data sources it wants to be notified of.

The notifications that are passed to user-code must be handled within a certain timespan. This timespan will depend on the type of application that the data is used for. Often these types of applications require that the notification to the user must be seemingly instant, which means within a second [7]. This means that our system needs to be able to pass notifications faster than that. This is independent of the retrieval of the specific data sources, since some data is easier to retrieve than others (memory is faster

than a off-site database). The type of data source should be chosen wisely by the user to meet the required timespan.

We are only interested in the latest state of the data and not the individual changes. Therefore the notifications may miss some of the changes. However, when there is a change, the user-code must be notified within the timespan described previously. This means when there are multiple changes to a data source shortly after each other, the user only needs to be notified about the last of those changes. With this mechanism, the user-code cannot keep track of all states of the data, but is still always notified about the latest state of the data.

We also do not mind multiple notifications for a single change. The goal is to have the user know about the latest state of the data. When multiple notifications are send to the user without any actual change, the user can still find out that there is indeed no change by comparing the current version of the data source with the previously read version.

Lastly we want our system to run smoothly along with other code. Therefore the notification-system may not induce too much overhead. Not only must the notification system not require too many of the system resources, but the user-code that handles the notification must also not be forced to use many resources. For instance the notification system may not notify the user too often about a single change. This property depends greatly on the exact resource requirements of the system, but we elaborate for each of the solutions whether this property is satisfied.

To summarize the additional requirements we want our system to have:

- User-code must be notified of changes of data
- User-code must know which data source has changed
- User-code must be able to specify what data-source to track
- Execution of user-code must not be blocked
- Notifications must be received within a reasonable timespan after a change
- The system must be conservative with system resources

In the next sections we describe the possible solutions we have found and which properties are satisfied or not.

3.2 Polling

We mentioned earlier that iTasks makes use of a technique called 'polling' between the server and the client. This technique is widely used in web-oriented applications where request-response queries are common.

This technique works by getting data from its source at regular intervals. Changes are found by keeping a copy of the data from a previous 'poll' and comparing this with the newly retrieved data.

Polling is widely used because most webservers only support request-response queries. Data is retrieved by requesting it from the webserver. The webserver responds with the requested data. Doing this at regular intervals can keep the data on clients up-to-date.

For shared data sources this technique can also be used. Data can be retrieved from the data source by reading its data through the interface of the shared data source. Doing this at regular intervals gives us the ability to notice changes by comparing the data. We can optimize this by not retrieving the data itself, but retrieving the version of a data source. One of the properties of shared data sources describes that when the data has changed, the version is also changed.

Polling can however have a number of downsides. First, the time until the next poll must be predetermined. It is often not clear how long before the next change will occur. Choosing this wrongly can affect performance or user-interactivity badly.

Secondly, retrieving the data (or version) of shared data sources can be a demanding operation when used on certain sources (like disk). It becomes a problem when this is performed frequently. Therefore the time between polls needs to be balanced so it does not cost too much performance, but still gives a reasonable rate of noticing changes. Balancing these properties greatly depends on the use of the data.

Another problem is the performance impact of polling multiple data sources. When a user wants to be notified of changes across the system, many data sources need to be polled. This makes polling even more heavy on the system resources, since every poll requires getting data from multiple data sources. Having their data on different systems requires the application to query all of these systems every poll.

With these downsides we think polling is not a suitable technique for most of our uses.

3.3 Callback on writing

In iTasks we have a number of possible operations that we can use on shared data sources. In the earlier examples we showed some of these operations: 'get', 'update' and 'showSharedInformation'. All queries or alterations that tasks perform on data sources go through similar operations like these.

This property makes it plausible to add functionality to these operations for the user to notice when a data source is being updated. We could for example add functionality to the update operation that notifies the user that the data source has changed after the actual data was written.

However this approach only works for some types of data sources that are only altered by our program. We show why this is the case using an example.

Suppose we access a file through a data source. We do this from two different programs, program A and program B. Program A accesses its data sources using our proposed altered operations which notifies the user of changes, however program B does not. When program A writes to the files, it is possible to notify the code of program A of that change. However the problem becomes more complex when program B is needed to be notified without program A knowing about program B. Next to that, program B can write to that same file. In this case both program A and program B are not notified about the change of their data source.

The type of data source where this method does work is memory that is only accessed by one program, since the program is the only one reading and writing its own memory. Therefore this problem does not apply to these types of data sources.

Enforcing all systems that are using a certain data-source to implement the same notification-system is a solution. However, since there often already are programs in place to handle the data it becomes hard to extend or replace these with such a notification system.

Since we attempt to solve the problem of notifications of changes in shared data sources independent of the type of data source, this solution is not sufficient on its own.

3.4 Waiting for changes

In this section we explore the possibilities of waiting for data sources that will change. We will show the different methods we have found with which

we can wait for changes. We also show what to do when we have waited for a change. We conclude each waiting-method with its positive and negative aspects.

3.4.1 Using atomic properties for waiting

In section 2.2 we spoke about the mechanisms of the core function in the Clean implementation of shared data sources, named 'readWrite'. We showed that readWrite performs the read, write and redo operations depending on the atomic function being used as shown previously in Figure 2.5.

As a first solution we attempt to implement change-notifications of shared data sources using the existing implementation of readWrite and in particular using the mechanism that is provided by `Redo`. We call a user-defined function after `readWrite` when the atomic function notices a change. `readWrite` and the atomic function can achieve this using the `Redo` operation.

To better understand what `Redo` does, we show an example that implements a message-queue using shared data sources. In this example, the message-queue is a shared data source that contains a list of messages. The only two operations that we can perform on that list are `sendMessage` and `receiveMessage`. `sendMessage` adds an element to the end of the list. `receiveMessage` removes the first element from the list and return that element as the result of the function. When `receiveMessage` finds that there are no elements on the list, it waits until an element is added.

```
// A MessageQueue is a data source that contains a list of messages
:: MessageQueue msg st :== Shared [msg] st

// sendMessage adds a message (msg) to the end of a message-queue (queue).
sendMessage :: msg (MessageQueue msg *World) *World → *World
sendMessage msg queue st
  # (_, world) = readWrite addMessage queue Void st
  = world
where
  addMessage messages version _ = Write (messages ++ [msg]) Void

// receiveMessage removes the first message from the message-queue (queue) if
// there is at least one message in the list.
// If there are no messages in the message-queue, the function waits until
// the list is changed.
```

```
receiveMessage :: (MessageQueue msg *World) *World → (msg, *World)
receiveMessage queue st = readWrite popOrRedo queue Void st
where
  popOrRedo [] _ _ = Redo Void
  popOrRedo [msg:msgs] _ _ = Write msgs msg
```

First we defined the type `MessageQueue` for our queue. We define it as an `Shared` type, which makes it an shared data source that has the same type for its read data as the written data. The data of the queue is a list of `msg` as defined with `[msg]`. In this case `msg` can be of any type. This `MessageQueue` is used in `sendMessage` and `receiveMessage`.

`sendMessage` uses `readWrite` to get the list of messages and add the message that is passed as the argument of `sendMessage` to that list. It puts the new list back to the source using `Write`.

We can see the use of `Redo` in `receiveMessage`. When the list that is stored in the data source is empty (`[]`) the atomic function (`recF`) results in a `Redo`. This causes `readWrite` to wait until the list is changed. When this happens, the atomic function (`recF`) is rerun and executes the second case (`[msg:msgs]`) where the first element of the list is returned and the tail is written back to the data source.

In this example `Redo` is used to wait for a change, which is exactly what we want to do. However, the example relies on the type and data of the shared data sources. In this case it relies on the data source being a list and only use `Redo` when that list is empty.

For our case we want `readWrite` to be waiting for a single change of the data. To do this, we need the atomic function to return `Redo` only once. Unlike `receiveMessage` we cannot check for the data. For this we need to make use of the state that `readWrite` allows us to pass along with a `Redo`.

In the following example we use the state of `readWrite` in combination with `Redo` to have `readWrite` wait for a change of a data source, named `exampleShare`.

```
example :: (RWShared r w *World) *World → *World
example exampleShare world
    # (_, world) = readWrite atomicFunction exampleShare True world
    # world = print "Done" world
    = world
where
    atomicFunction data version state = if state
        (Redo False)
        (YieldResult Void)
```

First `readWrite` is called. Along with the atomic function and the share, an initial state is passed to `readWrite`. In this example the value that we pass as the initial state is `True`. Next `readWrite` calls `atomicFunction` instantly. The state is passed to this function. Since the initial state is `True`, `Redo False` is the result of the first execution of `atomicFunction`. With `Redo False` we not only instruct `readWrite` to rerun `atomicFunction`, but also set the state to `False`. The next run of `atomicFunction` only happens when `exampleShare` is being written to as described in the properties of shared data sources. Presuming some other process does indeed write to `exampleShare`, `atomicFunction` is called a second time by `readWrite`. This time the state is `False`. Therefore `atomicFunction` results in `YieldResult Void`. This instructs `readWrite` to return nothing as the result of `readWrite` and does not cause `readWrite` to rerun `atomicFunction` again. Quickly thereafter `"Done"` is printed. Figure 3.1 visualizes this mechanism.

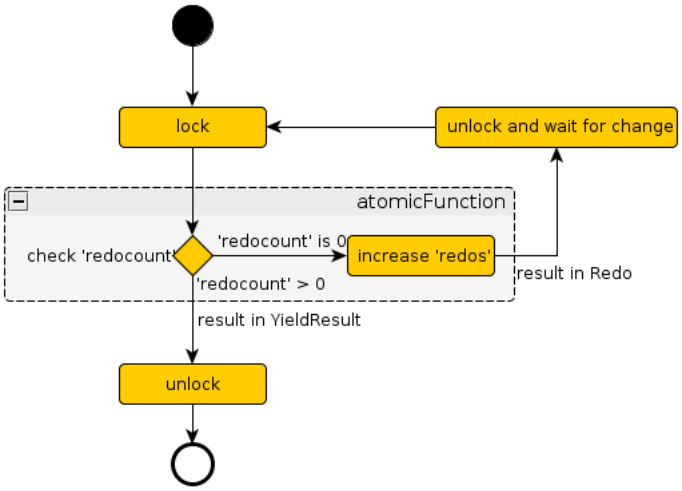

Figure 3.1: Visualized mechanism of a single redo

Now that we have established the core mechanism, we need to have some way for user-code to register for changes of some shared data source. We allow the user-code to register a function (`*World` → `*World`), which gets called when the data source is changed. We can define a new function to be able to register for changes as the following:

```
:: Callback :== *World → *World

subscribeForChange :: Callback (RWShared r w *World) *World → *World
```

We can supply the function two arguments. The first is a function that is called when such a change occurs. We call this function a callback-function. The second argument is the share we want to track changes of. We also want `subscribeForChange` to be non-blocking. This means that we can continue to evaluate right after `subscribeForChange` is called without being *required* to wait for changes. This is in line with the requirement to not block execution of user-code. It allows the user to register for changes without sacrificing the thread on which the user is currently executing code.

We can allow user-code to continue executing while we wait for changes by creating a separate thread on which we use the waiting-mechanism described before. On that thread we wait for changes and afterwards call the callback function. This mechanism is shown in Figure 3.2.

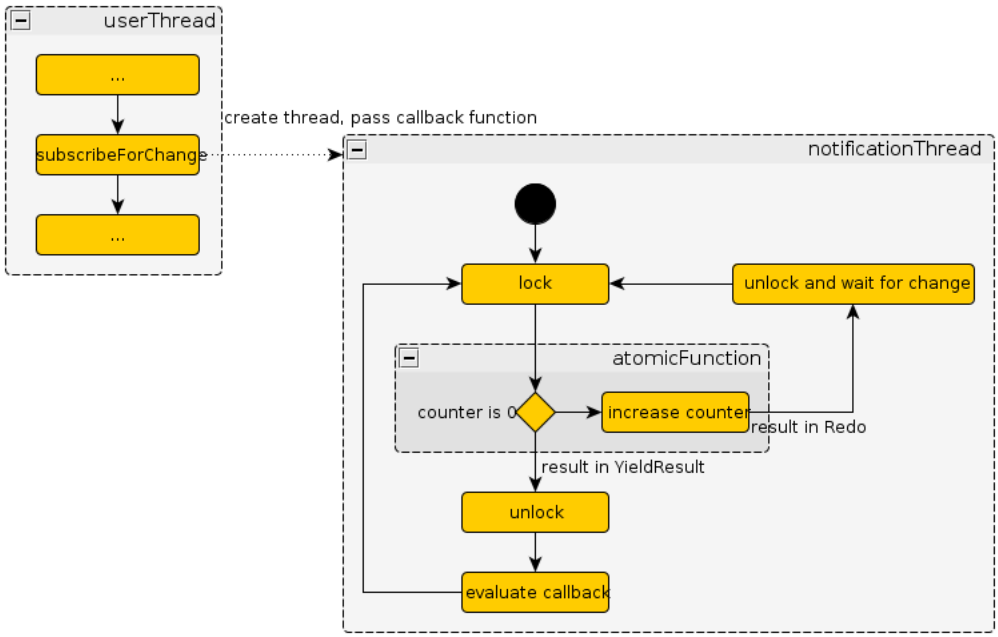

Figure 3.2: Visualized mechanism of subscribeForChange using readWrite and Redo

In Clean new threads can be created using `fork`. Here a function is supplied to `fork` and that function is executed on a new thread. We can implement the desired mechanism shown in 3.2 using the following code:

```
subscribeForChange :: (RWShared r w World) (*World → *World) *World → *World
```

```
subscribeForChange share callback world
    (_, world) = fork notificationThread world
    = world
where
    notificationThread :: *World → *World
    notificationThread world
        # (_, world) = readWrite atomicFunction exampleShare True world
        # world = callback world
        = notificationThread world

    atomicFunction data version state = if state
        (Redo False)
        (YieldResult Void)
```

In this function we use `fork` execute the function `notificationThread` on a new thread. There, we use `readWrite` in combination with `atomicFunction` to wait for a change, using `Redo` once. After `readWrite` has executed, we call the callback function and start over waiting for changes.

`subscribeForChange` can be used in other programs as follows:

```
example :: (RWShared Int Int *World) *World → *World
example exampleShare world
    # world = subscribeForChange exampleShare printOnChange world
    // Do something else in the program.
    # world = writeShare 3 exampleShare
    = world
where
    printOnChange world = print "exampleShare has changed" world
```

This example program prints `"exampleShare has changed"` when the program writes to `exampleShare`. It shows that this method does detect changes, however the described implementation does have a number of issues.

First we have the problem that the function starts waiting for changes later than we would expect. We create a thread where we wait for changes using `readWrite`, but continue without waiting for the `readWrite` function to be actually started. This would cause the user-code to miss changes.

The following example is quite similar to the previous one. The difference is that `writeShare` is evaluated directly after calling `subscribeForChange`. It causes a bigger chance that the change being made by `writeShare` is not triggering `printOnChange` to be executed. `writeShare` can be executed before the separate thread is executing `readWrite`:

```
example exampleShare world
    # world = subscribeForChange exampleShare printOnChange world
    # world = writeShare 3 exampleShare
    = world
where
    printOnChange world = print "exampleShare has changed" world
```

Therefore the example sometimes does print `"exampleShare has changed"` and sometimes it does not. This problem is however solvable using proper locking.

Another problem is that we can miss changes of the data source that happen between calls of `readWrite`. After each change, we call the callback-function and afterwards call `readWrite` again. During the time between the calls to `readWrite` we are not waiting for any changes, thus those changes are missed. Most of these missed changes can be checked by using the version number of the data source that we can retrieve in the callback-function, however there is still a period where we actually miss changes, like the time between the callback and `readWrite`. This problem is a lot harder to solve.

Lastly we have a problem of system resources. If we would want to be notified of multiple shared data sources, we would need to call `subscribeForChange` for each of those shared data sources. For each call a new thread is created. For example if we want to wait for changes of 10 different data sources, which is not uncommon, we would need to create 10 threads.

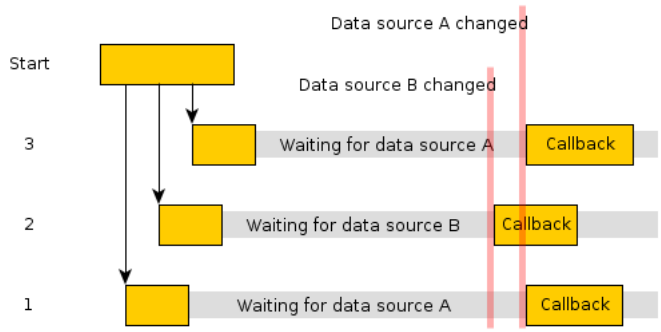

Figure 3.3: Each registered callback gets its own thread

In Figure 3.3 the threads are visualized. The program first calls `subscribeForChange` to register 3 callback functions: 2 callback functions to wait for data source A, 1 callback function to wait for data source B. Thus

the number of threads in this example is equal to the number of callback functions that the user wants to register. Such number of threads costs our system quite some memory. Each of them need a stack, a heap and operating-system specific data, like security context. When those threads are created, they mostly are waiting for changes, so this would be a waste of resources.

These problems all go against our requirements and so this solution is not sufficient enough to be used in iTasks.

To get a better understanding of how we could solve these problems we must look how shared data sources work under the hood and see whether we can exploit that mechanism to get a more reliable and efficient way to implement `subscribeForChange`.

3.4.2 Analyzing the waiting mechanism of shared data sources

At the time of writing this document the implementation of shared data sources is only compatible with the Windows operating system. Therefore we only focus on this implementation.

Under the hood, all shared data sources use Windows event-objects to notify waiting `readWrite` operations. This happens when the atomic function instructs `readWrite` to perform a redo when the shared data source has changed. It accomplishes this with three Windows API calls: `CreateEvent`, `EventSet` [8] and `WaitForSingleObject` [9]. Calling `CreateEvent` results in a `HANDLE` that represents an event-object. Calling `WaitForSingleObject`, with the `HANDLE` as argument, waits until `EventSet` is called on that same `HANDLE`.

When the atomic function returns `Redo`, it first creates a new event-object using `CreateEvent`. Thereafter it registers that event-object with the shared data source and use `WaitForSingleObject` to wait for the event-object to be signaled. Once the shared data source changes, it signals all event-objects registered to the shared data source using `SetEvent`. The waiting `readWrite` operation is signaled and re-evaluates the atomic function.

To wait for changes we can also use `WaitForSingleObject`, just like `readWrite` does when the atomic function results in a `Redo`.

Because waiting for changes works very similar, it also shares the characteristic of blocking while waiting for changes. To work around this problem a thread can be started where we wait using `WaitForSingleObject`. After `WaitForSingleObject` has returned the callback can be evaluated.

This method works reliably, but one outstanding issue is that it needs one thread per shared data source. When we want to be notified of a combined shared data source, we need to create a thread for each basic shared data source the combined shared data source consists of. Since there can be many shared data sources, this is not an acceptable solution. Reducing the number of threads needed can result in a better solution.

3.4.3 Exploiting the waiting mechanism of shared data sources

In this section we show some of the ways we can exploit the use of event-objects in the implementation of the data sources. We can use different Windows API calls that allow us to wait for signals of the event-objects in different ways.

Waiting for multiple events at once

The problem of needing one thread for each event-object to wait on comes from making use of `WaitForSingleObject`. Windows also provides another API which is called `WaitForMultipleObjects` [10]. This function accepts an array of up to 15 `HANDLE`s which can be waited on. It returns when one of those `HANDLE`s is signaled. This allows us to wait for 15 data sources on a single thread.

Until now we tried to handle the threads which were used for waiting. It is quite a complex task to implement this safely and efficiently. Windows has a few functions in its API that helps us implement waiting on event-objects more easily.

Registering for changes of shared data sources

The first approach is using the function called `RegisterWaitForSingleObject` [11]. This function uses a global thread pool to wait for `HANDLE`s. It accepts a pointer to a callback function that is called when the `HANDLE` is signaled. However in this section we refer to this callback function as a 'native callback function', since it requires a function pointer.

`RegisterWaitForSingleObject` works as follows. When a `HANDLE` is registered to be waited for, a wait-thread is picked from the thread pool. Windows handles what threads it uses as wait-threads from the thread pool. Here a single wait-thread can be used to wait for multiple `HANDLE`s. The default

behavior picks, once the `HANDLE` is signaled, another free thread from the thread pool to execute the native callback-function on. This thread is called the work-thread. This mechanism is shown in Figure 3.4.

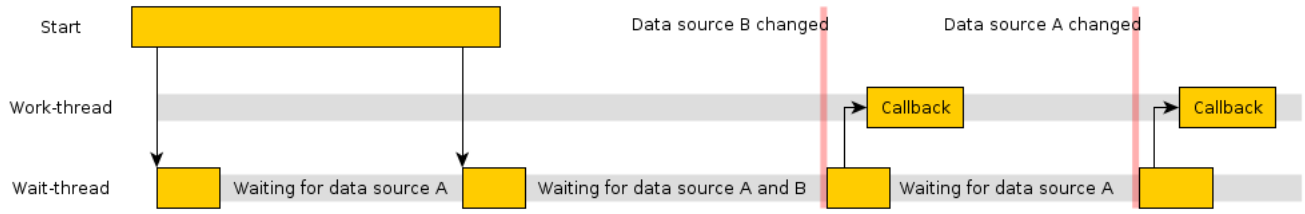

Figure 3.4: The threads of RegisterWaitForSingleObject

The advantage compared to `WaitForMultipleObjects` is that it already makes multiple threads when needed to wait for a large number of `HANDLE`s. Another advantage of Windows handling all that work for us, is that Windows could do this more efficient than we can. We only need to pass a `HANDLE` and a function, instead of a whole array. With `WaitForMultipleObjects` the whole array needs to be supplied every call, which means every time we got a signal we need to call the function again to wait for the rest of the `HANDLE`s. For `RegisterWaitForSingleObject` however, only a single `HANDLE` and function needs to be supplied each time we want to register once at the time of registration.

For most cases this method is suffice. However one aspect of Clean is that it does not have a global heap like most languages have. Clean has one heap per thread to enforce thread safety. For every thread requiring to run Clean code, the thread first needs to have its Clean environment initialized. Similar, we want to clean up the Clean environment once the thread has finished its execution. This becomes a problem with `RegisterWaitForSingleObject`. We do not care on what thread the `HANDLE` is being waited on, but we do care on what work-thread the callback is executed. Usually the same thread is used for all callbacks, but when the work-thread is busy handling a previous callback, a new work-thread is picked. This is problematic, since we have very little control when these threads are created or destroyed.

As a workaround we can detect whether the callback is executed on a thread that does not yet have an initialized Clean environment and initialize the environment on that moment. This however leaves 'dangling' Clean environments when another work-thread is picked, since we do not know when the earlier work-thread is finished.

What we need for this approach to work is more control over when and on what threads callbacks are being executed. From the documentation it did not seem possible to reliably enforce the desired method of allocating threads.

Registering for changes of shared data sources in Windows Vista

With the release of Windows Vista a new API for thread pools was introduced [12]. This API gives its user more control over what threads it should use for waiting and what threads to use for executing callbacks. First it allows the user to create thread pools themselves, so that there can be multiple thread pools inside a single process, each made for a specific task, as opposed to having one global thread pool.

This allows us to know beforehand what threads are used for waiting and what threads for working. This should solve most of the problems we had with `RegisterWaitForSingleObject`.

However this API is only available in Windows Vista and higher. Since we are aiming for iTasks to be run on any server, including Windows 2003 servers and Windows XP, we did not attempt to implement the desired mechanism using this thread pool API.

3.4.4 Conclusions

We have shown that waiting mechanisms can be used to detect when changes in data sources happen. The various methods to implement a waiting mechanism vary between their ease of implementation, resource efficiency and operating system requirements.

Waiting for changes using 'readWrite' is easy to implement, but costs a high amount of threads. Using Windows APIs we can reduce the number of threads by a factor of 15, however this is complex to correctly implement and still costs more threads as the program grows without these threads actually executing code. All of the shown methods suffer from excessive use of threads, which is the main negative aspect of the waiting mechanisms which we cannot work around.

3.5 Callback functions

Waiting for changes has shown to be hard to implement efficiently. In this section we try a very different approach. We show how callback-functions can (directly or indirectly) invoke user-code on noticing changes of data.

First we describe the basic idea how we are going to use callback functions and how we store these functions for later use. We show what problems arise when these stored functions need to be invoked and show how to use a queue to avoid these problems.

3.5.1 Storing callback functions

In the previous sections we already introduced callback functions. These functions were used to notify the user about the changes. We would wait for changes and we would call such a callback function when such a change occurred. We showed that waiting for multiple data sources can get complex, which is why we try to simplify our approach in this section.

Instead of using callback functions to be called after waiting, we allow each type of data source to handle its own detection of changes and execute the callback functions from there. This way we can implement notifications more efficiently while we do not need to wait for event-objects. However, without the use of event-objects, there can be different ways of noticing changes. This differs per type of data source. This means that each type of data source needs to implement its own way of noticing changes.

Shared data sources reading and writing memory of the program itself can make use of the write-operation of the data source. Changes are in this case always performed through this operation, therefore changes of that memory can be detected by hooking into the write-functionality of the program. This applies to memory specifically, because it is not shared across multiple processes.

For file data sources this is different. Multiple processes can write to the same file. Detecting changes of such a file needs to rely on operating system APIs to pass those changes to our program. The operating system APIs for doing this do exist which makes this is a solvable problem. Therefore we will not go into detail of implementing change-detection of file data sources.

Data sources that are retrieving data from other systems use different protocols to read and write data. Change detection should have been built into such protocols for our data source implementation to detect such changes.

One example is external databases, where change detection is not always obvious or even possible. Detecting changes depends on the database-system being used. When such an external database is not able to supply us with changes in real-time, polling can always be used as a fallback mechanism. Examples of database systems supporting change notifications are Microsoft SQL Server 2005 through query notifications [13] and CouchDB through the continuous changes API [14].

In this and further sections we show the global mechanism that can be used in combination with a data source specific implementation. We show how this global mechanism works by defining an API that the user can use to register for changes and the specific data source implementation to notify the registered users.

Like before, we want our user-code to know when a change has happened. We use the same callback function and `subscribeForChange` definition that we defined in 3.4. The implementation of `subscribeForChange` will now however not create threads nor will it wait for changes.

When `subscribeForChange` is called we want the callback function to be stored into all basic data sources so that the implementation of such a data source can access the callback functions that are registered. We add a function to the basic data source called `addCallback` that stores the callback function. `addCallback` is called for all basic data sources contained in the supplied `RWShared`.

Now each basic data source contains a list of registered callback functions. The specific implementations of the data source can now access this list making sure the callback functions can be called.

For our test we implemented this for shared memory. In the case of shared memory, `EventSet` is called right after new data is written to memory. This is where we add the functionality to call the stored callbacks. This approach did work, however it had a reliability problem.

When shared data sources are being read or written, all basic data sources first are locked in a specific order. The specific order is used to avoid causing deadlocks. There is a situation when executing callbacks where the data is not locked in the correct order. This can happen when a callback function is being executed just after writing data to memory and before unlocking the data. When the callback itself needs to write to different data sources, of which the one already locked is one of them, locking them all in the specific order to avoid deadlocks is no longer possible. This could in some cases cause deadlocks.

Another problem we found is that the callback function is executed on the thread where the data is being written. It is not executed on the thread the user expected. This can lead to problems, since the heap in Clean is not shared between threads. Values required during execution of the callback need to be serialized at all times. Even though this is not a huge problem, it is still something to avoid.

To keep the API reliable we need a way to solve these two problems. This is described next.

3.5.2 Queueing callback functions

To resolve the problem of deadlocks we need a way to execute the callbacks after all basic data sources are unlocked. To resolve the problem of executing callback functions on undesired threads, we need some way for the user to specify what thread we can execute the callback function on.

We attempt to solve these two problems by creating a queue of functions. The queue is used on an 'execution thread' where it empties and executes the functions from the queue. On the other end the queue is used by the data source by adding the callback function that needs to be executed on the queue.

For this implementation we defined a new type `Work`:

```
:: Work :== *World → *World
```

This is similar to the previously defined `Callback`, but since we can use these functions in this sections for other purposes than just callbacks, we have called these `Work`. The callback type will be reintroduced and extended later in this section.

We have called the queue where the work is queued on `WorkQueue`. The `WorkQueue` is defined as a `MessageQueue`, which is defined in section 3.4.1. With the cross-thread `MessageQueue` we can define our specific queue where we store `Work`:

```
:: WorkQueue :== MessageQueue Work World
```

Next we need a thread to be getting the work from the queue and execute that work. To do this, we have implemented a function `run` that is needed to be called on some thread. This function loops endlessly:

```
run :: WorkQueue !*World → *World
run workQueue world
```

```
// Retrieve work from the queue. This function
// waits when there is no work on the queue yet.
# (work, world) = receiveMessage workQueue world
// Execute the work.
# world = work world
// Loop.
= run workQueue world
```

To queue `Work` on the `WorkQueue` we can use `sendMessage` as follows:

```
sendMessage (print "I am doing work") workQueue world
```

In this example `print "I am doing work"` is the work that is performed on the execution thread where the `run` function is looping.

Now that we have an execution-thread and a way to queue work onto that thread, we need to make this work with the callback system we defined earlier.

Earlier we only needed one function `*World` → `*World` for a callback. Now we also need a `WorkQueue` to specify where that function is needed to be put once a shared data source has changed. We redefine `Callback` as follows:

```
:: Callback :== (Work, WorkQueue)
```

As a convenience we also define a function that queues the `Work` onto the `WorkQueue` for a particular `Callback`:

```
scheduleCallback :: Callback !*World → *World
scheduleCallback (work,workQueue) world = sendMessage work workQueue world
```

We can use this function, when a data source is changed, to queue the callback onto the execution thread. Note that we now have to store a function (`Work`) and a reference to the `WorkQueue`, where before we only stored a function.

In Figure 3.5 we can see how the different functions work. The execution thread loops in `run`, where it dequeues work from the `workQueue` by removing the first entry (workA) from the `workQueue`. Other threads can at any time schedule work to be run by queuing the work onto the `workQueue`.

To integrate this in the shared data sources we use the function `subscribeForChange` like we defined before, but now with the new `Callback` definition. The rest of the implementations are the same.

We also use `sharedMemory` to create a shared data source in memory. This part of the standard functionality of shared data sources in Clean. We use it for the `workQueue`, so that the queue is created in memory, and for `exampleShare`,

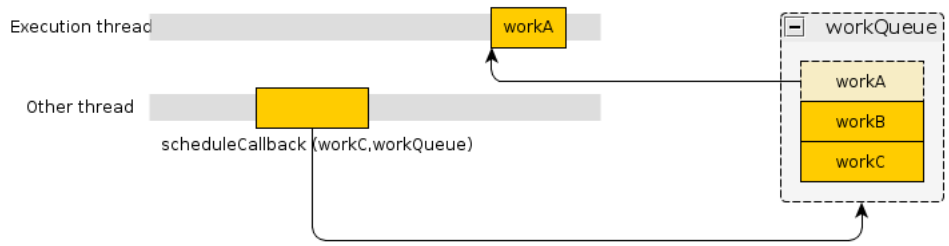

Figure 3.5: WorkQueue with its execution thread and thread using scheduleWork.

which we use as an example data source. We pass a value to `sharedMemory`, which becomes the initial data of the shared data source.

Another function we use is `sleep`. This function waits for a specified amount of milliseconds. It is only used in this example to illustrate multiple changes that have some time in between them.

Now we can make use of all the defined functions as the following example shows:

```
Start :: *World → *World
Start world
    # (workQueue, world) = sharedMemory [] world
    # (exampleShare, world) = sharedMemory 0 world
    # world = subscribeForChange (printOnChange,workQueue) exampleShare world
    # world = fork (exampleThread 0 exampleShare) world
    = run workQueue world
where
    exampleThread :: Int (RWShared Int Int World) World → World
    exampleThread counter exampleShare world
        # world = sleep 1000 world
        # world = write counter exampleShare world
        = exampleThread (counter+1) exampleShared world

    printOnChange = print "exampleShare has changed!"
```

In this example first the workQueue and the shared data source where we want to be notified of changes (exampleShare) are created. Then the function `printOnChange` is registered to be called when `exampleShare` is changed. Thereafter a new thread is started, which writes to `exampleShare` every second. Lastly `run` is used to start retrieving and evaluating the work that is on

the workQueue.

Every time `write` is called from `exampleThread` the data in `exampleShare` is changed. This triggers code in `sharedMemory` that calls `scheduleWork`, that puts the function `printOnChange` onto `workQueue`. `run` was waiting for work to be put onto `workQueue` and is now triggered. `run` removes the `printOnChange` from `workQueue` and evaluates it. Then the message `"exampleShare has changed!"` is printed.

`printOnChange` is being executed on the thread where `run` is being executed. `exampleThread` only writes to the shared data source. This is the behavior we wanted.

Note however that `printOnChange` is being executed for all changes of `exampleShare` until the whole program terminates. At the moment there is no way to stop listening for changes.

To stop listening for changes using a certain callback, we need to have a function that can actually cancel the callback. We call this function `cancelCallback`. It also requires us to distinguish between the different subscriptions that have been made. This is not the case with (`Work`, `WorkQueue`): `WorkQueue` is most of the time the same instance and `Work` is a function that is not uniquely identifiable. The following example shows that this can not work:

```
example workQueue exampleShare world
    # world = subscribeForChange (printOnChange,workQueue) exampleShare world
    # world = subscribeForChange (printOnChange,workQueue) exampleShare world
    # world = cancelCallback (printOnChange,workQueue) exampleShare world
    = world
```

In this example it is unclear whether we want to cancel the first subscription, the second or both. Therefore we need `subscribeForChange` to give us a unique identifier for the subscription we created. We call this type a `CallbackId`.

To make the `CallbackId` unique we have different options. One option is to have some globally stored number in our program that we increase every time a `CallbackId` is needed to be made. In this case `CallbackId` would be a number. Another option is to store such a unique number in each basic data source. When we need to get the `CallbackId` of that basic data source, we can increase that number and return that new number. This means the number is only unique for that particular basic data source. When we want to have a unique number of a composed data source, we can take all basic

data sources, increase all their numbers and the result of the function is a list of those numbers. The type of `CallbackId` in this case would be a list of numbers.

In our case we chose for the second option, since we want to depend on global variables as little as possible. Therefore the definition of `CallbackId` is:

```
:: CallbackId :== [Int]
```

Next we have to alter the definition of `subscribeForChange` so we can get that `CallbackId` to be used in `cancelCallback`:

```
subscribeForChange :: Callback (RWShared r w *World) *World → (CallbackId, *World)
```

The of `cancelCallback` is extended with this `CallbackId`:

```
cancelCallback :: CallbackId (RWShared r w *World) *World → *World
```

Now we can indeed distinguish between the different subscriptions, as the following example shows:

```
example :: (RWShared [Work] [Work] World) (RWShared Int Int World) World → World
example workQueue exampleShare world
    # (callbackidA, world) = subscribeForChange (printOnChange,workQueue) exampleShare world
    # (callbackidB, world) = subscribeForChange (printOnChange,workQueue) exampleShare world
    # world = cancelCallback callbackidA exampleShare world
    = world
```

For our use case in iTasks we also wanted to only wait for a single change, instead of waiting for all changes indefinitely. Tasks in iTasks depend on certain data sources. When a task reevaluates, its dependencies changes. This means that there is a chance a callback is not needed to be subscribed to when the task is reevaluated. For this reason we want to wait for a single change and resubscribe when needed. We explore how to do this based on `subscribeForChange` and `cancelCallback`. There are different possibilities to solve this. What we found intuitive was canceling the callback when the callback is first being executed. To implement this intuitively, we need to have the `CallbackId` in the callback-function, so that we can cancel the next callbacks using `cancelCallback` and the passed `CallbackId`.

This asks for another extension to `subscribeForChange`. Instead of passing a function `World → World`, which we named `Work`, we need an extra argument to pass the `CallbackId` to the callback function. We can define this new callback function as `CallbackId *World → *World`, which is equal to `CallbackId → Work`. For convenience we create a new type, much like `Callback` to be used in `subscribeForChange` as follows:

```
:: CancellableCallback :== (CallbackId → Work, WorkQueue)
subscribeForChange :: CancellableCallback (RWShared r w *World) *World
                   → (CallbackId, *World)
```

With this redefinition, we need to alter the function `printOnceOnChange` in our example to be able to accept the introduced `CallbackId`:

```
printOnceOnChange callbackid world
    # world = cancelCallback callbackid exampleShare world
    # world = print "exampleShare was changed" world
    = world
```

Figure 3.6 illustrates how the system works with one shared data source, one callback and one time the data is changed.

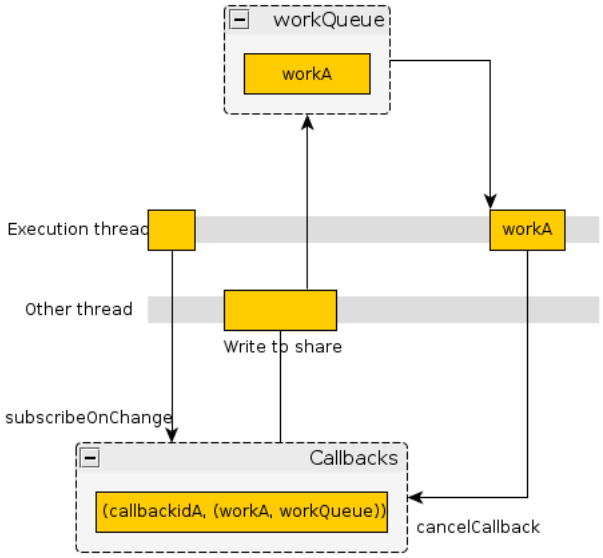

Figure 3.6: Subscription and execution of a callback

As seen in Figure 3.6, this approach seems to be working reliably. However, there is an edge case where the callback is still executed multiple times.

When the shared data source is being written to twice (or more) very fast in succession, it will often happen that the callback is not yet picked up by the execution thread. When that occurs there are multiple callbacks on the workQueue. These are both executed and therefore the callback is executed multiple times. Even though multiple callbacks for a single change

is permitted, it can be confusing to still see callbacks being executed after the `CallbackId` was already canceled using `cancelCallback`. For this reason we will eliminate further callbacks that happen after calling `cancelCallback`.

Figure 3.7 illustrates this problem. The second time `cancelCallback` is called, it essentially does not remove any callbacks, since all related callbacks are already removed by the first time `cancelCallback` is called. We can exploit this knowledge by only continue executing if there were actually callbacks removed by `cancelCallback`.

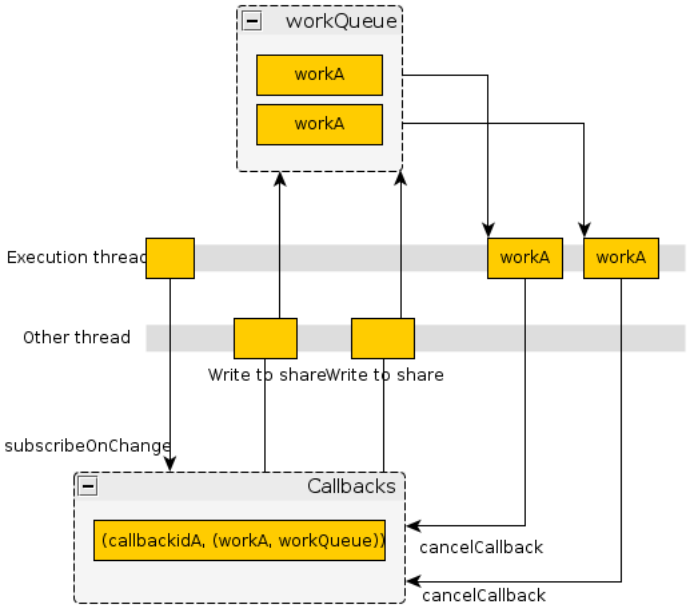

Figure 3.7: The shared data source is written twice before the work could be executed.

To achieve this, we alter `cancelCallback` such that it returns whether it has found and removed the specified `CallbackId`. The new definition of `cancelCallback` then becomes:

```
cancelCallback :: CallbackId (RWShared r w *World) *World → (Bool, *World)
```

The result of `cancelCallback` becomes `True` when the callback was indeed found and canceled and `False` when it did not find the specified callback.

We alter the previous example to make use of the new definition of `cancelCallback`:

```
    printOnceOnChange callbackid world
        # (found, world) = cancelCallback callbackid exampleShare world
        | found = world
        | otherwise = print "exampleShare was changed" world
```

Now the actual user-code (printing a message) is not executed a second time, even though the 'work' itself is still executed twice. Since this only happens in edge cases, it does not matter in terms of performance.

The example shows how users can implement this mechanism themselves. It is better to create a function that does all the work consistently. We introduce the function subscribeForOneChange:

```
subscribeForOneChange :: !Callback (RWShared r w *World) !*World
                      → (CallbackId, *World)
```

Note that the callback in this function is still cancelable from outside the callback itself, but not from within the callback. It is defined like this since the callback is already canceled before the actual user-work is being executed.

The implementation of subscribeForOneChange is:

```
subscribeForOneChange (work, workQueue) share world
    = subscribeForChange (workWrapper share work, workQueue) share world
where
    workWrapper :: (RWShared r w *World) Work CallbackId *World → *World
    workWrapper share work callbackid world
        # (found, world) = cancelCallback callbackid share world
        | found = world
        = work world
```

We can rewrite the previous example to make use of subscribeForChange:

```
example workQueue exampleShare world
    # (callbackidA, world) = subscribeForOneChange
                (printOnceOnChange,workQueue) exampleShare world
    = world
where
    printOnceOnChange world = print "exampleShare was changed" world
```

In Figure 3.8 the new mechanism is visualized. It shows that the work is now only executed once. The workWrapper will not execute the work when it has not found the callbackid while canceling the callback.

With this approach we still maintain all properties of shared data sources, since the user-code is executed on the execution thread *outside* a lock of the

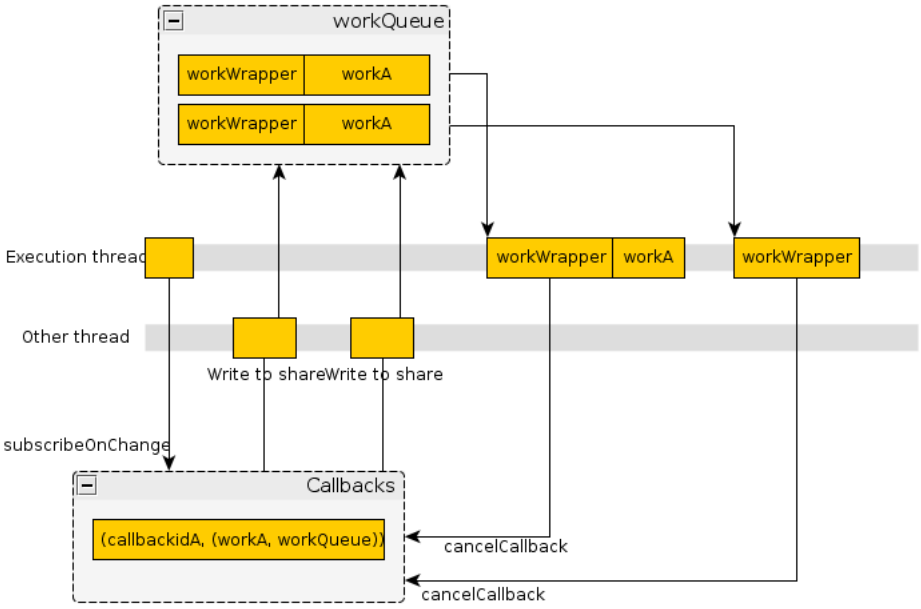

Figure 3.8: The share is written twice before the work could be executed and workWrapper will not execute the work a second time.

resources. Because the data sources are designed to be thread-safe we do not have to worry about concurrency problems when reading or writing.

We do miss some of the changes when they are performed in fast succession, but will not miss the latest state of the data. As we can see in Figure 3.8, two write operations are being performed and only one callback is being executed. The callback that is executed does however execute *after* the last of the two write operations. Therefore we did not miss that latest state of the data.

Changes that happen after the callback is executed are 'ignored'. Either the subscription was removed from the data-sources, or the `workWrapper` sees its callbackid was already canceled. Therefore the callback function is always executed at most once, but not more.

The resource-usage of this system is still something to be concerned about. The functions we need to store in the list of callbacks and the workqueue always need to be serialized. All data that is curried in that function is also stored inside that serialization. This is required, since the function and its curried data are going to be accessed and used by other threads. In Clean,

the heap is not shared between threads, therefore all data needs to be stored into one serialization.

Another related point is the fact that we now define what we want to do on subscription: we supply a function with all of its required data curried into that function when we subscribe for changes. What we actually would like to do is figure out what to do when the change has happened. This could minimize the amount of data that is shared between threads, since data is not needed to be curried into a function that needs to be executed on another thread.

3.6 Queueing messages

In the previous sections we were storing functions. These functions had some of their required data curried into them so that this data could be used upon execution of the function. This approach is not very efficient, since that data plus the type-information of the function needs to be serialized to be usable across the multiple threads.

If we can let the user-code stay in the same thread as we were subscribing, we would not need to serialize as much data, since most data is already available in the heap of the execution thread.

This is where messages come into play. Messages are data from which *we can determine* what to do, opposed to functions, which *describe* what to do.

Instead of the earlier `WorkQueue`, which consist of functions, we can have a `MessageQueue` which consist of messages. A message can be of any type. The user determines what type a message should be, so the user is also in control of the creation of messages.

Upon subscription the user creates a message and passes the message alongside a `MessageQueue` to `subscribeOnChange`. This pair is then stored in each basic shared data source. When a subscribed shared data source changes, the messages are sent to the `MessageQueue`. The message is then picked up by user-code that constantly receives messages from that queue. It then determines what it should do, for instance using pattern-matching.

In the case of iTasks we only need to store the id of the task we want to re-evaluate on changes. We create a message that only contains an id and on receiving such a message, we can find the task. In this scenario a message could be defined as:

```
:: ITaskMessage = ReevaluateTask TaskId
```

Just like the solutions earlier, we would have a function where we can subscribe for changes to a certain shared data source:

```
subscribeForChange :: (a, MessageQueue a *World)
    (RWShared r w *World) *World → (CallbackId, *World)
```

In this function `a` determines the type of the message and `MessageQueue a *World` determines the queue to which the message can be sent. As an example, in the case of iTasks, we would pass in (`ReevaluateTask 3`, `itaskMessageQueue`), so that we can later retrieve `ReevaluateTask 3` from `itaskMessageQueue` to know we should re-evaluate task 3.

Retrieving messages from the `MessageQueue` is much like retrieving functions from the `WorkQueue`. This also happens in an loop. However instead of executing functions, we can now pattern-match the messages and execute code depending on their contents. We can define such a mechanism as follows:

```
run :: (a *World → *World) (MessageQueue a *World) !*World → *World
run messageHandler messageQueue world
    # (message, world) = receiveMessage messageQueue world
    # world = messageHandler message world
    = run messageHandler messageQueue world
```

This function accepts a `messageHandler` and a `messageQueue`. The queue is where the message is put and where `run` gets its messages from. The `messageHandler` is where the actual pattern-matching is going to happen. This function determines what should happen when a certain message is received.

For iTasks such a `messageHandler` could look as follows:

```
handleMessage (ReevaluateTask taskId) world = evalTask taskId world
```

This function matches for `ReevaluateTask` and determines what to do on receiving such a message. In this example it always executes `evalTask`, which is an example function that evaluates a task in iTasks.

This implementation of `subscibeForChange` is very similar to the implementation of the earlier defined `subscribeForChange` using functions. Now, instead of storing a function, we now store messages. There is however one part of the previous implementation that we can not accomplish using arbitrary messages. In the previous solution we wrapped the callback function with a function that would cancel the callback. With arbitrary messages we cannot wrap messages with other messages from within `subscribeForChange`, since the type of a message is only known in user-code.

The naive solution to this problem is to cancel the callback just before we send the message to the queue. This, however brings stability problems for certain types of data sources. In the case of `SharedMemory` we notice the change of data inside the write-operation. When this operation is working, the data source is locked. When we want to cancel the callback might need to be removed from several different data sources, since the subscription can be made on a composed data source. In this case, these data sources all need to be locked before we can remove the callback. One of these data sources is the data source we are writing to. This data source is already locked. When we lock the other data sources we might end up with a deadlock, since locking of the basic data sources will not be performed in a fixed order any more.

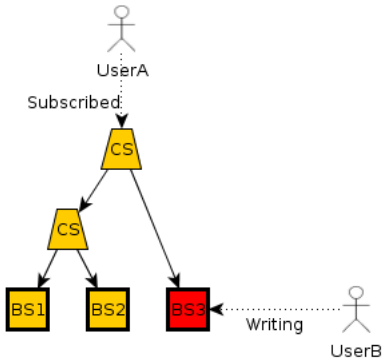

Figure 3.9: Deadlock situation when subscribing and writing SharedMemory

In Figure 3.9 we see an example of the situation that can occur. User A subscribes to a composed data source. The data source is composed of 3 basic data sources: BS1, BS2 and BS3. They are also numbered in this order and will therefore be locked in that order (BS1, BS2 and BS3). User B writes a new value to BS3. While BS3 is locked, we want to remove the callbackid from the other basic data sources (BS1 and BS2). This would cause the locking to occur in this order: BS3, BS1, BS2. This order is incorrect and causes dead-locks when yet another user is going to lock BS2 and BS3.

To solve this situation we need to first unlock the basic data source we are writing to. However in that case, we also need to unlock all other basic data sources that were part of the same write-operation. Afterwards we would need to relock all basic data sources that are part of the subscription and remove the callback from those.

The reason callback functions do not suffer from this problem is because they can execute arbitrary functions at a later time. At a later time the data sources are not locked anymore and the callback function can relock the basic data sources in the correct order.

After measurements, we decided this was not worth the effort of the efficiency-gain we would get when moving from callback functions to callback messages. For this reason we did not implement this method and kept using callback functions. However, this solution is worth looking into for later research.

3.7 Conclusion

In the previous sections we discussed the different ways of doing change notifications we have found.

Polling has shown to be a concept applicable to any sort of readable data source. It is suitable for tracking changes of small sets of data. However it becomes inefficient when multiple sources needs to be tracked.

Detecting changes inside write-functions of iTasks works very efficient, but will only notice changes performed by the application itself. When multiple programs or systems alter the data, other solutions must be found.

Waiting for changes is a solution that can make good use of the atomic properties of shared data sources. It does however require threads to be created to wait on. It has many different possibilities when it comes to implementation to make it more efficient, but that makes the system very complex and in the end still not efficient enough.

Callback functions by themselves, as seen in the previous sections, are unreliable in the sense that they invoke user-code on threads that the user might not suspect. Queueing callback functions resolves this problem. Handling the queue requires a minimum of one thread. Due to the strictness of functions, resources that are required to handle the notifications need to be curried into the callback function. Curried data and functions needs to be serialized to be used on other threads. This can become inefficient.

Queueing messages makes the queue reusable for the user, so that other kinds of messages can be put on the queue to be handled. It also gives the user more control by letting it handle messages its own way, without the need of currying needed resources into a function. However the increase of control for the user makes the queue less flexible. In our approach the

change notification system will not have enough control to reliably handle the notifications.

The use of callback functions on a queue has shown to be the most flexible and reliable solution that we have found. Even though a message queue can be more efficient, we have not found a good way to make it flexible enough. For these reasons we use callback functions on a queue in iTask to handle change notifications.

Chapter 4

Change notifications in iTasks

Now that we have established a way to detect changes and notify user-code about them, we can start looking at how we have integrated this into iTask.

In iTasks, when users operate on a task, the task is evaluated using its current state with the added input. While evaluating, the task can also use (read and write) shared data sources. This means that the task not only depends on its own local state and user-operations, but also on the contents of the shared data sources it uses. What we want is to reevaluate a task when the data sources have changed that the task depends upon.

First we need some way of determining what data sources a task is depending on. This can be achieved in several ways. One way would be to let the developer of the task specify explicitly which data sources the task relies on. This could be done by annotating workflows with the depending data sources. Another way is to determine this from the behavior of the task, so that the system can know implicitly on what data sources the task depends. We chose the latter so that tasks can be defined more easily. We found that manually specifying what data sources the task relies on is a burden for the user and the user could also make mistakes.

To do this we want to extract the depending data source of a task by tracking the operations the task uses on data sources during its evaluation. iTask has a fixed set of operations that a task can perform on data sources. These range from simple operations like `readShared` and `writeShared`, which just reads and writes data respectively, to more complex operations like `showSharedInformation`, which reads and shows the information to the user.

Next we want the task to be reevaluated when those depending data sources have changed. This means we want to reevaluate the task when

the data sources have changed during or after the evaluation of the task. However, we do not want the task to be reevaluated when the task itself is changing its data sources during its evaluation. We only want to reevaluate the task when the data sources have changed by other tasks or programs during the evaluation.

To accomplish this we introduce a list in which we keep track of what shared data sources we have used during the evaluation of a task. We extend operations that can be executed by a task on data sources like `showSharedInformation` to add the data source it operates on to that list.

After the evaluation of the task, we can subscribe to the data sources in the list using `subscribeForOneChange`. This way we can reevaluate the task when any of its depending data sources have changed. This works well for circumstances where we only have a single data source that needs to be subscribed to. However when there are multiple data sources, we need to make sure all data sources are unsubscribed before we begin the evaluation of the task: we do not want to be notified of the changes of the task itself.

A simple way of doing this is canceling all subscriptions of the task to be evaluated. We use `cancelCallback` on all existing subscriptions to make sure there are no subscriptions when the task itself is evaluating. Doing this however raises another problem: the changes from other programs while evaluating the task are now missed, since there are no subscriptions anymore. This behavior is shown in 4.1.

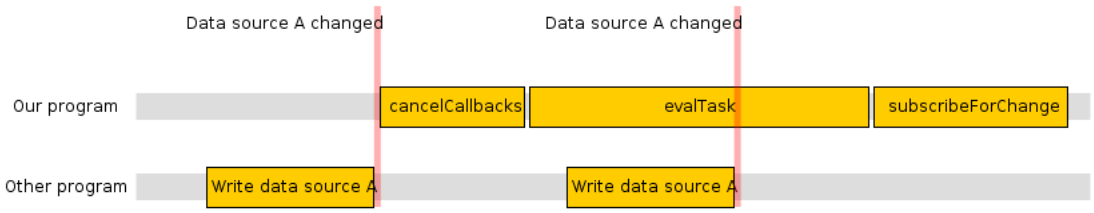

Figure 4.1: Changes of data sources will not be picked up during task evaluation without comparing versions after evaluation

For this reason we use the version number of each of the depending shared data sources. The version is already part of the existing specification of shared data sources. It can be used to tell us at what version we have last seen the value of the shared data source. This makes it possible, when the task is evaluated, to compare the last seen version to the latest version of

each shared data source. This way we can know whether the data source has changed, since all data sources have the property to always increment their version when they change.

Each operation that a task can use on a shared data source also assign the version of the last-seen value of that shared data source to the list of depending data sources. It depends on the kind of operation when to add a shared data source to the list of depending shared data sources and what last-seen version is assigned. For instance, reading operations always add a new entry to the list when the shared data source is not yet in the list. The last-seen version that is assigned to the shared data source is always the version it reads from the data source. When the shared data source is already in the list, it only updates the last-seen version with the one it just read.

On the other hand writing operations never add a new entry to the list, but only updates existing entries. Writing a shared data source without reading its value, means that the task does not depend on the data of that data source: it did not do anything based on its value. For this reason all write operations only update the version of existing depending shared data sources to the version of the data sources after a new value was written. When the shared data source being written to does not yet exist in the list of depending data sources, it will not be added by this operation. This way the version of the depending shared data sources can always be used to see whether the data sources have changed compared to their last-seen operations.

Figure 4.2 displays the process of handling a task evaluation in graphical form. Before a task is evaluated, all its subscriptions are first canceled. Then we start evaluating the task with an empty list of depending shared data sources. When an operation is used on a data source, the operation adds or updates an entry in the list of depending data sources with the respective version number. When the task has finished evaluation, we compare the last seen versions to the actual versions of the depending data sources and reevaluate when these do not match: the shared data source was modified by another task or program while the task was evaluating. When the versions do match, we can subscribe to those data sources to wait for future changes.

This mechanism can work well, however we still have to fill in the details. There is still a way to miss changes and those are the changes that happen between execution of version comparison and subscription. Suppose at this point the versions have matched, but right after the comparison one of the

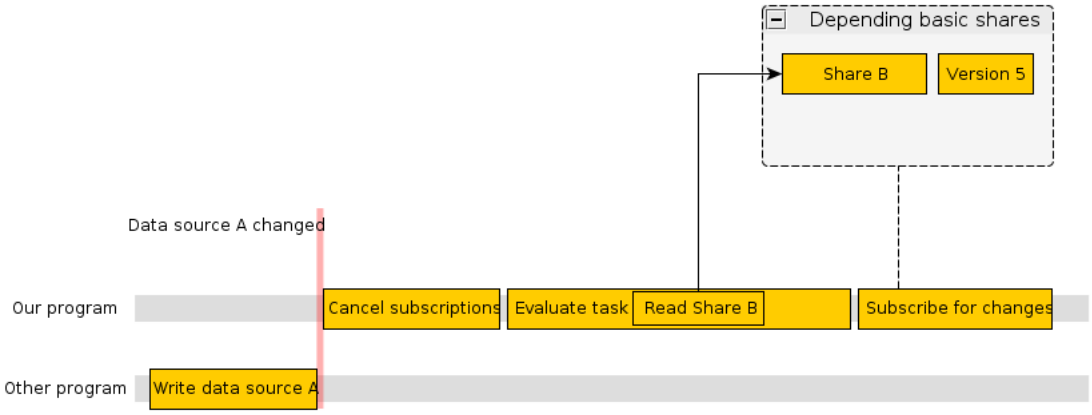

Figure 4.2: Task evaluation with the list of depending shared data sources

data sources changes and notify all subscriptions. Now this notification is missed, since `subscribeForOneChange` has not been called yet: the change goes by unnoticed. These two steps need to be combined into one operation that executes within a lock, so that changes in between these steps can not occur.

Another problem we found are the number of subscriptions this mechanism results in when a task uses multiple composed data sources. At the moment we only talk about data sources in the general sense, which means basic data sources as well as composed data sources. However, changes of data can only occur to basic data sources, where the actual data resides. This also means that we only need to subscribe to those basic data sources.

As shown in Figure 4.3 one task (TaskA) can use multiple composed data sources. When denoting the basic data sources of these composed ones, we can see a lot of overlap. For example, CS1 and CS2 both contain BS1. Therefore BS1 is indirectly referenced twice. This means that when we subscribe for changes of CS1 and subscribe for changes of CS2, we are actually subscribing for changes of BS1 twice.

In the example this would mean that we would create 9 subscriptions (2 for BS1, 3 for BS2, 3 for BS3 and 1 for BS4), where we would only need 4 subscriptions.

For this reason we would like to reduce this number to the number of basic data sources that we used. To do this, we introduce a new function (`getBasicShares`) that will result in a list of all basic data sources contained in a specified data source. Using this function we can change our dependency

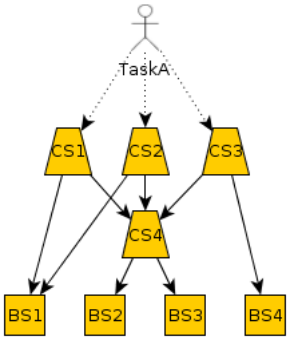

Figure 4.3: TaskA uses multiple composed data sources with overlapping basic data sources

mechanism: instead of keeping a list of general data sources, we can keep a list of basic data sources. Removing duplicates from this list will result in a minimal set of the basic data sources that the task depends on. Subscribing to these will then result in a potential reduction of subscriptions.

Now we need to use these basic data sources and we still need a solution for the unnoticed changes. We define a new alternative function for `subscribeForOneChange` that we can use in iTasks. We call this function `subscribeForOneChangeBS`:

```
subscribeForOneChangeBS :: Callback [(BasicShare *World, Version)] !*World
                        → (CallbackId, *World)
```

Here we use 'BasicShare' so that we can only pass in basic data sources and not combined ones. This is a type of the shared data source-implementation in Clean.

`subscribeForOneChangeBS` accepts a list of basic data sources combined with their last-seen version. The function compares the last-seen version of each shared data source with the current version for all passed shared data sources in the list. When one of the data sources have a newer version, the callback is used instantly and the callback-function is pushed directly to the `WorkQueue`. When the data sources did not change and their version numbers match, a subscription is made for all these basic data sources. The `CallbackId` can then be used to cancel the subscription when needed. This all happens within a lock on all passed basic data sources. It makes sure no changes can happen between comparing the versions and making the subscription.

With these changes our mechanism for evaluating a task has changed. In Figure 4.4 the new mechanism is visualized. Here we have a task that uses a composed data source named 'Share B'. This composed data source consists of two basic data sources, 'BS1' and 'BS2'. Notice that the basic data sources are added to the list of depending shares. The version numbers are the versions of those basic data sources. Also notice that the comparison of versions and the subscription for changes to the data sources now happen in a single operation.

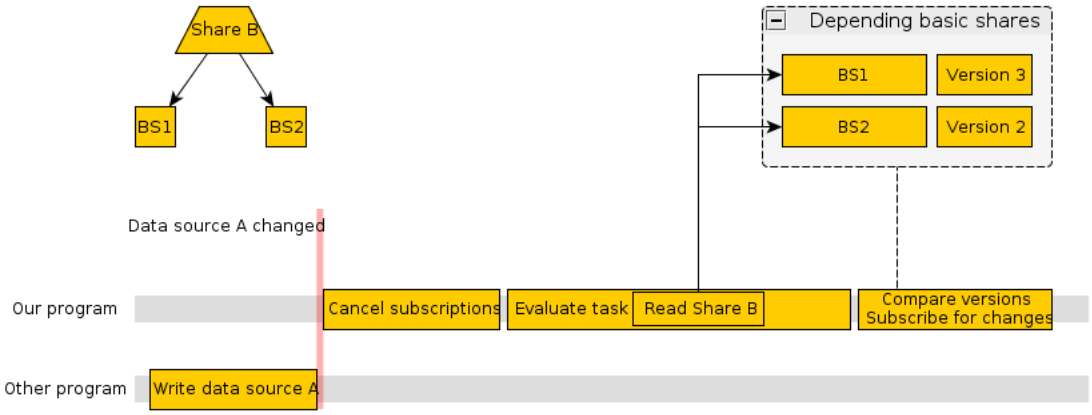

Figure 4.4: Task evaluation with the list of depending basic data sources

In this section we have seen how we used our previously defined notification system in iTasks. For our needs it was required to alter a number of aspects to get the desired behavior. These aspects include:

- Ignoring changes from the task itself
- Subscribing based on a version, so that subsequent cancellations and subscriptions can be made reliably without missing changes
- Reducing the number of subscriptions by subscribing to basic data sources instead of data sources in general

Even though these are important aspects for our needs, they also come with a cost. To be able to ignore changes from the task itself we need to be able to cancel existing subscriptions. In turn that requires us to use the subscribing-mechanism based on a version. The downside of this mechanism

is that a version number is needed to be retrieved from the data source at the end of task evaluation, so that we could compare the last seen version with the latest version. Depending on what type of data source we use, this can take some time. For example, a version can be stored inside a file and reading a file is costly. This downside was worth the cost for our needs, though this will not apply to all other applications of data sources and change notifications.

The other aspect that we changed was subscribing to basic data sources instead of general data sources. Knowing what basic data sources are residing in a composed data source allows us to more effectively know what changes to track without overlap of data sources. This results in less subscriptions. Even though this is more efficient it also results in the program knowing more about the inner workings of data sources. This can result in unwanted circumstances when security plays a more prominent role: it is possible to know from and to what basic data sources it is being read from and written to, even though that data should be hidden. Again, for our purposes that was worth the cost, but this will not apply to all applications.

We have seen how elements of change notifications can be used in a system like iTasks. The simple way of subscribing to a data source is replaced by a more feature-rich functionality allowing user-code to get basic data sources and use basic data sources, along with their versions, to subscribe to. We have also shown that this applies to our needs for iTasks, but not without costs. Other applications might have different requirements and might need another variation of the solution shown here.

Chapter 5

iTasks simulation

In the previous section we have shown how we used change notifications in a system like iTasks. iTasks has been worked on for a number of years. It has gone through a great number of iterations. It now has a web-server to deliver content to its users, a web-interface to represent tasks in a GUI through the browser and a number of other systems to facilitate the process. Needless to say, the system as a whole is complex. At the time of writing the system also did not include a recent version of the Clean implementation of shared data sources and did not yet support threads. These aspects make it hard to integrate the system for change notifications we have shown.

To be able to find what was needed for change notifications to do in iTasks we developed a very simplified version of a task oriented framework like iTasks. This test-system makes it easier to make changes and easier to find the problems that would happen when we use change notifications to evaluate tasks.

This system includes a minimal set of iTasks features. The system has a way of defining simple tasks, a way to evaluate tasks and a way for tasks to interact with shared data sources. We also use the same kind of structure as iTasks has to implement these features.

To operate, the existing iTask implementation uses a global state to store and access its tasks and store the currently evaluated task. Next to that, it uses a list of read shared data sources that it fills while evaluating its task, similar to the list of depending data sources from section 4. In our simulation we use a similar global state to match the implementation of iTasks.

We have defined our global state as follows:

```
:: *IWorld = {
```

```
    tasks :: Shared (Map TaskId Task) World,
    currentTask :: Maybe TaskId,
    readShares :: ReadShares,
    scheduler :: WorkQueue,
    world :: *World,
    }
```

We first have the defined tasks, each named using a `TaskId`. This is accessed as a shared data source, so it can be accessed from different threads. Next we have the currently evaluating task, which is identified using its `TaskId`. Because most of the time there is no task being evaluated, the value is of type `Maybe`. `readShares` is similar to the implementation of iTasks and contains the data sources the currently evaluating task is depending on. The `scheduler` is the queue where the callbacks are pushed onto. Lastly we have the global state `World` which is used by Clean to access external operations, like file IO.

The tasks in iTasks are very complex structures that are hard to use as-is in our simulation. For this reason we simplified tasks to only a name (`TaskId`) and an action. The action in our case is just a function that passes the global state. In such a function we define what a task would do when it was evaluated.

We also need some way to keep track of what subscriptions have been made for each shared data source. This is needed to be able to cancel the subscription at the start of each evaluation. On creating a task this list is empty.

We end up with the following definitions:

```
:: TaskId :== String
:: InnerCallbackId :== Int
:: Task = {
    taskid :: TaskId,
    action :: (IWorld → IWorld),
    shares :: [(BasicShare *World, InnerCallbackId)]
    }
```

Note that we use `InnerCallbackId` for callback-identifiers that relate to basic data sources. In section 3.5.2 we defined `CallbackId` as a list of `Int`, where each element is a version of a basic data source. Here we split that list up so that each element is paired with their related basic data source.

The above definition makes it possible to define tasks, like the following example of a task named `taska`:

```
taska :: Task
taska = {taskid = "taska", action = (log "TaskA is executing"), shares = []}
```

Now that we can define tasks we need a function that implements the evaluation process as described in section 4. This function would require a task to be evaluated and use it on the global state as defined before. This results in the following declaration:

```
evalTask :: !Task !*IWorld → *IWorld
```

This function essentially executes the action that is defined within the task, however it also handles the subscriptions for that task. It makes sure that the field `readShares` is empty before executing the action. The `readShares` field is filled while the task is evaluating. How this works will be explained in this section while we describe the process of reading and writing shares. After executing the action, the list in `readShares` is used to subscribe for later changes. These subscriptions are stored inside the field `shares` of the task. When the task is evaluated later using `evalTask` again, the field `shares` is used to know what subscriptions are made for the task so that it can cancel all these subscriptions.

Now a task can be evaluated, however until now we have not yet been able to interact with shared data sources as a task. We define two new functions that emulate some of the interactions performed by a real iTasks-system, like `showSharedInformation` and `updateSharedInformation`:

```
readShare :: (RWShared r w *World) *IWorld → *(r, *IWorld)
writeShare :: w (RWShared r w *World) *IWorld → *IWorld
```

Naturally these functions read and write a shared data source respectively. Internally these functions use the functions of shared data sources to read and write their data, like `readWrite`. However these functions also need to keep track of the data sources they have read and written. They need to record at what version they are read and what version is written to the shared data sources. This means these functions add the shared data source to the list `readShares` of the global state. The basic data sources as well as their versions are put in the list. Later this list can be used by `evalTask` afterwards to create subscriptions for these data sources. Creating the subscriptions happens after the execution of the task's action-function.

Now we have everything in place to define tasks, evaluate tasks and have the tasks interact with shared data sources. To illustrate the workings we are going to show an example with a single shared data source and three tasks.

One task reads the data source, another only writes to the data source and the third first reads and then writes to the data source. First we define the shared data sources as follows:

```
(testShare, world) = sharedMemory 0 world
```

This shared data source contains an integer, initially with the value 0. Next we define the three tasks, all within the context of the same function (`testTasks`) to share `testShare` more easily:

```
testTasks testShare world = /* ... */
where
    taskA :: Task
    taskA = {taskid="taskA",action=eval,shares=[]}
    where
        eval :: !*IWorld → *IWorld
        eval w
            # (v,w) = readShare testShare w
            # w = log ("TaskA read value " +++ toString v) w
            = w

    taskB :: Task
    taskB = {taskid="taskA",action=eval,shares=[]}
    where
        eval w
            # w = log "TaskB writes value 3" w
            # w = writeShare 3 testShare w
            = w

    taskC :: Task
    taskC = {taskid="taskC",action=eval,shares=[]}
    where
        eval w
            # (v,w) = readShare testShare w
            # w = log ("TaskC is increasing value from " +++
                toString v +++ " to " +++ toString (v+1)) w
            # w = writeShare (v+1) testShare w
            = w
```

Here we see that `taskA` reads the value from the shared data source and logs its read value. `taskB` writes value 3 to the shared data source. `taskC` increases the value of the data source by one and logs its read and to be written values.

To increase the value it first does a read operation and later a write operation with the increased read value.

Now we have to simulate users triggering an initial evaluation of the tasks. To do this, we execute `evalTask` for the different defined tasks in different orders to see their outcome in the log. As a first example of a user interaction, we first evaluate `taskA`, then wait for a second and thereafter evaluate `taskB`:

```
userInteraction1 w
    # w = evalTask taskA w
    # w = sleep 1000
    # w = evalTask taskB w
    = w
```

This function results in the following logged output:

```
TaskA read value 0
[Waiting 1 second]
TaskB writes value 3
TaskA read value 3
```

As we can see, `taskA`, waiting and `taskB` are all executed like we defined in our user interaction. However, `taskA` is executed again after `taskB` has executed. This is the behavior we wanted. Since `taskA` first reads `testShare`, it becomes dependent on that shared data source. Therefore a subscription is created, so every time `testShare` changes, `taskA` is reevaluated. When `taskB` writes the value 3 to `testShare`, a change notification is generated and the callback of the subscription is invoked. The callback in turn evaluated `taskA` using `evalTask`. Therefore `taskA` reads the shared data source again and prints its new value, which is now 3.

A more interesting example is one where we use `taskC`, which both reads and writes. We show this using another user interaction:

```
userInteraction2 w
    # w = evalTask taskA w // User-action 1
    # w = sleep 1000
    # w = evalTask taskB w // User-action 2
    # w = sleep 1000
    # w = evalTask taskC w // User-action 3
    # w = sleep 1000
    # w = evalTask taskB w // User-action 4
    = w
```

We have annotated the different user-actions with numbers, so that we can refer to them more easily. This user-interaction results in the following output:

```
TaskA read value 0
[Waiting 1 second]
TaskB writes value 3
TaskA read value 3
[Waiting 1 second]
TaskC is increasing value from 3 to 4
TaskA read value 4
[Waiting 1 second]
TaskB writes value 3
TaskA read value 3
TaskC is increasing value from 3 to 4
TaskA read value 4
```

Here the first three actions are the same as in the earlier example, `taskA` gets evaluated (User-action 1), then `taskB` is evaluated (User-action 2) and `taskA` is reevaluated right thereafter. Now `taskC` is evaluated in User-action 3. This results in `taskA` being reevaluated, since `taskC` has changed the value of `testShare`. Notice that `taskB` is not reevaluated right thereafter: it is not depending on the value of `testShare`, since it has not read the value of the shared data source.

Now due to user-action 4 we evaluate `taskB` for a second time. This results in `taskA` and `taskC` being reevaluated. `taskA` was, just like the previous interactions, depending on the value of `testShare`, but `taskC` is now also depending on its value: it has read `testShare`. Right after `taskC` evaluated, `taskA` is reevaluated once again, since `taskC` has increased, and therefore changed, the value of `testShare`.

This process works as expected. All tasks that read a data source are evaluated once the data source has changed. The tasks always evaluate with the latest value in the data source.

Chapter 6

Related work

In this thesis we touched a number of topics: noticing changes in a reliable way, notifying user-code about changes and reacting to the changes efficiently. We only focused on shared data sources and iTask to get to a solution that fits these systems. Many contributions have been made about notifications for changing data and notifying other systems. We show related work that can be used alongside the information in this thesis.

There are different database systems that support change notifications in different ways[13][15][14]. In [16] the workings of versions and change notifications are documented for a database-system called ORION. ORION is a object-oriented database that features change notifications on objects by handling updates, deletions and creation of those objects as well as changes of references between objects. Their contribution shows how they have integrated the change notifications into ORION and show what overhead this can incur.

We showed in this thesis how we defined an abstraction for the change notifications so that change notifications can be implemented across different data-systems, like databases or files. This is different from the mentioned works. These explain how to define change notifications specifically for one database system. These contributions can help to apply our concept of change notifications to the different databases, since one still has to implement the specific functionality to make change notifications work for a specific type of shared data source. When one has a database system that still needs to support change notifications, and not just following our abstraction, the work that has been researched for ORION can give a good insight in how this is achieved and what problems might occur.

The solution for change notifications that we have shown in this thesis is similar to a messaging-system called publish-subscribe. There are many contributions about this technique [17][18][19]. It is used often in distributed systems where multiple systems want to know when new information is published. Therefore protocols have been described to implement a publish-subscribe system across large networks. In [20] it is described how to efficiently send notifications to multiple subscribed systems using multi-cast. This method can be used to efficiently publish notifications to a huge amount of subscribers from a central server. In [21] a publish-subscribe protocol is described that uses a peer-to-peer approach. This makes such a protocol suitable for networks of systems that have no central server and are self-organizing. This increases the stability of such a network, since there is no single point of failure.

These contributions describe network protocols to distribute messages in a networked publish-subscribe system. This thesis describes how to handle change notifications, that can be described as messages, on the side of the consumer. The contributions could be used for implementing such messaging-systems over the network. Using our abstraction these systems could be used for change notifications the same way as any other change notification, independent of its source.

Apart from these network protocols, there is also work on how to handle publish-subscribe mechanisms in general. In [22] filtering method is described. This method can be used to filter messages in publish-subscribe systems efficiently. In this thesis we showed how to subscribe for changes of any data in data sources. When however we are only interested in parts of the data, like a section of a text-file, it would be more efficient to only be notified of changes we are interested in. In such case efficient filtering of notifications is needed. This work on filtering messages can be helpful to give insight into such problems.

A more general view on publish-subscribe mechanisms is given in [23]. It describes and classifies the many different variants of publish-subscribe mechanisms. Each of these variants have different benefits and shortcomings, in terms of their interfaces as well as their implementations. In our thesis we described a number of solutions to handle change notifications in our systems and have chosen a solution that turns out to be publish-subscribe. Future usage of the system will show whether there need to be more requirements, in which case investigation of different publish-subscribe systems may be needed.

These contributions give information about optimizing and categorizing publish-subscribe messaging systems. In this thesis we described how a messaging system could be used to send change notifications safely, but also showed the problems that we have found. When one wants to learn about improving the messaging system for change notifications, these contributions may be of help.

Linda [24] is a language which is used to model parallel systems. The language aims to simplify problems that occur when constructing highly concurrent systems. This includes multi-threaded programs, where components need to act among each other using multiple threads or processors, but it can also be used for systems that communicate via a network. Linda is also referred to as a coordination language, as it is responsible for coordination and communication between the systems. Because it is such a domain specific language, it can be embedded in other languages, like Java or C, where it uses a few simple operations to more easily express how the program should communicate in a safe manner.

Linda can be used to better model how our described system works. Describing the communication between the different threads within the program could give better insight in the workings of the system. It might reveal other solutions to the problems that we have shown. Apart for using it directly for the internal system, it can also be used to describe the change notifications over many different systems that need to communicate the changes over a network. This could be useful when systems need to be build that handle data from different external systems.

Chapter 7

Conclusion

In this thesis we have shown a number of ways how one can detect and react to real-time changes of shared data sources in a system like iTasks. We have split this problem into two parts. First the detection and notification of changes and second how to use change notifications to evaluate tasks reliably without losing information.

For change detection and notification we have found and shown many different methods. These include polling, notifying on writes, various ways of waiting, notifying by pushing callbacks onto a queue or notifying by pushing messages onto a queue. Each of these methods has different advantages and disadvantages. We chose to use notifying by pushing callbacks onto a queue for our extension of the iTasks system.

We have shown how we used the notification system to know when to evaluate tasks and how to evaluate tasks in a system like iTasks. A way was needed to know for what shared data sources change notifications should be received and also how to handle these change notifications. We have shown how we have solved these problems in our system.

Using the knowledge gathered from the change notification system and the ways to know when and how to evaluate tasks, we have created a simulation that is a simplified version of the iTasks system, but also includes the different aspects which we use to react to changes in real-time. We have shown how we defined the simulation, touched on how it works and have shown the results that can we gathered from using the simulation.

This all should give a good indication of what problems need to be solved and some of the solutions that we have found to be working when creating a real-time task oriented framework that can react to changes of shared data sources in a pure functional language.

Bibliography

[1] Ian Hickson, Google, Inc., "The WebSocket API," http://dev.w3.org/html5/websockets/.

[2] ——, "Server-Side Events," http://dev.w3.org/html5/eventsource/.

[3] E. Foundation, "Etherpad Foundation," http://etherpad.org/.

[4] S. Michels and R. Plasmeijer, "Multi-purpose shared data sources in a functional language," http://wiki.clean.cs.ru.nl/images/3/3c/Sharing_Data_Sources.pdf.

[5] R. Plasmeijer, P. Achten, and P. Koopman, "An introduction to itasks: defining interactive work flows for the web," *Central European Functional Programming School*, pp. 1–40, 2008.

[6] T. Brus, M. van Eekelen, M. van Leer, and M. Plasmeijer, "Clean—a language for functional graph rewriting," in *Functional Programming Languages and Computer Architecture*. Springer, 1987, pp. 364–384.

[7] T. Butler, "Computer response time and user performance." in *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*. ACM, 1983, pp. 58–62.

[8] Microsoft, "SetEvent function," http://msdn.microsoft.com/en-us/library/windows/desktop/ms686211.aspx.

[9] ——, "WaitForSingleObject function," http://msdn.microsoft.com/en-us/library/windows/desktop/ms687032.aspx.

[10] ——, "WaitForMultipleObjects function," http://msdn.microsoft.com/en-us/library/windows/desktop/ms687025.aspx.

[11] ——, "RegisterWaitForSingleObject function," http://msdn.microsoft.com/en-us/library/windows/desktop/ms685061.aspx.

[12] ——, "Thread Pool API," http://msdn.microsoft.com/en-us/library/windows/desktop/ms686766.aspx.

[13] ——, "Using Query Notifications," http://msdn.microsoft.com/en-us/library/ms175110.aspx.

[14] J. L. J. Chris Anderson and N. Slater., "Continuous Changes," http://guide.couchdb.org/draft/notifications.html.

[15] P. G. D. Group, "PostgreSQL: Documentation: Manuals: NOTIFY," http://www.postgresql.org/docs/8.1/static/sql-notify.html.

[16] H. Chou and W. Kim, "Versions and change notification in an object-oriented database system," in *Proceedings of the 25th ACM/IEEE design automation conference*. IEEE Computer Society Press, 1988, pp. 275–281.

[17] A. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. White, "Towards expressive publish/subscribe systems," *Advances in Database Technology-EDBT 2006*, pp. 627–644, 2006.

[18] A. Gupta, O. Sahin, D. Agrawal, and A. Abbadi, "Meghdoot: content-based publish/subscribe over p2p networks," in *Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*. Springer-Verlag New York, Inc., 2004, pp. 254–273.

[19] Y. Huang and H. Garcia-Molina, "Publish/subscribe in a mobile environment," *Wireless Networks*, vol. 10, no. 6, pp. 643–652, 2004.

[20] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. Strom, and D. Sturman, "An efficient multicast protocol for content-based publish-subscribe systems," in *Distributed Computing Systems, 1999. Proceedings. 19th IEEE International Conference on*. IEEE, 1999, pp. 262–272.

[21] W. Terpstra, S. Behnel, L. Fiege, A. Zeidler, and A. Buchmann, "A peer-to-peer approach to content-based publish/subscribe," in *Proceedings of the 2nd international workshop on Distributed event-based systems*. ACM, 2003, pp. 1–8.

[22] F. Fabret, H. Jacobsen, F. Llirbat, J. Pereira, K. Ross, and D. Shasha, "Filtering algorithms and implementation for very fast publish/subscribe systems," in *ACM SIGMOD Record*, vol. 30, no. 2. ACM, 2001, pp. 115–126.

[23] P. Eugster, P. Felber, R. Guerraoui, and A. Kermarrec, "The many faces of publish/subscribe," *ACM Computing Surveys (CSUR)*, vol. 35, no. 2, pp. 114–131, 2003.

[24] D. Gelernter and N. Carriero, "Coordination languages and their significance," *Communications of the ACM*, vol. 35, no. 2, pp. 97–107, 1992.