A Distributed Server Architecture for Task Oriented Programming

Master Thesis

Arjan Oortgiese

Supervisors:
prof.dr.dr.h.c.ir. M.J. Plasmeijer
dr. B. Lijnse

Nijmegen, January 2017
Contents

1 Paper 4

2 TOP / iTasks in a Nutshell 17
   2.1 Loops . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17

3 Enabling Distributed Evaluation of Clean 18

4 Distributed iTasks 20
   4.1 Distributed Domain Controllers . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
   4.1.1 Domain Controller . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
   4.1.2 Local controllers . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
   4.2 Accessing Remote Shared Data Sources . . . . . . . . . . . . . . . . . . . . . . . 20
   4.2.1 Get a value from a Remote Shared Data Source . . . . . . . . . . . . . . . 21
   4.2.2 Observing a remote shared . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
   4.3 Evaluating Tasks Remotely . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
   4.4 Local controllers . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
   4.4.1 Running iTask as a library . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
   4.4.2 Reconnecting . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
   4.4.3 Android app . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27

5 Case study: Service engineer 28
   5.1 Service engineer status . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
   5.2 Creating a ticket . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
   5.3 Getting the GPS location of an address . . . . . . . . . . . . . . . . . . . . . . . 30
   5.4 Obtaining a route . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
   5.5 Get engineers travel distance and time from address . . . . . . . . . . . . . . . . 32
   5.6 Selecting engineer . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
   5.7 Working on a ticket . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
   5.8 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37

6 How to use the distributed version 38
   6.1 Docker . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
   6.2 MacOS . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39

7 Conclusion 40

Bibliography 42
Preface

First of all I would like to thank my supervisor Rinus Plasmeijer for all the discussions and feedback during the research internship and the master thesis process. He encouraged me to write an extended abstract for IFL 2016 in Leuven. He also supported the process of giving a presentation for a conference and writing a paper during my master thesis.

The first time I wrote a paper was very hard because I learned about paper writing but I never actually wrote one. Giving a presentation at IFL 2016 and writing a paper was a great learning experience for me.

I also want to thank John van Groningen for supporting me to build the first applications with the Clean compiler for ARM, the support and help by writing the paper. Bas Lijnse and Tim Steenooorden have also supported me with their valuable comments on the draft paper and IFL presentation.

During my research internship I made it possible to run iTask on an Android device using the ARM support in the Clean compiler. With this support, we developed a way to send tasks to another device and evaluate these tasks there. In the last phase of the internship and during the summer break we wrote an extended abstract and submitted this to IFL 2016 in Leuven (https://dtai.cs.kuleuven.be/events/ifl2016/). This extended abstract was accepted for a presentation.

When one has given a presentation at IFL, one can submit a full paper for the post-symposium proceedings. I had two goals for my master thesis project:

1. Write a full paper that we can submit for the IFL 2016 post-symposium proceedings.

2. Improve the result from the research internship

This master thesis is an extended description of the paper that is submitted for the IFL 2016 post-symposium proceedings.
Introduction

The iTasks framework offers a special flavour of functional programming, called Task-Oriented Programming (TOP) \[8\]. TOP focuses on an important application domain: multi-user, web-based applications. It is in particular meant for people working distributively, but nevertheless need to closely collaborate together to accomplish a certain goal.

The basic idea behind TOP is that a programmer of a complicated, collaborating application should be able to focus on two things: the description of the tasks that need to be done, and how these tasks influence each other. The programmer should however not be worried about the technical details to realise the application. TOP offers such a very high-level, abstract, declarative style of programming.

In TOP, tasks form the main abstraction for defining applications. The starting point of an iTask application is a task which is defined in terms of smaller (simpler) sub-tasks by using task combinators. Tasks can be assigned to a specific user or any user with a certain role. Users can work on tasks assigned to them in any order, very similar to the way somebody can work in any order on the emails they have received. An important difference, compared to answering an email, is that if somebody is working on a task and updates some information, it can directly influence the content of tasks others are working on. It behaves like a distributed spreadsheet several people are working on at the same time.

The current iTasks implementation works with any modern browser. That means that it works on mobile devices such as tablets or smartphones. Like many other web applications depending on a central server using a browser as a client, the current architecture also has some drawbacks:

1. The central server may become a bottleneck when too many clients need to be served.

2. When the connection with the central server is lost, somebody will not be able to work anymore: the client cannot inform the server about the progress tasks make, while the server cannot inform the client about the progress made by others. The end-user has to wait until the browser is reconnected to the internet.

3. To realise complex tasks in the browser, we have to use JavaScript. Although we compile Clean functions to state-of-the-art JavaScript code, it executes about ten times slower than native Clean code. For certain CPU intensive applications, this can be unacceptably slow.

4. The complexity of any computations one can do in a browser is limited as well, due to JavaScript stack size limitations by default imposed by browsers to avoid unresponsive behaviour.

5. For security reasons, browsers furthermore do not allow access to operating system facilities, the file system, and many sensors which can be found on modern devices such as smartphones and tablets.

The last three drawbacks are actually general problems for any web application and are therefore not iTasks specific. Techniques that can be used to solve these problems are e.g. using Java-applets, Silverlight, or browser extensions. These techniques have drawbacks as well: not all browsers support them because they are battery hungry or have security issues. Furthermore, some, like Java-applets \[6\], depend on technology that will be removed from some browsers. This means that JavaScript currently is the way to create web applications because it is supported by
INTRODUCTION

all major browsers. We, therefore, want to keep the support of JavaScript but also want to be able to offer a faster alternative.

In this master thesis, we present an extension of the current architecture. Instead of a central iTask server we can have an arbitrary number of them, which are called controllers from now on. Controllers can act like an old fashion iTask server, but can also be connected as client to some other controller. Controllers can run distributively and efficiently in native code, on any processor, both on server devices as well as on clients devices. They are used as follows:

- To avoid the bottleneck of a central server (issue 1.), any controller should be able to delegate the handling of a task to another controller on some other CPU of a serving device. This is not trivial, because tasks can influence each other, and they now may be administrated on different machines.

- To reduce the dependency on the availability of a global server (issue 2.), it would help if one could execute controllers not only on a server but also on a client. Servers and clients may use different platforms: we need platform independent controllers.

- If someone can have controllers on a client, and someone is able to execute native code on the client, e.g. in an App, someone can largely bypass JavaScript (issue 3. and 4.), and have access to all the facilities offered by the local operating system and local hardware as well (issue 5.). When the app would also include a browser interface, e.g. by including a library like WebView [2], we can have the best of both worlds.

This master thesis has a different structure than expected because during my master we wrote a paper and submitted this for the IFL 2016 post-symposium proceedings. IFL 2016 is the 28th Symposium on Implementation and Application of Functional Languages. The goal of the IFL symposium is to bring together researchers actively engaged in the implementation and application of functional and function-based programming languages [3].

This master thesis is structured as follows: The paper submitted for the IFL 2016 post-symposium proceedings is included in chapter 1. Additional information that was not included in the paper because of the page limit is explained in chapter 2 until 4. In chapter 5 we present a case study with the distributed version of iTasks. In chapter 6 we explain how someone can compile an iTasks application for different platforms.
Chapter 1

Paper

We included the paper that is submitted to the post-symposium proceedings of IFL 2016. There are three authors, where I am the main author. John van Groningen did most of the work on the graph-copy module that allows to evaluate closures on another machine. I changed only a few small bits in the module that are explained in chapter 3. John van Groningen also made the ARM code generator (e.g. Clean for ARM). Rinus Plasmeijer is my supervisor.
A Distributed Server Architecture for Task Oriented Programming

Arjan Oortgiese    John van Groningen    Rinus Plasmeijer

Abstract
The iTasks framework is a Clean library for developing multi-user, web-enabled applications. It offers a special flavor of functional programming, Task-Oriented Programming (TOP), where the notion of tasks play the central role. In iTasks one specifies the tasks, that end-users and systems have to do to accomplish a certain goal. From such a specification a central web-server is generated which coordinates the work described. To provide a specific end-user client with a user-interface for doing the task, web-pages are dynamically generated for the tasks to be done, that can be inspected by any HTML 5 compatible browser. In this way, any PC, tablet or phone can be used by the end-user to do his or her work.

The current architecture of iTasks is a client-server architecture that is easy to maintain, but it also has certain disadvantages. The use of one centralized server can become a bottleneck when it has to serve too many clients. Furthermore, one cannot work offline on tasks when the connection with this central server is lost: the server constantly needs to administrate the progress clients make. A feature of the iTask system is that arbitrary complex browser applications can be defined as client. For this purpose Clean functions are compiled to JavaScript. But, certainly compared to native Clean code, browsers execute JavaScript code extremely slow, while only relatively small browser applications can be generated, due to stack size limitations commonly imposed by the standard browsers. Finally, due to security restrictions, not all resources offered by a platform can be accessed in a browser, such as the file system or certain hardware like the Bluetooth connection on a tablet or smart phone.

In this paper, we present a solution to address these drawbacks. In the new iTask architecture one can have an arbitrary topology of distributed iTask controllers. An iTask controller can act as a server like in the old iTask architecture, as well as a client of some other controller. The coordination of tasks can be distributed over these controllers, e.g. to decrease the task load of a serving device or to work off-line on a task on a client device. A first implementation is made that works for Intel and ARM processors running MacOS, Linux, or Windows. Android Apps can be generated that include a local server, include a browser facility, and have access any resource available on the platform.

We believe our solution is of general interest for anyone who is interested in generating distributed multi-platform applications from one single source code.

Keywords  Functional Programming; Pure Functional Languages; Web Programming; Distributed Applications; Task-Oriented Programming; Client-Server Architecture; Platform Independent Code Generation; Clean; Haskell

1. Introduction
The iTasks framework offers a special flavor of functional programming, called Task-Oriented Programming (TOP) [15]. TOP focuses on an important application domain: multi-user, web-based applications. It is specially meant for people working distributively, but nevertheless need to closely collaborate together to accomplish a certain goal. A typical example is the Search and Rescue actions that need to be performed by the coast guard [10]. Here different teams closely need to work together to rescue people, and therefore everybody constantly needs to be well informed about the latest state of affairs of all teams involved in the rescue action.

The basic idea behind TOP is that a programmer of a complicated, collaborating application should be able to focus on two things: the description of the tasks that need to be done, and how these tasks influence each other. She should however not be worried about the technical details to realise the application. TOP offers such a very high-level, abstract, declarative style of programming. In TOP, tasks form the main abstraction for defining applications. The the starting point of an iTask application is a task which is defined in terms of smaller (simpler) sub-tasks by using task combinators. Tasks can be assigned to a specific user or any user with a certain role. Users can work on tasks assigned to them in any order, very similar to the way one can work in any order on the emails one received. An important difference, compared to answering an email, is that if somebody is working on a task and updates some information, it can directly influence the content of tasks others are working on. It behaves a like a distributed spreadsheet several people are working on at the same time.

This reactive behaviour is realized as follows. From the tasks being described, a central web-server is generated which coordinates the work assigned to people and systems. An end-user can login into the system to see which tasks he can work on (the task list) and which of these task he is already working on (instantiated tasks). When an end-user is working on one of these tasks, the progress made and the latest state information of the task is constantly communicated to the server in an event stream. The server determines the consequences of the event, not just for this task but also for all other users depending on the changed task. These consequences may be that values have changed, views need to be updated, options have changed and new tasks are generated, while
others may get obsolete and are removed from the task list. The screen of every user logged in will be updated automatically.

The current iTasks implementation works with any modern browser and is thus works on mobile devices such as tablets or smartphones. However, like many other web applications depending on a central server using a browser as a client, the current architecture also has some drawbacks:

1. The central server may become a bottleneck when too many clients need to be served.
2. When the connection with the central server is lost, one cannot work any more: the client cannot inform the server about the progress tasks make, while the server cannot inform the client about the progress made by others. The end-user has to wait until the browser is reconnected to the internet.
3. To realize complex tasks in the browser, we have to use JavaScript. Although we compile Clean functions to state-of-the-art JavaScript code, it executes about ten times slower than native Clean code. For certain CPU intensive applications this can be unacceptably slow.
4. The complexity of any computations one can do in a browser is limited as well, due to JavaScript stack-size limitations by default imposed by browsers to avoid unresponsive behavior.
5. For security reasons, browsers furthermore do not allow access to operating system facilities, the file system, and many sensors which can be found on modern devices such as smartphones and tablets.

The last three drawbacks are actually general problems for any web application, and are therefore not iTasks specific. Techniques that can be used to solve these problems are e.g. using Java-applets, Silverlight, or browser extensions. These techniques have drawbacks as well: not all browsers support them because they are battery hungry or have security issues. Furthermore, some, like Java-applets [13], depend on technology that will be removed from some browsers. This means that JavaScript currently is the way to create web applications because it is supported by all major browsers. We therefore want to keep the support of JavaScript but also want to be able to offer a faster alternative.

In this paper we present an extension of the current architecture. Instead of a central iTask server we can have an arbitrary number of them, which are called controllers from now on. Controllers can act like an old fashion iTask server, but can also be connected as client to some other controller. Controllers can run distributively and efficiently in native code, on any processor, both on server devices as well as on clients devices. They are used as follows:

- To avoid the bottleneck of a central server (issue 1.), any controller should be able to delegate the handling of a task to another controller on some other CPU of a serving device. This is not trivial, because tasks can influence each other, and they now may be administrated on different machines.
- To reduce the dependency on the availability of a global server (issue 2.), it would help if one could execute controllers not only on a server, but also on a client. Servers and clients may use different platforms: we need platform independent controllers.
- If one can have controllers on a client, and one is able to execute native code on the client, e.g. in an App, one can largely bypass JavaScript (issue 3. and 4.), and have access to all the facilities offered by the local operating system and local hardware as well (issue 5.). When an App would also include a browser interface, e.g. by including a library like WebView [6], we can have the best of both worlds.

This paper is organized as follows: First we give a short overview of the most important concepts of the TOP style in Section 2, namely Tasks and Shared Data Sources. iTasks is an implementation of TOP and we briefly explain the current iTask architecture with one central server. For our new implementation we have to be able to run a Clean application distributively over different platforms. Until now, we compiled the same Clean source code twice: to native Intel code to run on the server, and to JavaScript code to be interpreted by the browsers on the clients. To create controllers which can run efficiently on clients as well, such as tablets and smartphones, we now also need to generate code for ARM processors. Furthermore at run-time we have to be able to send any closure to be evaluated remotely from one platform to any other. How this is achieved is explained in Section 3. The ability to run Clean functions distributively across different platforms is necessary, but not sufficient for realizing a distributed version of the iTask system. We have to be able to run any task on a remote processor, while its result can be observed elsewhere. Tasks can access Shared Data Sources which can now be located on another computer. How we support this, and how controllers can be set-up distributively is explained in Section 4.

Finally, related work is discussed in Section 5, and conclusions are drawn and future work is discussed in Section 6.

2. TOP / iTasks in a Nutshell
There are many papers on iTasks (e.g. [15]). For people not familiar with the concept we give a short overview here.

The iTasks framework can be regarded as a reference implementation of a special flavor of functional programming, namely Task Oriented Programming (TOP). Therefore TOP and iTasks are often used as synonyms. TOP extends the classical way of functional programming by offering an Embedded Domain Specific Language (EDSL) with Clean as host language. TOP is focusing on the development of multi-user web-based applications. Notice that the iTask framework is actually “just” a plain Clean library, no change in the standard syntax and semantics of Clean has been made.

In this Section we first introduce the core concepts of iTasks as EDSL. Next we give an example of an iTasks specification, showing how a chat application between two end-users can be defined. Hereafter we explain the major components of the current iTask run-time system before we explain what is needed to turn this architecture in a distributed one.

2.1 Core Concepts of TOP / iTasks
The most important constructs offered by the iTasks framework are:

2.1.1 Tasks
A task is “just” an ordinary Clean function returning a value of type :: Task a for some concrete type :: a. The type :: a can be any Clean type, under the context restriction that an instance of class (iTask a) exists for a. This iTask class consists of a number of generic functions [9] which are needed to evaluate tasks on servers and clients, such as functions to serialize, de-serialize, compare values, and create GUI’s. Fortunately all these generic functions can be automatically derived for any concrete first order type. So, the context restriction is not a real restriction in practice.

Tasks are ordinary Clean functions, but they are defined in a special recursive way [9]. As a result, a task of type :: Task a delivers an observable value of type :: a, which value can change over time. Such a change can occur when an end-user works on a task in a browser and changes something. So, the task value reflects the current state of the work taken place. The state of a task can be empty (nothing), when no work has been done yet, or when it is decided to start all over from scratch. It can be an unstable value
2.1.2 Editors
An editor is an overloaded basic task, returning a value of \(\text{:: Task}\ a\) which can be created for any concrete first order type \(a\). Special about editors is that, by using type driven generic functions [9], they can be used to automatically generate a user interface in the browser with which only values of that concrete type \(a\) can be constructed (see [9]). Examples of such editors are: \(\text{viewInformation}\) to display a value of type \(a\), and \(\text{enterInformation}\) to enter a value of type \(a\).

2.1.3 Task Combinators
With Task Combinators, tasks can be defined in terms of other tasks. There are \(\text{parallel}\) combinators and \(\text{sequential}\) ones. E.g. with the parallel and \(\&\&\&\) operator, two tasks \(ta\) and \(tb\) can be started in parallel, in the resulting task returns a tuple combining the results of both tasks, so \(\text{\{ta \&\&\& tb :: Task\(a\),\(b\)\}\}}\). With the parallel or \(\text{||}\) operator, the value of the task which has become stable first is returned, so \(\text{\{ta || ta2 :: Task\(a\)\}\}}\). Frequently used variants of this operator are \(\text{||}\) (only return the current value of the first task, ignore the second), and \(\text{||}\) (the other way around).

The step operator is a sequential choice combinator denoted as \(>>\). In \(ta >> [\text{action}_1\ p_1\ \text{atb}_1,\ldots,\text{action}_n\ p_n\ \text{atb}_n] ::\) \(\text{Task\(a\)}\) \(\text{\{b\}}\) the task \(ta\) :: \(\text{Task\(a\)}\) is being observed. The actions in the list are converted to buttons on the screen. When button \(\text{Action}_i\) is pressed by the end-user and predicate \(p_i\) holds, where \(v\) is the current value of task \(ta\), the task \(ta\) is ended and one continues with task \(\text{atb}_i\ v\) :: \(\text{Task\(b\)}\). There are also simple variants of this sequential operator, like a monadic bind \(>>\.). Also other monadic style operators like \(>>\) (Haskell: \(>>\)), and return have been defined.

One-and-the-same end-user may work on several parallel tasks at the same time. But an important aspect of a parallel task is that it can be assigned to someone else. In \(\text{Bob @: sometask}\) the task \(\text{sometask}\) is assigned to user \(\text{Bob}\). In this way an arbitrary number of tasks can be assigned to someone. In the default set-up, the tasks somebody can work on are listed in a to-do list, and the end-user can freely choose on which tasks he or she will work.

The iTask library offers a rich set of combinators, build on top of only two swiss-army knifes combinators: one for the parallel case and one for sequencing (see [14]).

2.1.4 Shared Data Sources
Although tasks can observe each others progress, and information can be passed from one task to another, tasks also need to have access to shared data. For any type of data that can be shared, one abstract interface is made, called a \(\text{Shared Data Source or SDS}\) [11]. There are different types of shares e.g: file shares where shared data is stored in a file, database shares where the shared data is stored in a relational database, and memory shares where the data is temporarily stored in main memory. The current date and time is also treated as a (read-only) share. Shares are used internally to administrate the users who can login, the current user working on a task, and so on. In the current iTask architecture, \(\text{SDSes}\) are only available on the iTasks server and can not be put on a client.

On shares the following operators are defined: \(\text{get, upd, set and watch}\). The \(\text{get}\) operation retrieves a value from the share. The \(\text{upd}\) updates the shared value using a lambda expression. \(\text{set}\) writes a new value to the share without considering the old value. \(\text{watch}\) allows the share to be observable using, for example, the step combinator. One can also use shares in editors. When one want to view the content of a share one can use \(\text{viewSharedInformation}\). The content can be updated with \(\text{updateSharedInformation}\) where the share is updated with every change that is made by the end-user in the client.

An important aspects of shares is the following. Whenever someone changes a share, all tasks which are currently looking at it are automatically informed about the change made. For example, if an end-user changes the content of a share via \(\text{updateSharedInformation}\), all other users who look at the value via \(\text{viewSharedInformation}\) will see the changed value as well, albeit a bit later due to the latency of the internet. In this way shares behave like a kind of distributed spreadsheet. Every share is at run-time administrated in the public-subscribe system of iTasks where it is managed which tasks need to be informed when something is changing.

2.2 A Small iTask Example: Chat
As a running example we present an iTask application that enables two people to chat with each other using a browser. We define it in a general way so that we can easily modify it later in this paper. It is not important to understand all details, we want to give you an idea about what task definitions look like, and use it later on to explain what the consequences are for a distributed architecture.

In Figure 1 we show the task \(\text{createDuoChat}\) that takes a task as argument. This argument of type \(\text{User -> Task\(\text{Maybe a}\)}\) can be any task that enables an end-user to enter some information. The task \(\text{createDuoChat}\) first determines the name of current end-user who wants to chat with someone, \(\text{me, line (3)}\). Next, using the bind combinator \(>>\), it will ask this user to select someone else to chat with \(\text{4, you}\). The predefined SDS users from which someone is chosen, contains a list of all users currently known in the system. Hereafter, a new SDS is created in shared memory and initialised with an empty list \(\text{[]}\) (5). This list will be used to memorize the chats between the two end-users. This shared list is passed to the \(\text{duoChat}\) task with which we continue.

```haskell
createDuoChat :: (User -> Task (Maybe a)) -> Task a |
createDuoChat myChat |
  get currentUserId |
  >> \me -> enterChoiceWithShared "Chat with" \[\] users |
  >> \you -> withShared \[\] (duoChat me you myChat)
```

Figure 1. Select a user, and create an SDS for the chat history.

Figure 2 shows the \(\text{duoChat}\) task taking two users, a task to enter a chat, and a share to memorize the chats, as argument. The function is overloaded. For the chat type \(\text{a}\) and users \(\text{u}\) the iTask class is needed. The task \(\text{duoChat}\) creates two symmetrical \(\text{chatWith}\) tasks in parallel \(\text{(-|-)}\), one assigned \((\&\&\&\&)\) to me \(\text{(4)}\), the other assigned to you \(\text{(5)}\). When the chat session is finished by one of the end-users, the chats memorized in the SDS share are returned.

```haskell
duoChat :: u u \(\text{u -> Task (Maybe a)}\) (Shared \(\text{[\]}\)) -> Task \(\text{a}\)|
  | iTask a & toString u |
  | me you myChat share |
  | = (me 0: chatWith me you myChat share) |
  | -|- (you 0: chatWith me you myChat share) |
  | \| get share
```

Figure 2. Start two chats\(\text{With}\) tasks in parallel.
Figure 3. The task that is handling the storage of the chats

So, both end-users can work on the task chatWith (Figure 3 in parallel. Each user will see the chats entered so far, thanks to viewSharedInformation (4). At the same time (||-), they can enter a new chat (5). The enterChats task uses the higher order function myChat that is given as argument to chatWith to let the user enter a new chat message. This function might return nothing to indicate that the chat has ended, or a new chat value Just new (11). This new chat value is stored in the shared data source using the upd operation, after which the next chat can be created (12). When the share is updated, both parties will see the new chat due to the viewSharedInformation on this share.

Figure 4. Creating a chat that supports sending text messages

Finally we need to define some task to enter a chat. There are many options. When we store a chat, we also want to know who said something at what time. For this purpose we introduce the polymorphic record type :: Msg a in Figure 4 (1). The task enterChat is still very general, it can be used to enter a message of arbitrary type (5). If such a message is entered, the end-user can decide to Send this message (8). The task then calls newMsg (12) which determines the current time (14), and returns the message, packed into the record with user name and current time. The end-user may always decide to Stop (9), in which case Nothing is returned.

Finally, lets assume that we want to have a simple chat where end-users can enter a text string. For this purpose we have defined the task chatString (19). It simply calls createDuoChat with enterChat as argument. Notice that the kind of messages that can be typed in is determined by the type of chatString only. We can choose any first order type here to change the type of messages that can be typed in.

Figure 5. Screenshot of the simple chat application where Alice and Bob are chatting.

Figure 6. Central iTasks server with arbitrary clients (browsers)
creation date and time, priority, deadline, the creator of the task and for whom the task is created for. The created for attribute can be a specific user, or someone with a specific role.

- **Task instance administration.** A task instance is an instance of a task administrated in the task pool someone is actually working on. The relation between the task pool and instance pool is one to at most one. Only one person can work on a certain task at the same time. Tasks may switch from owner though (e.g. someone with the same role) at any time. Local status information, stored at the client, may get lost when someone takes over a task from someone else.

- **The Shared Data Source administration.** Contains tasks are currently depending (subscribed) on which SDS’s. This is used to inform those tasks when an SDS is changed by someone.

- **Web server.** The iTasks system includes a build-in web server, although any standard web server can be used as well. Every action on a browser is propagated to the server as an event and handled by the iTasks kernel. The kernel continuously processes events from clients, tasks and shared data sources, TCP connections and the like to coordinate the tasks.

The iTasks system contains several pluggable components as extension, to reduce the number of core components to a bare minimum. The authentication process of users and its administration is an example of such an extension. Also the way tasks are presented in a browser to end-users can be freely defined in a task. A standard way to do this is to present all tasks someone can work on in a dedicated to-do list, such that the end-user can choose where to work on by opening one or more of these tasks.

2.3.2 iTasks Clients

The clients in iTasks are HTML 5 compatible browsers. It initially consists of a small client-side application written in JavaScript. It sends events that happen on the client, such as key input or mouse actions, to the iTasks server and processes its responses. It also accepts new JavaScript code that is just-in-time compiled from Clean code when it is needed for the evaluation of a specific closure. It allows programmers to define custom editors in Clean (see [4]), e.g. for displaying a map or for using JavaScript WebAPIs like getting the current location or access the camera.

2.3.3 Limitations of the Standard Run-Time System

The standard client-server iTasks architecture works fine, but has some important limitations, as already has been summarized in the introduction. The use of one central server will cause efficiency problems when there are too many clients. When the server is down, no one can work any more. Clients cannot inform the server about the progress of the tasks being executed. The server cannot communicate the latest state of the SDS’s to the relevant clients. Due to the use of JavaScript, clients are relatively inefficient and cannot perform complex computations. Also access to client-side resources such as file systems, Bluetooth devices, smart cards, are not possible to avoid security issues. We can solve these problems if we could have an arbitrary number of distributed iTasks servers, called controllers, running on any platform.

3. Enabling Distributed Evaluation of Clean

To realize a distributed version of iTasks we first need to enable distributed evaluation of Clean on different platforms. Therefore we have extended the Clean compiler to generate code for ARM processors as well. So, when compiling Clean for our new distributed architecture, we generate:

1. Native (Intel) x86-64 code for Windows, Mac, and Linux platforms;
2. JavaScript code for HTML 5 compatible browsers;
3. And now also native code for ARM platforms such as Raspberry Pi and Android.

With these code generators Clean applications can run on most popular platforms. To realize distributed evaluation of a Clean application, two additional facilities are needed. First of all, the corresponding code needs to be available on the remote processors in the network. Secondly, we have to be able to ship a closure that needs to be evaluated to the intended remote processor, and be able to return its result after evaluation.

3.1 Distributing Code

If we ship a function for evaluation to another machine, we have to ensure that the corresponding code needed for the evaluation of that particular function is indeed available on the remote machine.

Clean has a virtual machine, the ABC machine. One option would be to ship virtual machine code and let the remote machine perform the last phase of the compilation for its platform. Instead we have decided to generate all code at once when the application is being developed, so that is ensured that the required code is available. All code for all platforms is generated from the same Clean source code. Fortunately, because the Clean compiler is very fast, the developer is not hampered by this. To give an idea: the Clean compiler, which is not a small application, can compile itself from scratch in 11 seconds, on a smart phone like the Samsung Galaxy Note 4. The last phase, code generation (ABC to native code), even takes less time. Code is generated for the target platforms: an ARM version for Android tablets, an x86-64 for computers having an Intel CPU, and SAPL. SAPL [7] is a very simple functional language that is dynamically (at run-time) just-in-time translated to JavaScript code. SAPL has as advantage that from the code one can relatively easy determine which functions are needed for the evaluation of a given closure.

Next we have to ensure that the required code is available on the (remote) machine before we ship a closure for evaluation to it. The code can be distributed in different ways:

1. eagerly, i.e. all code, the complete image, has to be stored on the target machine in advance.
2. lazily, i.e. only that part of the code which is needed for the evaluation of the closure in question is shipped just-in-time to the target machine.

The eager method has as advantage that the code only has to be shipped once, while with that image any possible closure received can be evaluated. It has as disadvantage that a complete image can be huge, and perhaps one does not want to show all code on the remote machine for security reasons. The lazy method is more secure: only the code which is really needed for the evaluation of the closure is shipped. It has as disadvantage that for each closure one has to determine which code, not present yet, needs to be shipped. Clean is a lazy language, so one might need to ship additional code for unevaluated functions hidden in the closure. When several closures are send, one needs to administrate which code has already been send previously to reduce shipping overhead.

Currently, for x86-64 (Intel) and ARM processors the eager code distribution approach is used. One has to ensure manually that the image is indeed available on the remote machine. For browsers, JavaScript code is automatically shipped lazily to the browser using push technology.

In the future, we also want to support lazy shipping of code for all platforms where possible. Note that this implies that we need to be able to dynamically extend an application under execution.
For Windows we already have such a dynamic lazy linking facility which we can use to add plug-ins to a running application via Clean Dynamics [17]. For all other platforms and operating systems this facility is future work.

### 3.2 Distributing Functions

To send a closure for evaluation from one machine to another we need to be able to serialize any function at any time on any platform. The serialized function needs to have a platform independent representation such that it can be unpacked at any of the other platforms to be evaluated remotely. Of course, the result of the evaluation has to be send back in the same way as well.

Using native code has the drawback that serialization and de-serialization of closures and its resulting value becomes platform dependent. The run-time architecture on a platform may differ in all details: in code, address locations, data representations, stack lay-out, and heap lay-out. At run-time therefore a closure has to be reconstructed symbolically from the actual content of the stacks and the heaps. Closures furthermore can refer to data types and functions which are located at a different address in the image of the other platform. So, when reconstructing a closure we have to know where to find the information at run-time and replace concrete memory addresses by the corresponding symbolic function and constructor names. Since all code is generated from the same Clean source, these symbolic names are known in all code variants and can therefore be translated back to the proper run-time format needed for the evaluation of the closure.

The library function `graph_to_string_with_names :: a -> String` is able to reconstruct a closure symbolically and serialize it. It uses the descriptor name of the function or type that is known at runtime. Platform dependent de-serialization can be done with the library function `string_to_graph_with_names :: String -> a`. It uses the symbol table of the executable to look-up the memory addresses of the descriptor. We can obtain the symbolic names from the symbol table in the corresponding object/executable format. There are several object/executable formats we have to deal with, e.g. ELF for Linux (and Android), Mach-O for MacOS and PE/COFF for Windows. For every of these object formats we are able to obtain the required symbols.

When generating code, one can choose to generate code for 32-bit machines or for a 64-bit ones. We allow to mix these formats, but one has to be careful. An integer of course has a different size on a 64-bit machine than on a 32-bit one. We can translate the data of 32-bit machines to a 64-bit format and backwards as well. When an integer value become larger that 32 bits on a 64 bits machine, down scaling when creating a symbolic closure or its resulting value will yield incorrect numbers, and will cause a run-time exception. Furthermore, in this mixed setting we currently do not support the hardly used unboxed reals and unboxed arrays. This is future work.

### 4. Distributed iTasks

In this Section we explain the architecture of the new distributed iTasks system. Instead of having one central server, we now can have an arbitrary number of controllers, and an arbitrary network topology can be made which can change dynamically. All controllers are programmed in Clean, and can therefore run on different platforms, i.e. Intel and ARM processors. We assume that all devices on which controllers are running have all the necessary code to their disposal and are able to evaluate any closure shipped to them for evaluation (see Section 3).

In the new setting we have two types of controllers in the network, domain controllers, and local controllers. We first look at a topology where we only have distributed domain controllers (see Section 4.1) and explain what the consequences are for iTask applications in this setting and how the distribution of controllers affect the implementation. To be able to evaluate iTask tasks distributively, we have to deal with the fact that a Shared Data Source (see Section 4.2) can now be located on another device i.e. not at a centralised server. The evaluation of a task on another device is slightly more complicated than the evaluation of an ordinary function because the task value of a shipped task that is remotely being evaluated has to be made observable by the shipping controller. This is explained in Section 4.3. In Section 4.4 we show how one can reduce the workload of a serving device by splitting up the task coordination over several serving controllers. Next, in Section 4.5 we introduce local iTask controllers. One can use such a local controller e.g. on a tablet to work off-line. In Section 4.6 we show some examples of tasks running distributively on a network of controllers, domain controllers, and local controllers running as Android App, or located on a Raspberry Pi. The consequences of having a distributed architecture versus a centralized one is discussed in Section 4.7.

#### 4.1 Distributed Domain Controllers

Domain controllers for iTasks have things in common with domain servers for handling email traffic. The iTask set-up is much more complicated because working on a task can have direct effects for other tasks of users located elsewhere on the net, while answering an email has no global effect at all until the email is actually send. An iTask network has the following properties.

1. It is assumed that the locations of all domain controllers (e.g. their IP addresses) are globally known and accessible for all domain controllers in the net.  
2. For usability we also assume that all the domain controllers have a domain name that can be resolved using DNS. When DNS is not used the IP address is used as domain name for example: Alice@192.168.1.112.  
3. There has to be at least one domain controller in any iTask network. If the domain controller is the only one in the network, the architecture behaves just like the old standard configuration with one central server. The new architecture is a real extension of the old one.  
4. Each domain server is serving a group of users who are administrated in that domain and therefore can login into the server.  
5. The domain controller takes care of the authentication process. By default the current user and roles administration of iTasks is used. However, any other back-end such as an LDAP server or Single Sign On server can be used well. Any back-end server currently has to support two operations: Authorising, given a user name and password, receive a unique user identification, including the roles the user can fulfil. It can return a list of all currently administrated users, which can be used to (interactively) select a user to assign a task to. For implementing Single Sign On mechanisms we refer to [3].

#### 6. Tasks can be assigned to a user, or a user with a specific role, as usual. To address a user uniquely over the net, the domain name is added as postfix, just like we do in email addresses. So, a user is addressed by `user@domain_name`. If no domain is specified when a task is assigned to someone, by default the domain of the sending user is added as postfix.  
7. All tasks sent to a certain domain are administrated in the task pool administration of both the sending...
domain as well as in the administration of the receiving domain controller. When a user is logged-in into a domain controller, only relevant tasks can be seen.

Let us have a look at the very simple distributed network configuration of Figure 7. There are only two domain controllers, one for domain A and one for B. If Alice wants to chat with Bob who are both administrated in the same domain, the chat example from section 2.2 can be used unchanged and works as before.

```haskell
createDistChat :: (DomainUser -> Task (Maybe a)) -> Task [a] 
| iTask a 2
createDistChat myChat =
  get currentDistributedUser >>=
  v (me, myDomain) -> enterDomain
  >>= yourDomain -> usersOf yourDomain
  >>= enterChoice "Chat with" []
  >>= you -> withShared []
  (dusChat (me 0, myDomain))
  (you 0, yourDomain) myChat

| NoTask
| E.a: Remote_Task (Task a) TaskAttr InstanceNo & iTask a
| NoTask
```

Figure 8. Chatting with a user in another domain.

We can generalize the chat example such that we can communicate with any user administrated anywhere in the network as shown in Figure 8. We select a domain first, and next a user from that domain. All other code remains the same. Assume that this task would be used to let e.g. Alice on domain A talk with Dave on domain B, we already see interesting consequences. The SDS of type :: isOpen String is displayed to both chaters to show the communication that has taken place so far, will be located on the domain controller of Alice who started the chat. So, the task shipped to Dave running on controller B needs to have access to the SDS remotely located on domain controller A. Furthermore, Dave on controller B can decide to stop with the chat session, which has consequences for the the parallel task -|1| located on A (see Figure 2). So, one has to be able to observe task values of tasks being evaluated elsewhere.

4.2 Accessing Remote Shared Data Sources

In the original setting, all the Shared Data Sources were hosted on the same iTasks server. In the distributed setting, a task under evaluation on a certain controller, can create a new SDS which is located on that device. Let calls this device the SDS-host. Future tasks may want to have access to this SDS, also when such a task has been shipped to another controller for evaluation.

At run-time it is known on which device a specific SDS is located. When an SDS is located on the same device as the task accessing it, access can be handled as usual. To enable access to remotely located SDS’s, new task operators &r_get, &r_upd, &r_set and &r_watch are implemented which emulate remote SDS access as a proxy, in such a way that a remote SDS behaves and reacts in the same way as a local SDS. The programmer does not need to be aware where an SDS is located: at run-time an application of the get, upd, set, and watch operators will automatically be redirected to a call of &r_get, &r_upd, &r_set and &r_watch in the case that the SDS is located elsewhere.

To access a remote SDS we send a serialized closure (using graph_to_string_with_names) to the SDS-host. The host de-serializes the closure (using string_to_graph_with_names) and applies it to the SDS. This allows us to access any SDS no matter where and how it has been remotely created and stored. It even works for parametrized SDSes [11], SDS projections which allow a task to access a specific part of an SDS enabling a more fine grained and more efficient access to SDSes.

The result of the remotely applied closure is send back to the requesting controller who is waiting for the response. In this way calls to &r_get, &r_upd, and &r_set at a requesting controller can simply be realized by remotely applying the standard get, upd, and set at the SDS-host. This approach also ensures that the operators are applied in an indivisible way to the latest value of the SDS, and prevent us from having to deal with synchronization and version conflicts.

The handling of the &r_watch task is slightly more complicated because it cannot straightforwardly be translated to applying a watch on the SDS host. It would generate a lot of network traffic if we would implement this in the same way as the other SDS operators and unnecessarily block the requesting controller. Instead we implemented a notify_me request which will evaluate a closure when a given predicate holds at a host, after which the requesting controller is notified asynchronously.

We use this notify request facility to implement the &r_watch task. First we fetch the current value of the remote SDS via an &r_get and store a copy in a local SDS at the requesting controller. We then locally watch if this SDS copy has been changed. It will only be changed if the SDS-host has notified the requesting controller that the original remote SDS has obtained a value which is different from the copy locally stored. For testing equality we can use the generic equality function which can test equality for any first order type. When we receive a new value from the SDS host it is stored in the SDS copy, which change will trigger the waiting task as usual. This process is repeated as long as the &r_watch operation remains active.

4.3 Evaluating Tasks Remotely

In Section 3 we already explained how Clean functions can be serialized, shipped to another processor, de-serialized there and evaluated. As explained in Section 2.1.1, tasks are just Clean functions of a specific type Task a for which a context restriction holds, namely class iTask a has to be available for type a. In order to evaluate a task on another controller, it is not enough just to send over a closure of the task function. Also all generic functions defined in the type class iTask for type a are needed, otherwise the task cannot be evaluated. For this purpose we define the following container type Remote_Task for shipping tasks to another processor:

```haskell
:: Remote_Task
  = E.a: Remote_Task (Task a) TaskAttr InstanceNo & iTask a
  | NoTask
```

This existentially quantified Algebraic Data Type can contain any task of any type (Task a) together with its task attributes and task number which is used as unique identification of the task. In this type definition & iTask a defines a context restriction: the ADT can only contain tasks of type Task a for which also an instance of class iTask a has been defined. A dictionary containing all the members of this class is automatically added to the constructor of the ADT when a value of this type is created. Hence, when the container is shipped, it will not only contain a task of a certain type, but also all required generic function instances for that type, such that all methods needed to evaluate the task elsewhere are included. When a message is received which cannot be de-serialized to a proper task as described, the container is not accepted.

When a task is being evaluated, we also might need to inform the sending controller about the current state of the task. The shipped task, like any other task, yields an observable task value which changes over time (see 2.1.1). Via e.g. a step operator one can observe a task value that might change because somebody is working on it. In the new distributed setting it has as consequence that the shipping controller must be able to observe the current value of the shipped task evaluated elsewhere.

To support this property in the distributed version of iTasks, we create two special tasks, a proxy task on the sending controller and an evaluating task on the remote controller. The proxy task
The ability to have multiple controllers can be used to decrease the workload of a domain controller. Another server-side controller can takeover part of the work from a domain controller to avoid that such a controller becomes a bottleneck.

A controller that wants to take over work from the domain controller can describe the tasks to take over via a claim filter. This is a closure of type \( \text{claim} :: \text{TaskAttributes} \rightarrow \text{Bool} \) that defines which tasks are wanted. The claim filter is send to the domain controller and when a new task is added to its task pool administration, the domain controller first tries to search for a controller that wants to claim the task. The first controller that is found (e.g. its claim filter yields true) gets the task assigned and then receives the task.

One can freely assign attributes to a new parallel task when a task is created. In this way one can specify for whom a task is intended, or define other demands, such as a specific role, processor, resource, or whatever. With the claim filter one can filter out those tasks one wants to handle or can handle. This can be used to decrease the load of a domain controller. For example, the following claim filter claims all tasks of the users with a name starting with a character of the first part of the alphabet.

```haskell
claimUsersAM :: TaskAttributes -> Bool
claimUsersAM attrs
  = let name = readAttr "createdFor" attrs in
    (\"a\" <= name && name <= \"m\") || (\"A\" <= name && name <= \"M\")
```

The domain controller of course needs to redirect users to the right controller after they have logged in.

### 4.4 Distributing Tasks Server-Side to Decrease Workloads

Handling task events can be done faster with a private controller located on the machine of the end-user. As long as no interaction with other end-users or remotely located SDSes is needed, one can even work off-line.

A local controller is an iTask controller, which can be used in the following ways:

- A local controller can connect under a user name to its domain controller. Only those tasks intended for that user can be seen and downloaded. If the end-user is the root user basically any desired task can be downloaded.
- Any number of local controllers can make a connection to their domain controller. The domain controller possesses an additional administration, the local controllers administration, in which the currently connected local controllers are administrated.
- Local controllers can connect or disconnect themselves from the domain controller at any moment in time. It may of course happen that a local controller is temporarily disconnected from the net.
- A local controller can subscribe to the task pool of the domain controller and define which kind of the tasks visible to the local controller are currently relevant and need to be downloaded. What is relevant can be specified via a claim (see previous section) send to the domain controller. Attributes can be freely assigned to tasks to do. These attributes can be used to filter the tasks to download to the local controller. One can filter the tasks intended for a specific user, and one can also ask for those tasks that need a camera, or other specific resources required for doing the task. A local controller can subscribe to the task pool of another local controller. A whole chain, or a tree, or any topology of local controllers can in principle be made this way.
- The effect of a subscription is that the local controller will receive a notification whenever new relevant tasks are added to the task pool of the controller they subscribed to. The notification message contains a wrapper task with a reference to the task in the task pool of this (domain) controller that is used to download the task when one wants to evaluate the tasks on the local controller. This evaluation of a task on the local controller takes place just as described before. So, the task might access SDSes located elsewhere on the net, and its task value can be observed by tasks located elsewhere as well.
- When a task is downloaded from a controller, it is administrated in the task instance administration which local controller is working on it. This is a kind of lock to prevent other controllers or end-users to work on the same task as well. In iTasks only one person or system is allowed to work on a certain task at the same time.
- Like on a standard iTask system, it remains possible to explicitly steal a task someone is already working on. This can be done by the same user who wants to switch to a different device or by another user with the same role, who has decided to take over the work under evaluation. If the task instance of the task to steal is available, most of the work done so far can be rescued. If the task has been downloaded to some other controller who is not available any more for some reason, the task instance is completely lost. However, the original task to work on is still available and can therefore be restarted from scratch.

By default, the tasks to work on are presented to an end-user for which the tasks are intended in a list, much like incoming emails are presented in an email server. The same end-user can login as many times as desired, directly on the domain controller, or indirectly via a local controller connected to a domain controller. See, for
instance the example given in Figure 9 where Alice is logged-in twice. However, Alice is not allowed to work on the same task on both machines. So, when she starts to chat with someone on one of her machines, this task is blocked on all others. She can steal the task onto the other machine by asking the iTask system to move the task. The system will warn her that in that case local state information is lost. For the chat example this means that only the current sentence being typed in the browser is lost, but the shared chats stored in the SDS remain unaffected.

4.5.1 Client-Side Configuration Options

There are different ways to connect from a client to a domain controller. Each way has certain advantages and disadvantages.

One can use a browser on any platform and login into the domain controller. This is the simplest way to interact. If too many end-users are connected to a controller, the interaction with a browser might slow down unacceptably, because all events generated by all browsers must be handled by the same domain controller. If one needs to perform heavy computations in the browser, it might get unacceptably slow also due to the use of JavaScript.

Using a private local controller at the client-side with a browser connected to this local controller is another option. It increases the client-side responsiveness because only local network traffic is needed to handle the browser events. Furthermore one obtains the possibility to work offline on tasks, which makes sense for tasks which require a lot of local work. Working offline with a browser can be done by connecting the browser to the local controller as localhost.

In the configurations above, standard browsers are still being used for doing the interaction. Another interesting option we offer is to create an all-in-one Android App as client.

4.5.2 Android iTasks Client App

An Android iTasks App consists of three components:

1. The first component is a local controller that is compiled from the same source code as the other iTask controllers. It is compiled as a shared library instead of an executable, such that it can be used inside an Android application.

2. The second component consist of additional functionality, clean functions which are compiled to native ARM code stored in the app. We do no longer need browsers to do client-side calculations, and are therefore not forced to use JavaScript. Clean compiled to native ARM code runs about ten times faster than JavaScript code and is not hampered by stack size limitations.

Another advantage is that in the App, if granted upon installation of the App, we have access to all hardware devices, resources, file systems, and operating system facilities which is not possible when a standard browser would be used. Any component which has a C interface can be accessed, since Clean offers a C interface. If a component needs to be accessed via a Java library, Java Native Interface (JNI) can be used to call Java methods from C. In this way we can e.g. access the Bluetooth stack or the cameras on the device.

3. Finally we have an option to include a browser component in the App and connect it to the local controller. In Android we use the WebView component that is part of the Android platform as a browser.

To show the possibilities we have added some library tasks to access specific Android devices. The Device.Camera API allow users to take JPEG pictures using the device’s camera. The function

\[
\text{mkPict} \colon u \to \text{Task} (\text{Maybe (Msg TextPicture)})
\]

returns Just (P p) :: Maybe (Msg TextPicture) (18). The infix operator \( \circ \) is function composition.

\[
\text{mkPict} \colon u \to \text{Task} (\text{Maybe (Msg TextPicture)})
\]

\[
= \text{get device} \\
\gg> \text{\{dev \to \text{enterInformation}'' []\}} \\
\gg> [\text{OnAction (Action "Send" []) (hasValue (newMsg me o T))} \\
, \text{OnAction (Action "Picture" []) (ifCond (hasCamera dev) \\
\text{\{msgPict me\}}) \\
, \text{OnAction (Action "Stop" []) (always (return Nothing))} \\
\}]
\]

\[
\text{mkPict} :: u \to \text{Task (Maybe (Msg TextPicture))} \\
= \text{takePicture} \gg> \text{maybe (enterPictureChat me) \{newMsg me o P\}}
\]

\[
\text{chatTextOrPicture} :: \text{Task [Msg TextPicture]} \\
\text{chatTextOrPicture} = \text{createDistChat enterPictureChat}
\]

Figure 10. Chat example with support for sending pictures using the viewInformation editor because there is an editor defined for JPEG image in iTasks.API.Extensions.Pictures.JPEG module. The Device.Location API allow users to share the location using the location service of Android and let you retrieve the current location using GPS. When the device has a public interface we can also make use of a task that connects to a TCP server that is managing the hardware, or a task that calls a web service.

4.5.3 iTask Controller on a Raspberry Pi

Since a Raspberry Pi uses ARM code, we can also run an iTask controller on a Raspberry Pi. This option is commonly not used to interact with end-users, but it is a very convenient option to provide access to the resources of the Pi. By providing an interface to its resources, one can simply write iTask tasks (in Clean) to be executed on the Pi. Interaction with the other tasks and SDSes one obtains for free.

4.6 Examples

To give an impression how iTask applications can make use of the new options offered by our distributed architecture we present some small examples.

4.6.1 Extended Chat Example on Android

The first example is an extension of our chat example of Section 2.2, with a new option to make a picture. In Figure 10 we show the code of the extended enterChat. All other code remains the same.

To enable sending either a text message or a picture, we define the Algebraic Data Type TextPicture (1). The characteristics of a device on which a task is executed, is defined in a record of type DeviceInfo stored in the share device. In (8) this information is read, and used later (11) to find out if this device has a camera. The new enterChat task is almost identical to the previous one. The end-user can still type in and send a piece of text (10), or stop chatting, but there is an additional option (11) added for making a Picture via the task mkPict (16-18). This option, can only be chosen if the device on which the task enterChat is executed, has a camera. If so, the takePicture task (18) will activate the camera and a picture can be taken. But an end-user may decide not to use the activated camera after all, that is why takePicture is of type Maybe Picture. In case a picture \( p \) is made, enterPictureChat returns Just \( \text{\{Maybe (Msg TextPicture) (18). The infix operator \( \circ \) is function composition.} \end{verbatim}
Figure 11. Alice and Bob using the extended chat task

In case the end-user decides not to use the camera, enterPictureChat is called recursively (18) offering all chat options from the start.

This extended chat example can run on any iTask configuration. A screen-shot of Alice and Bob using the enterChat task of Figure 10 is given in Figure 11. Here Bob is using an iTask App located on his Android Tablet, he has taken a picture with the camera and has send this picture to Alice.

4.6.2 Poll All Users in a Domain

In Figure 12 we present a poll application that asks all the users of a domain which option they prefer given a list of alternatives, e.g. which restaurant they prefer from a list, or for who they are going to have dinner. We define this task in a very general way: the option list can be of any type. The task pollUsers first asks its end-user what question has to be asked to all users (3), after which the list of options can be entered for the answers that they are allowed to give (4). Next, a domain can be chosen (5), after which all users (allTasks) in that domain will be asked the same polling question (7-10) using the timeoutTask. This timeoutTask starts a given task which will wait for a given time (here timeoutTime) for the result. Hence, if a user does not answer in time, Nothing is returned (27). Otherwise Just the answer given by the user is returned (28). All answers received in time are counted and sorted, and shown to the end-user who started the poll. Notice that pollUsers is overloaded, so basically one can ask for answers of almost any type. For example, in myPoll they are of type String.

One of the facts of life of giving tasks to users is that they simply may not do their work. This of course already was the case in the old architecture, but in the distributed settings it is more likely that tasks are not performed, or not performed in time. Users may be off-line for a while, local controllers might be switched off, or even crashed. So, a programmer has to be more aware that work does not take place as planned.

4.6.3 Measuring Temperature on a Raspberry Pi

The following example uses a task that is executed on a Raspberry Pi. It has a temperature sensor that is connect to the GPIO pins of the Raspberry Pi. For the nitty-gritty details how this works we refer to [8]. The measured temperature is stored in a file in the filesystem. The filesystem of the Raspberry is however not accessible via a Web API. One could create a small service using e.g. TCP to make the information available, or implement an SSH client to obtain the file content. However, in our new distributed system we can simply install a local iTask controller on the Raspberry Pi, ship tasks to it for execution to do whatever we want with full access to all available sensors and resources. For example, we can define a task which reads the content of the file containing the temperature, convert it to a real number, and communicate the result back to the requesting controller via an SDS. In Figure 13 the host task showTemperature first creates an in-memory share for storing the measured temperature, convert it to a real number, and communicate the result back to the requesting controller via an SDS. In Figure 13 the host task showTemperature first creates an in-memory share for storing the measured temperature.

Figure 12. Example showing a simple dinner poll application

Figure 13. Example showing measuring of the temperature.
ing a temperature (4). Next it continuously with the thermometer which displays the temperature stored in this share (8) while starting a task on the Raspberry Pi (9). This task repeatedly does the following: it waits a given time interval (17), reads the temperature from the file resulting in a real value (18), and only when the temperature differs from the previous measured one, it stores the new value in the SDS located remotely on the host (20). Meanwhile the host continuously displays this SDS containing the last stored value by the Raspberry Pi. Figure 14 shows a screen-shot of the host task. The thermometer is created by the updateSharedInformation task that has as updateOption the thermometer function that describes how to display the temp type as a SVG image using the SVG extension of iTask [1].

4.7 Properties of the Distributed iTask Architecture

The new distributed iTask architecture is quite different from the previous architecture with only one central server. Having one central server also has advantages: the architecture is much simpler and therefore easier to maintain. Deploying and maintaining a distributed set-up is more difficult. Besides that, one can wonder whether a distributed system behaves differently and has different properties. Clearly, the new architecture affects the responsiveness of the individual tasks. The coordination of tasks is now spread over several controllers instead of handled by one and the same central server. Generally the responsiveness of the task handling improves when several controllers are used, certainly when private local controllers are added. But it may also slow down a task when there is a lot of interaction between tasks or SDSes located on different devices.

In the old system, the clients are commonly browsers. Browsers can disconnect from the central server any moment in time. As a consequence, the client-side state is lost in the worst case. In the new system, this may also happen, but now any controller might disconnect for a while as well, and private controllers might even disappear forever. If a controller gets disconnected for a while, e.g. when its processor is restarted, everyone working on this controller has to wait until it is up again. The controller stores its state in persistent memory, so in general no information is lost and no harm is done. Other controllers that have a connection with the restarting controller have to suspend communication until it is up again.

If a local or private controller is not responding at all, e.g. because it is located on a tablet or smart phone which died, the task instances, and all work done on it so far, are lost forever. If they have caused side-effects, e.g. changed SDSes located elsewhere, the effects cannot be undone automatically. So, one has to realize that this may happen when SDSes are being used. When tasks have been delegated to a dead controller, the task in question can be delegated again to another controller, but the work is lost. When the crashed controller is waiting for tasks delegated elsewhere, these tasks have become garbage and at some point in time they have to be removed from the controllers involved.

Disappearing controllers cause problems, but similar problems can occur in the old system in a different way. End-users may refuse to do their tasks or they may stop working on them at any time, and in this way they may hold up the whole process on a similar way. Local controllers which disappear have the same effect, the work will not be done anymore by them. When defining tasks one has to take into account that people, systems, or controllers won’t do their job and define alternatives (using the parallel or combinator) or set deadlines. For example, in the chat example, both users involved in the chat are able to stop the chatting if the other one is not responding. Although we do not have a formal proof, we believe that, although there is a higher chance that something goes wrong and the behavior in time is different, semantically the new distributed system behaves as the old one. We also believe that the distributed setting does not introduce new race conditions. Controllers handle events and update requests of SDSes one-by-one indivisibly.

5. Related Work

With the controllers network, we have created a network of processes at different machines and connected them. One can compare this with the processes in Erlang [2]. They support a standard error recovery with the special property that, when a process terminates abnormally, the other processes are notified through the link and can respond. This mechanism even allows layering so that processes can be isolated and restarted when an error occurs. We do not have such a standard error recovery mechanism. A (local) server may terminate anytime due to e.g. a closed or crashed user device or an internet connection that is lost, hence we are not able to signal such a problem from the machine to the controller. However, controllers are connected to each other, so other controllers can find out that a certain connection is not responding anymore. If the disconnected controller returns from the grave, all can continue as before. If the controller is lost forever, we have the same ability as in Erlang to restart a task from scratch and assign it to some other controller.

Yinzhou Zhu and Baolin Yin describe an Application-level Web Component Framework for Distributed Workflow Management [18] where there is a notion of server nodes and client nodes. The clients in this system are browsers only. Task evaluation on a client is not supported. It is not clear to us what happens when a server is going offline due to a failure.

A similar approach of dividing work and working with nodes or peers is the Web Workflow Peers Directory (WWPD) system that offers a peer to peer (P2P) architecture for dynamic workflow management [5]. The system uses a list of all the peers that are available in the Web Workflow Peer Directory (WWPD). In this system a peer registers itself and offers its services. The task description language they use is different from ours. The WWPD system uses the Workflow Process Description (WPD), an XML based document containing the task description and their references like URLs. The system does not send over tasks to evaluate remotely. Instead it offers a reference to the place where the task (or system) can be found. The system uses an approach similar to our approach for the task instance pool, where a server manages the task pool and knows how to find all the available clients.

Deriving an executable from a specification in the form of a workflow diagram is described in [12]. The authors derive a distributed version where the work is split in the specification. In that case the tasks can be distributed. The iTask combinators are more general and do not rely on just splitting tasks, although we can express this easily.

Using a single source to generate code for clients and servers is also done in the Eliom [16] project that provides a framework for writing web applications. In their language it is explicitly stated in the code which part is intended for a client and which part needs to be executed on a server. In iTasks any function or task can be send to any other server or client and this decision can be made at runtime. So, we are much more flexible. The Eliom approach has the advantage that it is statically known what can run on a client, which
may be important to know for security crucial code. In our current implementation we assume that all code is available on all devices where controllers are running. Lazy, just-in-time code distribution is future work, but can be done without problems.

6. Conclusion and Future Work
In this paper we have presented a new distributed architecture for the evaluation of iTask tasks and discussed its implementation. The new architecture is a generalization of the old iTasks system that had only one central server. In the new architecture we can have multiple servers (controllers) running distributively over a network, both on the server as well as on the client side. Tasks can be pushed or pulled from one controller to another, regardless the platform they are running on. We are now able to solve a number of disadvantages of the old iTasks system: 1) The system is now scalable because we can divide work over multiple server-side controllers. 2) Private client-side controllers allow an end-user to download tasks to work on them off-line. 3) We can generate client-side Android Apps running in native code which enables us to create applications that perform CPU-intensive computations, avoiding the use of JavaScript. 4) Using an App as client also allows us to make use of any facility the Android platform offers, something which is not allowed in browsers as well.

Clean applications are fast because we can generate state-of-the-art native code. The implementation of the distributed platform independent iTask system builds on the new ability to ship, at run-time, any Clean function for evaluation from one platform to another. A symbolic, platform independent serialization of a closure can be constructed given the current state of stacks and heap on one platform, shipped over, and de-serialized to the proper stack and heap representation of the other platform. We currently support 32-bits ARM for Android and Linux (Raspberry Pi) and 32- and 64-bits Intel code for Mac, Windows and Linux systems. In addition we can compile Clean to JavaScript code to run in browsers.

The current implementation has the disadvantage that the native code (Intel or ARM) has to be available on all controlling devices on the server, either as executable or as App. If a browser is used, the necessary code to evaluate a closure is just-in-time shipped to it. Our future plans are to do something similar, as option, for the local controllers. Since we are running native code, and not an interpreter, it means that one has to be able to extend a controller with new code while it is running. We are already able to do this by using the Windows making use of Cleans Dynamos in combination with a dynamic linker [17]. This mechanism has to be extended to the other platforms as well. As an alternative option we are also working on an interpreter of the platform independent ABC-code the compiler generates. It will run slower than compiled code, but is easier to port and easier to extend with new code.

The source code of the distributed version is available at https://gitlab.science.ru.nl/distributed-itasks. It contains the examples and scripts to compile for multiple platforms.

Acknowledgments
The authors would like to thank Bas Lijnse and Tim Steenvoorde for their valuable comments on a draft version of this paper.

References
Chapter 2

TOP / iTasks in a Nutshell

In the paper we explained the most important combinators that were used in the paper. In this master thesis we are using (special) combinators that where not explained and used in the paper. The combinators that we explain here are used in the implementation or in the case study.

The >>- combinator is defined as:

\[(\triangleright\triangleright-) \quad \text{infixl 1} \quad : \quad !\text{(Task a)} \ (a \to \text{Task b}) \to \text{Task b} \mid \text{iTask a} \& \text{iTask b}\]

The >>- combinator is almost the same as the >>= combinator except that >>- continues when Task a has a stable value.

The @! combinator changes the result of a task whiteout considering the value of the task. This is used e.g. when two tasks are performed in parallel (-\|-) and need to have the same type.

\[(@!) \quad \text{infixl 1} \quad : \quad !\text{(Task a)} \ !b \to \text{Task b}\]

2.1 Loops

In the distributed version of iTasks, we are using some tasks that are performed continuously (e.g. as loops). Writing this in the following way can cause heap space problems:

\[
\begin{align*}
\text{myLoop} & : \text{Task String} \\
\text{myLoop} & = \text{someTask} \\
& \triangleright\triangleright\ \text{result} \to \text{if (isNothing result) myLoop (return (fromJust result))}
\end{align*}
\]

In iTasks the <! combinator can be used to write loops that are optimised to prevent the heap space problem. It makes sure that the old task is removed when the task is restarting.

\[
\begin{align*}
\text{(<!)} \quad \text{infixl 6} \quad : \quad !\text{(Task a)} \ ((a \to .\text{Bool}) \to \text{Task a} \mid \text{iTask a})
\end{align*}
\]

\[
\begin{align*}
\text{enterBiggerTwenty} & \quad \text{is an example of the <! combinator where the task enterNumber is restarted every time the user did not enter a number bigger than 20.}
\end{align*}
\]

\[
\begin{align*}
\text{numberBiggerTwenty} & : \text{Task Int} \\
\text{numberBiggerTwenty} & = \text{enterNumber <! (<<) 20} \\
\text{where} \\
\text{enterNumber} & : \text{Task Int} \\
\text{enterNumber} & = \text{enterInformation "Enter a number" []}
\end{align*}
\]

A variant of the <! is the forever task, this task performs the loop without ending (e.g. task <! (Const False)).

\[
\begin{align*}
\text{forever} & \quad : \quad !\text{(Task a)} \to \text{Task a} \mid \text{iTask a}
\end{align*}
\]
Chapter 3

Enabling Distributed Evaluation of Clean

Symbolic serialisation and deserialization

In the paper (section 3) we mentioned the graph-to-string-with-names and string-to-graph-with-names function. This function can serialise almost any type a to a symbolic representation. The graph-copy module which graph-to-string-with-names and string-to-graph-with-names functions are part of, is developed by John van Groningen. The graph-copy module is changed for the distributed version of iTask to support multiple architectures, the changes are available at https://gitlab.science.ru.nl/clean-and-itasks/clean-graph-copy. My contributions in this module are mainly adding support for Android and MacOS. In this chapter we will explain the most important changes in the graph copy module.

Graph-to-string-with-names

Graph-to-string-wit-names can serialise almost any type or closure to a platform independent representation except unique types like *World. This restriction is to prevent that one can copy the unique world. The working of the function is as follows: the graph-to-string-with-names function first uses the already available graph-to-string function to obtain a string representation of the graph that is platform depended. It contains memory addresses which are platform depended. In case the operating system uses Address Space Layout Randomization (ASLR), the string representation even depends on the current instance of the process.

The next step is that the function replaces every memory address with the Clean module name and symbol name. The memory address is then replaced by a reference to the table containing the symbols used in the string representation. To let the Clean compiler, include the symbols for the program, someone needs to compile using: cocl -desc -exl -d where cocl is the Clean compiler.

String-to-graph-with-names

To regenerate the type or closure from the symbolic representation the function string-to-graph-with-names reverts the process. The module requires the symbols and the corresponding memory addresses. We use the symbol table of the executable. There are several symbol table formats. We support:

- ELF for Linux 32- and 64-bit;
- Mach-O for Mac-OS;
- PE/COFF for Windows.
The function that reads the symbol table is: symbols-in-program. The version for Linux 32-bit (ELF) and PE/COFF is developed by John van Groningen, the version for Linux 64-bit ELF and Mach-O is developed by me.

The `string-to-graph-with-names` function uses the symbols in the symbol table to restore the addresses in the string representation of the type or closure for their address so that the string state can be converted back.

**Support for Address Space Layout Randomization**

To support Android and MacOS, we had to add support for Address Space Layout Randomisation (ASLR) because Android requires that all native code is compiled with the PIC flag since Android 5 [1]. PIC stands for Position Independent Code; this ensures that it does not matter for the code on which address it is located by referencing locations relative from the current location. See for more details: [7].

The support for ASLR is based on the solution for the function `graph-copy-with-descriptors`. The `graph-copy-with-descriptors` function also replaces the numbers with descriptors, it only supports serialisation in Clean and is used by SAPL. Deserialization is done in JavaScript. The basic idea is that we save all the descriptors relative to one of the first addresses in the symbol table. In Clean this is the `__ARRAY__` descriptor. Thus, for every symbol, we subtract from the address the address of the `__ARRAY__` descriptor.

In the module `string-to-graph-with-names`, we first search the `__ARRAY__` descriptor in the symbol table and the current memory location and calculate the difference. We then add the calculated difference to every address found in the symbol table before placing it back in the string representation.
Chapter 4

Distributed iTasks

In the paper we explained briefly how the distributed version of iTasks works. The explanation in the paper was describing this at a high level of abstraction due to the page limit. In this section we explain how the distributed version is implemented in more detail. We explain in section 4.2 how a proxy to a SDS in a distributed task is implemented. Section 4.3 explains the implementation of the `proxyTask` and `customEval` functions that allow observing a task that is evaluated on another machine. Finally, section 4.4 describes how local controllers are embedded in for example an Android app.

4.1 Distributed Domain Controllers

To make distributed domain controllers possible we implemented a TCP server for the domain controller and TCP clients for the local controllers.

4.1.1 Domain Controller

The domain controller is a TCP server written in the iTasks framework.

4.1.2 Local controllers

The local controllers are TCP clients that connect to the domain controller. The client ensures that:

- When a client is disconnected when, for example, the connection with internet is interrupted, the client waits for one minute and tries to reconnected. All the messages that need to be sent to the domain controller cannot be send and are delayed until the connection is re-established again. In section 4.4.2 we explain the reconnecting process in more detail.

Someone can wonder what happens if the SDS (e.g. queue) is full. The SDS is stored on disk so that means that there is no storage space left. When this occurs we simply cannot store values in the share and there will be an exception created. When the writer of the task does not handle it, the task will end with an exception. Storing on disk is done because this allows stopping (e.g. kill) the controller process (or app) and restart later.

4.2 Accessing Remote Shared Data Sources

In the paper, we explained that for every SDS (`get`, `set`, `upd` and `watch`) we implemented a new version called `r_get`, `r_set`, `r_upd` and `r_watch`. What these remote variants first do is checking if the task that is performing the operation is a local or a distributed task by using the function `distributedOrLocal` (Figure 4.1).
CHAPTER 4. DISTRIBUTED ITASKS

Figure 4.1: Check if the task is local or distributed

```haskell
distributedOrLocal :: (InstanceNo InstanceNo -> Task a) (InstanceNo InstanceNo -> Task a) -> Task a | iTask a
  = get currentTaskInstanceId
  >>= \taskId -> currentDistributedId
  >>= \instanceNo -> if (isLocal instanceNo)
    (taskLocal taskId instanceNo)
    (taskDistributed taskId instanceNo)
```

Figure 4.2: Getting the value from a remote shared data source.

```haskell
r_get :: !(ReadWriteShared a w) -> Task a | iTask a & iTask w
  = r_get shared
    = distributedOrLocal (\_ _ -> get shared) (remoteGet shared)
    where
      remoteGet shared taskId sharedId
        = addSharedHandler taskId (putSharedValueInDummyShared tempShared)
        >>| sendRequestToShareHost sharedId (message ref myRef shared)
        >>| watch tempShared >>* \[OnValue (ifValue (not o isNothing) return o fromJust)]
      message ref myRef shared = 'T'.concat "␣" ["share", ref, myRef, "get", sharedFunc ]
      where
        sharedFunc = (serializeToBase64 (Remote_Share shared))
```

The check to determine if the task, is a distributed task is checking if the task has an attribute containing the `taskId` of the task, if that is the case, the task is a distributed one.

4.2.1 Get a value from a Remote Shared Data Source

In Figure 4.2 the function `r_get` is presented. The remote get function is a simple function that obtains the value from a remote share using the following steps:

1. Create a dummy shared (line 8) to store the result in.
2. Add a shared handler. This is a function that stores the value received from the shared host in the given shared when received.
3. We send a request to the shared host to get the current value in the shared.
4. We wait using a watch for any value in the dummyShared that is not `Nothing`. The trick is here that we store the value as `Maybe a` where `Nothing` means there is no value yet. So, when the type is, for example, `Maybe String` the type we store in the shared is: `Maybe (Maybe String)`.

Someone can wonder what happens when the remote host is down, all this is handled by the client code. This code will resend the request or reconnect when it was disconnected. The code performing operations on shares is limited to describing how the format the request to the server and emulate the behaviour as of the SDS was located on the current controller.
4.2.2 Observing a remote shared

In the paper, we explained that the \texttt{r\_watch} tasks uses a proxy to keep the network traffic to a minimum and still offers the same functionality.

We consider only the case for a remote shared.

\begin{verbatim}
\begin{verbatim}
r_watch :: !(ReadWriteShared r w) \rightarrow Task r \& iTask r \& iTask w
r_watch shared
  = distributedOrLocal (\_ \_ \rightarrow watch shared) (remoteWatch shared)

remoteWatch shared taskid sharedid
  # myRef = (toString taskid) ++ "watch"
  # ref = (toString sharedid)
  # tempShared = dummyShared myRef Nothing
  = set Nothing tempShared
    >>=| addSharedHandler myRef (putSharedValueInDummyShared tempShared)
    >>=| sendRequestToShareHost sharedid (message ref myRef shared (const True))
    >>=| waitForNotify tempShared
    >>- \val ->
      let tempWatchShared = dummyShared (myRef ++ "dummy") val in
        set val tempWatchShared
        >>=| (watch tempWatchShared)
        >>| (loop val tempWatchShared taskid sharedid shared)

waitForNotify shared
  = watch tempShared >>= [OnValue (ifValue isJust (return o fromJust))]

message ref myRef shared notifyfunc
  = '\text{T}'.concat "," ["share", ref, myRef, "notify", shareFunc, "function", notifyFunc]

loop initialValue watchShared tempShared taskid sharedid shared
  = setValue initialValue tempShared taskid sharedid shared
    >>=| forever (repeat watchShared tempShared taskid sharedid shared)

repeat watchShared tempShared taskid sharedid shared
  = watch tempShared >>= [OnValue (ifValue isJust \v \rightarrow (set (fromJust v) watchShared))]
  >>= \value \rightarrow setValue value tempShared taskid sharedid shared
  # myRef = (toString taskid) ++ "watch"
  # ref = (toString sharedid)
  = set Nothing tempShared
    >>=| sendRequestToShareHost sharedid (message ref myRef shared (const True))
    >>=| return value
\end{verbatim}
\end{verbatim}

Figure 4.3: Watch operation support for remote shares.

The \texttt{remoteWatch} (Figure 4.3) is implemented using the following steps:

1. A local (\texttt{myRef}) is created so that the current controller knows what to do with a response.

2. A dummy shared is created (line 8). This is an SDS that is stored in memory and is used so that value results can be stored.

3. A notify request is sent to the shared host. This notify request will answer directly (\texttt{const True}) when the SDS has a value. It can also be the case that the SDS does not have a value yet (and not a default someone) in that case the notify will notify \texttt{when} the first value is
set. One could wonder why not using a \texttt{get} instead of this complicated trick. The answer is that a \texttt{get} will give an error when there is not a value set \textit{and} there is not a default value specified for the SDS.

4. When the notify is received a second dummy shared is created (line 13). We insert the just received value and then the dummy shared is used to perform a \texttt{watch} operation on. At the same time, we use the parallel combinator \texttt{-||} to ask for a notification when the value in the shared is not the same anymore than the current one (line 31). The parallel combinator \texttt{-||} ensures that we keep asking for updates as long as the observing tasks does not decide to continue (e.g. \texttt{watch myShared [OnValue (ifValue ((< 5)) return)]} where the watch is ended when the value in the shared is bigger than 5.)

A remark for the source code:

- The \texttt{forever} construction can also be written as a loop. Writing this as a loop can cause problems with the heap. To prevent this, we use \texttt{forever} that is optimised to prevent space heap problem in the case of loops.

### 4.3 Evaluating Tasks Remotely

In Figure 4.4 we list the \texttt{proxyTask} function that given a SDS \texttt{RWShared (TaskValue a) (TaskValue a)} emulates the observable result of \texttt{Task a}.

When the \texttt{proxyTask} is observed the value of the SDS is read (line 5), the value from the SDS is then returned as the \texttt{ValueResult} of the Task (line 7-10).

The second argument of the \texttt{proxyTask} is used to deal with the case that the task is destroyed e.g. is not needed anymore. This happens for example in the following example:

\begin{verbatim}
example :: Task Int
example
  = getDomain
  >>> \domain -> usersOff domain
  >>> users -> enterChoice users
  >>> user -> user @. domain @: enterNumber
  >>> [OnValue (ifValue (\v -> v > 20) return)]
where
enterNumber :: Task Int
enterNumber = enterInformation "Enter a number" []
\end{verbatim}

When the user enters a number bigger than 20 the example tasks is destroyed at the controller where the task is created. The \texttt{enterNumber} task is not destroyed at the controller where this is performed in the distributed setting and will not be destroyed. To solve this problem, we use the second argument of \texttt{proxyTask (onDestroy)} to send a notify the domain controller that the task is destroyed and the creator of the task is not interested anymore. The domain controller will notify all the controllers that know about this task and they will remove the task from the task list. When a user is working on the task the task is stopped to prevent that the user continues to work on something that nobody will use.

An example of the \texttt{onDestroy} implementation is:

\begin{verbatim}
onDestroy :: InstanceNo (Shared ClientShare) *IWorld -> *IWorld
onDestroy id share iworld
  # destroyMessage = "instance_destroy," ++ (toString id)
  # (_,iworld) = modify (\client -> (True, send client destroyMessage)) share iworld
  = iworld
\end{verbatim}

The \texttt{onDestroy} function is not in the task domain and we cannot use task functions. Because we use a SDS to queue messages for the domain controller we can use \texttt{modify} to change the content of a share without being in the task domain.
Figure 4.5 lists the `customEval` function that given a SDS `RWShared () (TaskValue a) (TaskValue a)` and a `Task a` stores all the observable values in the SDS.

```haskell
proxyTask :: (RWShared () (TaskValue a) (TaskValue a)) (*IWorld -> *IWorld) -> (Task a) | iTasks a
proxyTask value_share = Task eval onDestroy
where
  eval event eval0pts tree=:(TCInit taskId ts) iworld
  # (val,iworld) = ‘SDS’.readRegister taskId value_share iworld
  = case eval of
    Ok val  -> (ValueResult val (TaskEvalInfo{lastEvent=ts
      ,removedTasks=[]
      ,refreshSensitive=True}
      (finalizeRep evalOpts NoRep) tree, iworld)
    Error e -> (ExceptionResult e,iworld)
  eval event repAs (TCDestroy _) iworld
  # iworld = onDestroy iworld
  = (DestroyedResult,iworld)
```

Figure 4.4: Proxy task implementation

```haskell
customEval :: (RWShared () (TaskValue a) (TaskValue a)) (Task a)
  -> (Task a) | iTasks a
customEval value_share (Task eval) = Task eval'
where
  eval' event eval0pts state iworld
  = case eval event eval0pts state iworld of
    v:=(ValueResult _, _) -> storeValue v
    (ExceptionResult te, iworld) -> (ExceptionResult te, iworld)
    (DestroyedResult, iworld) -> (DestroyedResult, iworld)
    storeValue (ValueResult task_value info rep tree, iworld)
    # (res, iworld) = ‘SDS’.write task_value value_share iworld
    = (ValueResult task_value info rep tree, iworld)
```

Figure 4.5: Custom eval implementation

4.4 Local controllers

The Android App is an example of a local controller that is running on a user’s device. In this section, we explain the changes to make it possible to run iTasks embedded in an app. We split this section in generic changes that are also reusable for other platforms like iOS and in Android specific changes.

4.4.1 Running iTasks as a library

The iTask framework assumed that the application was always executed as an executable. The engine was, for example, searching in the startup arguments for parameters (e.g. `-port 8080` to set the web server port). There was no way of changing this port except using the command line arguments.

This is unpractical because when iTasks is embedded in another application as a library, the command line arguments can already be used by the host application. Another example is applications that do not have support for command line arguments like Android or iOS apps. To solve this problem, we proposed a change for the iTask framework that allows starting the engine using

A Distributed Server Architecture for Task Oriented Programming 24
# options =  
{ appName = "Fancy␣name"  
, appPath = "/path/to/the/app/folder"  
, sdkPath = (Just "/path/to/iTasks/SDK")  
, serverPort = 8080  
, keepalive = DEFAULT_KEEPALIVE_TIME  
, webDirPaths = Nothing  
, storeOpt = Just "/path/to/store/"  
, saplOpt = Nothing  
}  
= startEngineWithOptions [ publish "/" (WebApp []) (_-> someTask) ] options world

Figure 4.6: Embedding an iTasks application

implementation module example

import System._Unsafe  

StartAsLib :: Int  
StartAsLib  
= if (libraryStart start) 1 0  

// This requires that there is a StartAsLib definition in the .dcl file.  
foreign export StartAsLib  

libraryStart :: !(#World -> #World) -> Bool  
libraryStart start = appUnsafe start True  

Start :: #World -> #World  
Start = // Logic like it was a normal Clean executable.

Figure 4.7: Start point of the library

options. The proposed change (e.g. merge request) was accepted and is now part of iTasks-SDK for iTasks. An example to start iTasks with options (startEngineWithOptions) is given in Figure 4.6.

Some remarks:
- When appPath contains a folder WebPublic that contains the files from the Client folder in the iTasks-SDK. The sdkPath can be left empty (e.g. Nothing)
- When the iTasks-SDK is not found (e.g. sdkPath = Nothing clean.f needs to be located in the sapl folder. clean.f is located iTasks-SDK/Dependencies/clean-sapl/src/.

To embed iTasks in other languages than Clean it is compiled as shared library. To start iTasks someone can export a new start function as is done in Figure 4.7. Special about the StartAsLib function is that it does not have a #World parameter. An unique world is required to do IO in Clean and is required for iTask. To solve this problem, we create a new world with the method appUnsafe (line 13). The appUnsafe function uses ABC instructions to create a (new) world. In normal Clean applications creating a new world is unsafe because someone can now obtain multiple worlds and it is not usable anymore to force evaluation. In the case of a dynamic library, this operation is safe because we do not have the world and only one world is created.

1 https://gitlab.science.ru.nl/clean-and-itasks/iTasks-SDK/commit/65e528e5f0d3863fa10e815976f4a5292ee88ae7
4.4.2 Reconnecting

Local controllers like tablets or smartphones have the property that they can go (temporary) offline. This can happen because they depend on a Wifi connection and are temporary out of reach of a known Wifi network. For mobile devices with a mobile internet connection (4G), this does not happen so frequently in a country like the Netherlands where the mobile connections are quite good. Losing a mobile connection temporary can still happen when someone enters a building or is flying in an aeroplane. We have to deal with the situation in the following way:

- First of all, the tasks that need to communicate with another controller do this by putting the requests in a SDS that contains the requests for the controller. Helper tasks will make this easier. Because we use SDSs for the communication, all the requests can be done even when there is no connection. The SDS can be considered as a queue.

- The TCP client that connects to the other controller is responsible for sending all the messages that are stored in the SDS. When the TCP client disconnects, the TCP client reconnects after a few minutes. This is possible because all the communication that still needs to be transmitted is stored in the SDS. Figure 4.8 shows the task that handles the reconnecting, where the client task (line 3) is the TCP client logic that connects to the TCP server. Figure 4.9 shows a screenshot of the reconnecting screen.

- The domain controller handles the reconnecting clients in the following way: when a client connects for the first time, it expects the client to send a connect message. When this message is received, the client gets a new client id. When the client is disconnected, the client must reconnect with the client id.

```hs
repeatClient :: (Task (Maybe a)) -> Task (Maybe a) | iTask a
repeatClient task
  = (try task) <! isJust
where
  try :: (Task (Maybe a)) -> Task (Maybe a) | iTask a
  try task
    = catchAll task (\_ -> return Nothing)
    >>= \result -> if (isNothing result) tryAgain (return result)
          tryAgain :: Task (Maybe a) | iTask a
          tryAgain
            = waitForTimer' timeout @! Nothing
          where
            timeout = {Time| hour = 0, min = 1, sec = 0}
```

Figure 4.8: `repeatClient` task that reconnects when the client disconnects or some other connection problem occurs.

Figure 4.9: Screenshot of the local controller that is disconnected and waits for one minute before reconnecting.
4.4.3 Android app

The iTasks Android app is a very simple app that has three components:

- iTask controller embedded as a dynamic library;
- Interface between Android and Clean so that we can use the hardware of the device;
- A WebView [2] component that is a small embedded browser.

The embedding of the dynamic library for an iTasks application does not work. The user also needs the iTasks-SDK located on the tablet as mentioned before in section 4.4.1. The WebPublic and sapl folder are embedded in the Android App (APK file) as assets. Having files as assets inside an Android application has a few drawbacks:

- Assets are not available for the application as files. We need to copy them to the internal (private) storage that every Android app has;
- Assets that start with an underscore are ignored and not included in the APK file. In the sapl folder, there are several folders and files that start with an underscore. To solve this issue, we replace all the underscores with underscore-- and during the copy process to internal storage, we convert this back to underscores.
Chapter 5

Case study: Service engineer

To demonstrate the distributed version of iTask we created a realistic example the service engineer application. This example application is meant for a broad group of service engineers, for example, engineers that maintain coffee machines or engineers from your internet service provider, that repair your internet connection.

We start with describing the work of a service engineer that maintains, say, coffee machines. Let’s assume that a coffee machine is broken, somebody will call the maintainer to fix it. The persons who takes the call will write down all the information in a ticket. For example the location of the coffee machine, the person who reported the issue, contact information and type of the machine.

The ticket creator will then be able to see where the service engineers are (e.g. on a map) and can decide to assign the ticket to a service engineer.

The service engineers see a list of tasks (tickets) that need to be fulfilled. The service engineer picks one and opens the tasks, it will show the location of the issue (for example Radboud University Mercator 1 floor 1, Toernooiveld 212 6525EC Nijmegen). When the service engineer is at the location of the coffee machine, the repair of the machine begins. He will add a notation to the ticket and summarise the parts that are used to repair the machine.

Completing the machine repairing (from the perspective) of the service engineer means the machine is working again. The service engineer can continue to solve another ticket.

5.1 Service engineer status

Service engineers are assumed to be on the road. We use a task that is polling the location of the service engineer using an interval. This location is used to select an engineer that is nearby. To keep this case study simple, we do not consider lunch breaks or working times of the service engineers. We also do not consider the case that the engineer has already done enough work.

To get the status of the engineer we defined a task `engineerStatus` that is assigned to every engineer. The GPS coordinates are defined as a type in `API.Extensions.Device.Location`. For completeness, we included the definition of the type in Figure 5.1. To store the location of the service engineers we define a SDS (Figure 5.2) that contains a list of the names of the engineers and their last known coordinates.

```plaintext
:: Coordinates = LatLon Real Real
derive class iTask Coordinates
```

Figure 5.1: Coordinates type (defined in `iTasks-SDK/Server/iTasks/API/Extensions/Device/Location.dcl`)
The `engineerLocation` task (Figure 5.3) starts with selecting all the users in the current domain (line 3). The user then selects the user where the location status is needed (line 4). The `locationTask` (line 8) is then assigned to the user, so the user can open the task on a tablet where he or she is currently working on.

The `locationTask` is a task that is never ending (e.g. never obtaining a stable result). The task is waiting two minutes (line 11 and 22) and then obtaining the device location (line 12) using the `getLocation` task. This task uses the location sensor of the Android tablet to obtain the current location.

The `getLocation` task tries to obtain the current location. It can take some time. Especially the first time when the location is asked because there is some initialization needed by the Android device. Obtaining the location can also fail, in that case, `Nothing` is returned (line 18). In case the location is obtained (line 14). The SDS is updated with the current location. Because this task is assigned using the distributed version the share located on the server where `engineerStatus` is created is used.

When using the `engineerStatus` task we can now obtain the current location from all the service engineers where we assigned this task to.

```haskell
engineerLocation :: Shared [(String, Coordinates)]
engineerLocation = sharedStore "engineerLocation" []

Figure 5.2: Engineer location share

The `engineerStatus` task (Figure 5.3) starts with selecting all the users in the current domain (line 3). The user then selects the user where the location status is needed (line 4). The `locationTask` (line 8) is then assigned to the user, so the user can open the task on a tablet where he or she is currently working on.

The `locationTask` is a task that is never ending (e.g. never obtaining a stable result). The task is waiting two minutes (line 11 and 22) and then obtaining the device location (line 12) using the `getLocation` task. This task uses the location sensor of the Android tablet to obtain the current location.

The `getLocation` task tries to obtain the current location. It can take some time. Especially the first time when the location is asked because there is some initialization needed by the Android device. Obtaining the location can also fail, in that case, `Nothing` is returned (line 18). In case the location is obtained (line 14). The SDS is updated with the current location. Because this task is assigned using the distributed version the share located on the server where `engineerStatus` is created is used.

When using the `engineerStatus` task we can now obtain the current location from all the service engineers where we assigned this task to.

```haskell
engineerStatus :: Task Void
engineerStatus
  = get currentDomain
  >>= \domain -> usersOf domain
  >>= \users -> enterChoice "Select a user" [] users
  >>= \user -> user @. domain @: (locationTask user)

where
  locationTask :: User -> Task Void
  locationTask user
    = forever (
        waitForTimer interval
        >>= getLocation
        >>= \loc -> case loc of
          (Just cords) -> let id = toString user
                          in upd \data -> [i \\ i:=(user, loc) <- data | not (user == id)] ++ [(id, cords)]
                          )
          _ -> return Void
    )

interval :: Time
interval = {Time| hour = 0, min = 2, sec = 0 }

Figure 5.3: Service engineer status task

5.2 Creating a ticket

To let the user enter a new ticket when a service engineer needs to repair some machine we define the `Ticket` type (Figure 5.4).
CHAPTER 5. CASE STUDY: SERVICE ENGINEER

:: Address = { address :: String, zipcode :: String, city :: String }
:: ContactPerson = { name :: String, email :: Maybe String, phone :: Maybe String }
:: Ticket = { address :: Address, contactPerson :: ContactPerson, note :: Maybe Note }

derive class iTask Address, ContactPerson, Ticket

Figure 5.4: Ticket type

In Figure 5.5 we show the createTicket task that allows the user to enter a new ticket for a repair that needs to be done by a service engineer. The user interface for entering a new ticket is generated from the Ticket type by the iTasks framework (line 5). Figure 5.6 shows a screenshot of the editor derived from the Ticket type.

When the ticket is entered, we select an engineer using the selectEngineer (line 5) (see section 5.6) and assign the workOnTicket task. The workOnTicket task needs as parameter the just created ticket (line 6). When the service engineer has repaired the machine, we display the repairs that are part of the result of the task (line 7).

createTicket :: Task (Ticket, [Repair])
createTicket = get currentDomain
>>- \domain -> enterInformation "Create ticket" []
>>- \ticket=:{Ticket|address} -> selectEngineer address
>>- \engineer -> engineer @. domain @: (workOnTicket ticket)
>>- viewInformation "Ticket result" []

Figure 5.5: Ticket type.

5.3 Getting the GPS location of an address

In Figure 5.7 the getCoordinates task is listed that given an address, obtains the GPS coordinates using the Nominatim API from Open StreetMap [4]. This task has three steps:

- Encode the request as a URI (line 7 until 16). When we are encoding as a URL we replace all the spaces by the plus (+) character. Some people enter their zip code with a space for
example 1234 AB the Nominatim API wants the zip code without spaces. We handle this case at line 12 and 21.

- We use `callHTTP` (line 4) to send the HTTP GET request.

- When we have a response the function `parseFun` is called (line 23 until 22). This function obtains first the part of the response where we are interested in and then parses it as a JSON (line 24 and figure 5.8). When then extract the GPS location from the JSON response (line 26 until 31).

```haskell
getCoordinates :: Address -> Task Coordinates
getCoordinates address
  = case (url address) of
      (Just uri) -> callHTTP HTTP_GET uri "" parseFun
      _ -> return (LatLon 0.0 0.0)

where
  url :: Address -> Maybe URI
  url {Address|address,zipcode,city}
    = parseURI ('T'.concat ['http://nominatim.openstreetmap.org/search?q=
                                , encode address
                                , "+"
                                , encodeZipcode zipcode
                                , "+"
                                , encode city
                                , "&format=json"
                                )

  encode text = 'T'.replaceSubString "," "+" ('T'.trim text)

// 1234 AB is not allowed, use 1234AB
encodeZipcode zipcode = 'T'.replaceSubString "␣" "" zipcode

parseFun :: HTTPResponse -> (MaybeErrorString Coordinates)
parseFun response
  # data = toJSONResponse response "[" "]"
  # lat = case jsonQuery "0/lat" data of
         (Just val) -> val
         _ -> "0.0"
  # lon = case jsonQuery "0/lon" data of
         (Just val) -> val
         _ -> "0.0"
  = (Ok (LatLon (toReal lat) (toReal lon)))
```

Figure 5.7: Obtain the GPS location of an address.

```haskell
toJSONResponse :: HTTPResponse String String -> JSONNode

toJSONResponse {HTTPResponse| rsp_data} start end
  # s = 'T'.indexOf start rsp_data
  # e = 'T'.lastIndexOf end rsp_data
  | s == -1 || e == -1 = JSONNull
  # rsp_data = subString s (e - (s - 1)) rsp_data
  = fromString rsp_data
```

Figure 5.8: Convert a HTTPResponse to a JSONNode.
5.4 Obtaining a route

We now have the ability to obtain the GPS location of the tablet and the GPS location of the ticket location. Someone can calculate the distance between these points. However the distance is not realistic. The distance can be quit small but when someone needs to travel it takes longer because we need to cross a river and there is no bridge nearby.

To solve this problem, we use the yournavigation.org routeing API [5] that gives us the distance when travelling by car and the time in minutes it will cost to travel to the destination. The task has the same steps as the getCoordinates task.

```haskell
:: Route = { distance :: Real, traveltime :: Real }

derive class iTask Route

routeInfo :: Coordinates Coordinates -> Task Route
routeInfo fromLocation toLocation = case (url fromLocation toLocation) of
  (Just uri) -> callHTTP HTTP_GET uri "" parseFun
  _ -> return {Route| distance = 0.0, traveltime = 0.0 }
where
url :: Coordinates Coordinates -> Maybe URI
url (LatLon flat flon) (LatLon tlat tlon) = parseURI ('T'.concat [ apiUrl,
  "&flat=", (toString flat),
  "&flon=", (toString flon),
  "&tlat=", (toString tlat),
  "&tlon=", (toString tlon) ])

apiUrl :: String
apiUrl = "http://www.yournavigation.org/api/1.0/gosmore.php?" +++
"format=json&v=motorcar&fast=1&layer=mapnik&geometry=0"

parseFun :: HTTPResponse -> (MaybeErrorString Route)
parseFun response # json = toJSONResponse response "{" "}"
# distance = case jsonQuery "properties/distance" json of
  (Just val) -> val
  _ -> "0.0"
# time = case jsonQuery "properties/traveltime" json of
  (Just val) -> val
  _ -> "0"
  = (Ok {Route| distance = round (toReal distance) 2, traveltime = (round ((toReal time) / 60.0) 2)})
```

Figure 5.9: Obtaining route information using YourNavigation

5.5 Get engineers travel distance and time from address

Figure 5.10 shows the getEngineers task that shows all the users in the current domain (line 4) and shows the distance and travel time for the user to go to the given GPS location.

For every user, we use routeInfo function (line 18) to calculate the distance (in kilometers) and travel time (in minutes) to the given location when the location of the user is known.
CHAPTER 5. CASE STUDY: SERVICE ENGINEER

getEngineers :: Coordinates -> Task [EngineerChoice]
getEngineers location
  = get currentDomain
    >>= \domain -> usersOf domain
    >>= \users -> get engineerLocation
    >>= \locations -> mapTask (getRoute location locations) users
where
  mapTask :: (a -> Task b) [a] -> Task [b] | iTask a & iTask b
  mapTask _ [] = return []
  mapTask func [x:xs]
    = func x
    >>= \result -> mapTask func xs
    >>= \other -> return [result:other]

getRoute :: Coordinates [(String, Coordinates)] User -> Task EngineerChoice
getRoute location locations user
  = case [ loc \ (u, loc) <- locations | u == (toString user) ] of
      [loc:_] -> routeInfo location loc >>= \{(Route|distance,traveltime) -> return
        { EngineerChoice| engineer = user
          , distance = Just distance
          , traveltime = Just traveltime }
      _ -> return { EngineerChoice| engineer = user
        , distance = Nothing
        , traveltime = Nothing }

5.6 Selecting engineer

The selectEngineer task is listed in Figure 5.11. The task has an argument the address of the ticket and obtains the GPS coordinates from the address (line 6). The task then obtains all the engineers using getEngineers (line 8) that have as argument the location of the address. Then the user can select an engineer. Figure 5.12 shows a screenshot of the service engineer selection screen where the user Bob is selected.

selectEngineer :: Address -> Task User
selectEngineer address
  = getCoordinates adres
    >>= getEngineers
    >>= enterChoice "Select an engineer" []
    >>= \{EngineerChoice|engineer} -> return engineer

Figure 5.10: Get engineers and the travel time and distance to GPS coordinates

Figure 5.11: Select an engineer
5.7 Working on a ticket

The task `workOnTicket` (Figure 5.14) allows the service engineer to work on a ticket. The task has three steps:

1. The first step is displaying the location where the service engineer needs to go to (line 3 until 9). We do this by first obtaining the GPS location of the address that was given in the ticket with the task `getCoordinates` (see section 5.3).

   Then we show the address information and a Google Map that displays the location of the address. We do this by giving some Google Map settings (line 15 until 36).

2. The second step is doing the machine repairing. We defined the type `Repair` (Figure 5.13). Because we use this type the iTasks framework generates for us the user interface (line 11).

3. The last step is returning the original ticket and the repairing information (line 12).

```plaintext
:: Repair = TextLine Text
         | PartLine Part
         | CostLine Cost

:: Part = { sku :: String, count :: Real, price :: Real }
:: Text = { note :: String, type :: TextType }
:: Cost = { cost :: CostCode, price :: Real }

:: CostCode = Investigation
         | Installation
         | Other OtherCost

:: OtherCost = { description :: String }
:: TextType = VisableForCustomer | Internal

derive class iTask Repair, Part, Text, TextType, CostCode, OtherCost, Cost
```

Figure 5.13: Repair type.
workOnTicket :: Ticket -> Task (Ticket, [Repair])
workOnTicket ticket=:{Ticket|address}
>>> coordinates -> getLocation
>>> myCoordinates -> (viewInformation "Machine, location" [] adres)
||- viewInformation () []
   { GoogleMap| settings = mapSettings
   , perspective = (mapPerspective coordinates)
   , markers = (mapMarkers coordinates myCoordinates)
   }
>>> [OnAction (Action "Start, repairing" []) (always (viewInformation "Details" [] ticket)
       ||- (repair [] >>> \result -> return (ticket, result))))

where
mapSettings = { GoogleMapSettings| mapTypeControl = False
   , panControl = True
   , zoomControl = True
   , streetViewControl = True
   , scaleControl = True
   , scrollwheel = True
   , draggable = True
   }
mapPerspective (LatLon lat lon)
   = { GoogleMapPerspective| type = ROADMAP
       , center = {GoogleMapPosition| lat = (toString lat), lng = (toString lon)}
       , zoom = 13
   }
mapMarkers coordinates myCoordinates
   = [ toMarker coordinates Nothing ] ++ case myCoordinates of
       (Just cords) -> [ toMarker cords (Just (GoogleMapSimpleIcon "van.png")) ]
       _ -> []
toMarker (LatLon lat lon) icon
   = { GoogleMapMarker| markerId = "customer"
       , position = {GoogleMapPosition| lat = lat, lng = lon}
       , title = Nothing
       , icon = icon
       , infoWindow = Nothing
       , draggable = False
       , selected = False }

Figure 5.14: Task that allows the service engineer to work on a ticket.

Figure 5.15 shows a screenshot of step 1 on the tablet of the service engineer. The service engineer can see where the machine is located and where he or she should travel to. The map displays the location of the machine.

Figure 5.16 displays the machine repairing step. The service engineer can now enter parts that are used to repair the machine, add text lines and costs. All the fields that are visible are generated by the iTasks framework from the Repair type definition.
CHAPTER 5. CASE STUDY: SERVICE ENGINEER

Figure 5.15: Screenshot of the ticket location screen for the service engineer, ticket information from Figure 5.6.

Figure 5.16: Screenshot of the ticket location screen for the service engineer, ticket information from Figure 5.6.
5.8 Conclusion

This very simple service engineer application shows a very limited service engineer application that does not consider features like working times, brakes and preventing that a service engineer has too much work to do. The goal of the case study was not to write a complete application but to show a larger application that is distributed with the distributed extensions for the iTasks framework.

We also hope that one can see that adding the missing features is not very complicated and instead just a matter of extending types and changing some functions to support this.

The goal of this application was showing that it is now possible to write an iTasks application that allows users to work on multiple devices that may go temporarily offline. With the non-distributed version of iTasks an almost equivalent application can be written except that it will lack support for working (temporary) offline.

- Line 6 \((\text{engineer } \theta, \text{ domain } \theta: (\text{workOnTicket ticket}))\) from Figure 5.5 makes the task distributed. Together with the same kind of structure line 6 from Figure 5.3, this is the only distributed specific code.

- In the distributed version of iTasks, there is an iTasks app that supports obtaining the location direct (without the need for WebAPIs). However, for obtaining the location the distributed version is not needed because the not-distributed version also supports this using a WebAPI (see BasicAPIExamples in the iTasks-SDK examples folder). When we want to access for example the Bluetooth stack we cannot use WebAPIs. So, this is requiring the distributed version of iTasks with local controllers.

- In this example it is new that it is not required for the service engineer to be connected permanently to the iTasks server because he can work offline on the machine. So, repairing a machine at a location where is no mobile and Wifi connection is now possible were in the non-distributed version of iTasks the service engineer has to write all the actions down and enter them later when the tablet was online again.

These changes to enable the distributed version of iTasks will cause that the following features are available for the service engineer:

- Working offline on a tablet with the local controller. Recall that in the non-distributed version of iTasks it is impossible to make progress without an internet connection. The service engineer now use an app instead of the browser on the tablet.

- Using sensors and hardware of the tablet. In the non-distributed version of iTasks it was possible to access the GPS location using WebAPIs. This, however, requires the user to give permission every time. The local controller inside a app does not require this. We can now also access the Bluetooth stack, for example, to communicate with some dialogic device.

- Every service engineer has its own local controller on a tablet, so the server does not have to process their requests. This allows us to support more users on the system than in the non-distributed situation. When it is needed, we even support multiple servers when there are users that use a browser as a client.

- Some functions, like getting the location, are performed in native Clean. It can also be done for more complicated calculations to gain speed.
Chapter 6

How to use the distributed version

To build a binary for different platforms like Windows, Linux and Android, requires multiple compilers. To help readers of the paper to experiment with the distributed version of iTasks we created a set of cross-compilers that help to compile for different platforms.

The cross-compiler\(^1\) repository contains compilers to build images for different platforms from someone source code. For Linux, Windows, Raspberry Pi and Android the Docker image is recommended for targeting these platforms. For MacOS, there is a script to build the compiler for MacOS.

6.1 Docker

When somebody wants to build an iTasks program for example for Android one need to perform the following to be able to compile for Android:

1. Install Java SDK 8
2. Install Android SDK
   • Install Android platform tools 22
   • Install Android-23
3. Install Android NDK

When all the preparation is done, the following steps need to be performed to compile an iTasks program for Android.

1. Prepare the cross-compiler for Android using `sh make-target-android.sh`
2. Build the cross-compiler for Android using `cd target/android ; sh build-target-android.sh`
3. Initialise the Android compiler `cd target/android/clean ; make`
4. Compile the iTasks project `/path/to/target/android/clean/bin/cpm <project>.prj`

We created a Docker file that can be used to create a Docker image. The Docker image will perform all the steps that are listed above automatically without the need for the user to have any knowledge about Android development. The docker image supports more targets than Android:

- Linux
  - Linux 32-bit

\(^1\)https://gitlab.science.ru.nl/distributed-itasks/cross-compiler
– Linux 64-bit
  • Raspberry Pi (32-bit)
  • Android
  • Windows 32-bit

6.2 MacOS

The cross-compilers repository has also support for MacOS. The compiler for MacOS can be obtained by running the `make-target-macos.sh` script.
Chapter 7

Conclusion

In this master thesis and in the submitted paper for the post-symposium proceedings of IFL 2016, we presented a distributed version of the iTasks framework that aims to solve the following problems:

- Allow multiple servers to solve the single server bottleneck.
- Gain speed by executing fast Clean code on the client.
- Support working offline.
- Support access to client hardware e.g. sensors or Bluetooth connections.

We solved these problems with the concept of (domain) controllers and local controllers, because controllers can take over work we solved the server bottleneck. Local controllers that run on the client solve:

- Working offline because the local controller running at localhost is also available if there is no internet connection.
- Access to hardware like the Bluetooth stack is possible because the local controller runs inside app permissions and can request for Bluetooth permissions.
- Local controllers are written in native Clean and this makes it possible to execute fast native Clean code.

In this master thesis, we gave more inside in the developed components that implement the idea of (local) controllers. Important aspects are reconnecting when the connection is interrupted, source code explanation about the proxy shares.

We believe that our work is interesting for everybody that wants to write multi-user applications that require people to collaborate and use modern devices like smartphones and tablets. We hope that our work also helps in distributing iTasks even further like smartwatches and smart home devices.

We hope that our work also inspires further work. We would like to drop the requirement that all the code (e.g. image) needs to be available on the device. This is also known as eager linking. Instead, we would like to also send dynamically the needed code the device like with SAPL does e.g. use dynamic linking.

All of the source code is available at https://gitlab.science.ru.nl/distributed-itasks. The examples repository contains the examples from the paper and the service engineer repo contains the service engineer application and build instructions.

The distributed version is at this moment a fork of the iTasks framework. The fork of the iTasks framework is very easy to merge it back into the iTasks system because the distributed part is developed as an extension for the iTasks framework. To give an idea, merging the nit-reorg
branch (currently in development version of iTasks) with the distributed branch did not give any merge errors. This was mainly due to the changes needed in the core of iTasks to create a proposal (e.g., merge request) for the iTasks system.

Developing the distributed version of iTasks was something that could be added relatively easy in the iTasks framework. This was mainly because iTasks has great support for developing extensions. Concepts that are in most applications core functions like, for example, authentication is in iTasks framework an extension.

Another reason is that making iTasks distributed is a pure implementation problem. Conceptual we did not change anything for TOP to work distributed. The concepts of (domain) controller and local controller are pure for the implementation.

We spend some time to find a good generic solution of the concept of controllers. In the version (presented at IFL 2016) we used the idea of a root server and normal servers and private server. In this concept there was no global task pool. We solved this in the first place by adding roles to servers like instance role and authentication role. This started to become complicated and forced us to generalise the problem. With this process, we lost some time had to rewrite the implementation.

If we had to do this project again, we would still take this approach, working incremental and then improving and generalising the result worked very well. It helped us to see what the consequences are for making iTasks distributed. At the same time, the source code improved and became simpler due to the rewritings to a more generic solution.

We have now a generic solution that allows us to write applications like the service engineer application. This application is written without any change in the distributed version of iTasks. This was not the case with previous implementation where the small chat example in the paper already lead to changes that we had to make in the implementation.
Bibliography


