The following full text is a publisher's version.

For additional information about this publication click this link.
http://hdl.handle.net/2066/195231

Please be advised that this information was generated on 2019-10-21 and may be subject to change.
A Distributed Dynamic Architecture for Task Oriented Programming

Arjan Oortgiese
Institute for Computing and Information Sciences, Radboud University Nijmegen
Nijmegen
arjan@arjanoortgiese.nl

Peter Achten
Institute for Computing and Information Sciences, Radboud University Nijmegen
Nijmegen
peter88@cs.ru.nl

John van Groningen
Institute for Computing and Information Sciences, Radboud University Nijmegen
Nijmegen
johnvg@cs.ru.nl

Rinus Plasmeijer
Institute for Computing and Information Sciences, Radboud University Nijmegen
Nijmegen
rinus@cs.ru.nl

ABSTRACT
Task Oriented Programming (TOP) is a special flavor of functional programming for real-world application domains in which people and automated systems collaborate to achieve a common goal. The original iTasks framework, which implements TOP, uses a single server multi-client architecture. This is not suited for truly distributed application domains, such as deployed by the Dutch coast guard. In this paper we show how to turn this architecture into a distributed, dynamic architecture. This is done in an elegant way, by building on the core concepts of TOP and iTasks.

CCS CONCEPTS
• Software and its engineering → Functional languages; • Computer systems organization → Distributed architectures; • Human-centered computing → Computer supported cooperative work;

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

1 INTRODUCTION
Task-Oriented Programming (TOP) is a special flavor of functional programming. TOP focuses on application domains in which people collaborate in a distributed manner to accomplish a certain goal. Typical examples of such applications can be found at the Dutch coast guard. The Dutch coast guard monitors vessel traffic on the North Sea, which is one of the busiest waterways in the world. One monitoring duty is to analyze the sailing route and behavior of a vessel, using information from the past, to determine whether further action is required. This requires the coordination of computation and human decisions how to proceed. Another Dutch coast guard duty is performing Search and Rescue actions [12]. Here teams collaborate to rescue people, and therefore everybody constantly needs to be informed about the latest state of affairs of all involved teams. These kind of systems are dynamic: circumstances change, communication is temporarily impossible, the situation requires down- or up-scaling, new information is available and needs to be taken into account, and so on. This is not limited to the Dutch coast guard: one can think of the government, police, and hospitals.

The iTasks framework [1, 15] implements TOP as an Embedded Domain Specific Language, using the pure, lazy, strongly typed, generic, functional programming language Clean as host language. In Clean one can define type driven generic functions [13]. This feature is used to enable TOP in iTasks: for any first-order data type that arises when modeling the application domain, code is generated to deal with (de-)serialization, rendering GUIs, transferring data between tasks and shared data sources, and so on. TOP has a small core [16]. There are only three core concepts in TOP: tasks, editors, and shared data sources, and a monadic, applicative combinator library featuring two core task combinators: step and parallel. A task is a type-parameterized abstraction that represents any piece of work performed by (a combination of) people and computerized systems that may occur within the application domain. Tasks that interact with end-users are called editors. The task uses its type-parameter to expose its current progress via its task value that changes over time. It is essential to understand that this abstracts from what is really going on within the task, after all, it might be a human performing the task or a computerized system or any combination. Tasks are observable: any other task can look
at the task values of the tasks that it depends on and use this information to decide if and how it affects them. This dependency is defined via (a combination of) the task combinators (tightly) and shared data sources (loosely). The step combinator coordinates a dynamic number of conditional tasks to observe a task and decide with which conditional task to proceed, if applicable. The parallel combinator coordinates a dynamic number of sibling tasks that can observe each other’s task value. Shared data sources, finally, make the information sources at organizations accessible for tasks. Shared data sources can also be created dynamically.

Consequences of the above design and implementation decisions are that TOP programs are highly modular. The abstraction imposed by the task value hides the implementation of a task. This allows replacement of any task by another that has the same type and policy of task values. A human processed task can be transferred to another human processor in case of illness, holidays, or work-overload. Task value changes abstract from the fine-grained granularity of typical event-based systems: if a task processor (human or machine) is temporarily failing, it can always ‘catch up’ with a different task value change. These are important requirements when modeling real-world domains such as the ones described above. TOP offers these tools to create robust and realistic applications.

The current iTasks framework implements TOP as a single server, multi-user, web-based application. It works with any modern browser and thus works on mobile devices such as tablets or smart phones. This is fine under circumstances in which a reliable, sufficiently fast, internet connection is always available, and in which all shared data sources can be stored and maintained on the server. However, these circumstances are not to be expected for all application domains. The current architecture shares the following drawbacks with any single-server based web application: (1) The central server is a bottleneck and it allows the proper evaluation of tasks on a wide range of devices. In this topology of devices, arbitrary tasks can be pushed from one device to the other, and arbitrary tasks can be pulled. This bypasses the single server bottleneck and it allows the proper evaluation of tasks on many devices. The coordination of these ‘multi-servers’ is done via domain controllers and local controllers. Domain controllers partition the end-users and their tasks. Local controllers run on client-devices and thus allow working offline (for a while) and reducing the workload of the domain controller from which they retrieve tasks.

- In order to bypass the limitations of JavaScript running in a client browser, we additionally compile Clean code for ARM processors. This unlocks more efficient code on the client device, as well as the opportunity to use local device capabilities, such as camera systems and sensor data. The latter is achieved by means of an Android App that can run any task on its device.
- Shared data sources are re-implemented such that their value can be inspected locally and remotely. With these new shared data sources, proxies to any remote task can be created to implement remote, observable, tasks.

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.

The remainder of this paper is organized as follows. First we give a short overview of the most important TOP concepts in section 2. In section 3 we show what has been done to send any closure to be evaluated remotely from one platform to any other. Section 4 presents the extended iTasks architecture that uses domain and local controllers, and remote shared data sources to obtain the desired system. Related work is discussed in section 5. Conclusions are drawn and future work is discussed in section 6.

2 TOP / ITASKS IN A NUTSHELL

In this section we introduce the core concepts of iTasks. Next we give an example of an iTasks specification, showing how a chat application can be defined that works for an arbitrary number of end-users and that abstracts over how and which information is exchanged between the end-users. Hereafter we explain the major components of the current iTasks run-time system.

2.1 Core Concepts of TOP / iTasks

2.1.1 Tasks. Tasks are created by functions returning a value of type \( \text{Task } a \rightarrow \text{iTask } a \) for some concrete type \( a \). The type \( a \) can be any Clean type, under the context restriction that an instance of class \text{Task} exists for \( a \). This \text{Task} class consists of a number of generic functions \([8, 13]\) which are needed to evaluate tasks on servers and clients, such as functions to serialize, de-serialize, compare values, and create GUI’s. These generic functions can be automatically derived for any concrete first order type. So, the context restriction \( \text{Task } a \) is not a real restriction in practice.

Tasks are Clean functions defined in a special recursive way \([16]\). As a result, in contrast to ordinary functions, a task of type \( \text{Task } a \rightarrow \text{observable } \mathcal{V} \) of type \( a \) that may change over time. The task value reflects the current status of the work. When no meaningful information about the current status can be given, \text{NoValue} is used (for instance, no work has been done yet, or when it is decided to start over from scratch). It can be unstable (Value \( v \) Unstable) when one is still working on a task such that the current value \( v \) might change in the future. It can also become stable (Value \( v \) Stable) when the work is done and value \( v \) no longer
changes. All the time the status can be observed by the system and its current value might influence what others can see and do.

2.1.2 Editors. An editor is a basic task of type :: Task a. Editors make use of type driven generic functions such that the system is able to automatically generate a user interface in the browser for any concrete first order type a. With the generated interface end-users can only manipulate values of the required concrete type a. Examples of predefined editors are: enterInformation to enter a value of type a, updateInformation to change a given value of type a, or viewInformation to display such a value. For example, the default rendering of a record type is a browser form for filling in the fields (e.g. when the type is :: String, enterInformation shows an input field and the observable value of the task is the text entered in the input field). The user interface of an editor can always be customized. For instance, for the type :: Picture one can define a picture-editor, and for the type :: GoogleMap a map-navigator.

2.1.3 Task Combinators. The iTask library offers a rich set of Task Combinators, built on top of only two ‘Swiss army knife’ combinators: one for sequencing and one for the parallel case [15].

The sequencing combinator, called step and denoted as ++>, is a choice combinator. In ta ++> [onAction a1 p1 atb1, . . . , onAction an pn atbn] :: Task b the task ta :: Task a is being observed. The actions ai in the list (note that this list can be computed on-the-fly) are converted to buttons on the screen. When button ai is pressed by the end-user and predicate pi holds, where pi is the current value of task ta, that task is ended and one continues with task (atb1, v) :: Task b. There are also simple variants of this sequential operator, like a monadic bind >>*, return, and >> (Haskell: >>) [15].

The parallel combinator coordinates a dynamic collection of sibling tasks that can observe each others’ task values. There are several derived combinators for frequently occurring, less dynamic, parallel patterns. With the parallel and operator, ~&~<, two tasks ta and tb can be started in parallel, the resulting task returns a tuple combining the results of both tasks, so ta ~&~< tb :: Task (a, b). With the parallel or operator, ~|~>, the value of the task which has become stable first is returned, so ta ~|~> ta :: Task a. Frequently used variants of this operator are ~|~ (only return the current value of the first task, ignore the second), and ~|~< (the other way around).

One-and-the-same end-user may work on several parallel tasks at the same time. An important aspect of a parallel task is that it can become stable first is returned, so ta ~|~> ta :: Task a. Frequently used variants of this operator are ~|~ (only return the current value of the first task, ignore the second), and ~|~< (the other way around).

2.1.4 Shared Data Sources. Although tasks can observe each others’ progress with the step combinator, and information can be passed from one task to another, one needs to be able to share arbitrary global data between tasks as well. One can think of global information stored in data sources like memory, files and databases, data like information produced by sensors, the current date and time. For any type of data that one wishes to share, one and the same abstract interface is provided, called a Shared Data Source or SDS [5]. So, an SDS can be shared data stored in a file, in a relational database, or temporarily stored in main memory. An SDS is used internally to administrate the users who can login, the current user working on a task, and so on. Like tasks, SDS combinators are provided to create more complex SDS’s from simpler ones. In the current iTask architecture, SDS’s are only available on the iTasks server and cannot be put on a client.

The following atomic, basic operations are defined: get, set, upd, watch. With get the current value from the SDS is retrieved; set writes a new value to the SDS; upd uses a higher-order function to update the current value of the SDS; watch turns an SDS into a task that emits unstable task values reflecting the current SDS value.

Editor tasks (section 2.1.2) can be ‘connected’ with an SDS. For any first order type a user interface can be created, also when the value is stored in an SDS. To view the content of an SDS one can use viewSharedInformation. The content can be updated with updateSharedInformation where the SDS is updated with every change that is made by the end-user in the user interface shown in the browser. The core parallel task combinator (section 2.1.3) utilizes SDS’s to inform the parallel child tasks of the (continuously changing) task values of themselves.

Hence, it is important that whenever someone changes an SDS, all tasks that are currently observing that SDS are automatically informed about the change that is made. For example, if an end-user changes the content of an SDS via updateSharedInformation, all other users who look at the value via viewSharedInformation will see the changed value as well, albeit a bit later due to the latency of the internet. Every SDS is administrated at run-time in the publish-subscribe system of iTasks where is managed which tasks need to be informed when something is changing.

2.2 A dynamic, customizable, chat application

As a running example we present an iTask application that enables an arbitrary number of people to chat with each other. We define it in a general way so that we can easily modify it later in this paper. The entire specification is given in Figure 1. It is not important to understand all the details because we want to give you an idea about what task definitions look like. We will use it later on to explain what the consequences are for a distributed architecture.

The task function createChatSession takes two tasks, enter and update, as argument. The enter task can be any task producing a value of arbitrary type :: a. With the update task this information is turned into some convenient display format :: b to be stored in an SDS of type :: [b], used to display the chat history to all participants. First the current end-user is determined (4), ne, who wants to start a chat session. Next, using the bind combinator >>>, it asks this user to select the others to chat with (5–6). The enterMultipleChoiceWithShared task let the user select one of the elements in the users SDS (User) using a pull down box. The predefined SDS users from which the others are chosen, contains a list of all users currently known in the system. Hereafter, a new SDS of type :: [b] is created in shared memory to store the history which is initialized with an empty list [] (7). This shared list is passed to the startChats task together with the enter and update task and the list of users participating in the chat session.

The task function startChats assigns (using @:) to every selected person the chatWith task, using the parallel combinator allTasks (13–15). When all participants have ended their chat session the chat
history stored in chatStore is returned at the end of the conversation (15).

In chatWith, a person sees the messages entered so far, due to viewSharedInformation (20). At the same time (1|->), she can enter a new message with oneChat (21). The oneChat task uses the higher order task function enter to let the user enter a new chat message. Two buttons are created, Send and Quit. When Send is pressed, the newly created message is converted with the update task (27), and added to the chat history stored in the SDS chatStore, after which another message can be entered due to the recursive call of oneChat. When the chatStore is updated, all persons see the new message due to their viewSharedInformation on this SDS. When Quit is pressed, the user exits the chat and returns the updated history.

We can now proceed to create a custom instance (Figure 2). To inform all persons who created what message at what time, we introduce a new type :: Msg a that keeps track of this information for messages of type :: a. The task function chatExample1 creates the custom instance. For entering messages, the enter task uses the generic enterInformation editor task. Note that the kind of messages that can be entered is determined by the type of chatExample1 only. For this example we use String. These are converted to Msg String with the update task function.

Note that all the user interfaces and low-level event handling code is generated from the chat definition above. When a new message is entered by someone, it is stored in the SDS and all users automatically see the updated history.

To demonstrate the generic and customizable nature of the specification, we define an alternative version in which, at any time, any person involved in the chat session, can start a new chat session with new people and have the extra conversation added automatically to the ongoing conversation. The specification is shown in Figure 3. We first introduce a new domain type, ChatOptions, to reflect the choice every person has and generate the necessary generic instances. We only alter the first higher-order parameter of createChatSession. This new task, enter2, offers the current end-user a choice between entering a line, as before, resulting in the task value. The effect of this relatively small change in the specification is that whenever the new conversation has terminated, all of its content is automatically added to the ongoing conversation.

2.3 Standard iTasks Run-Time System

The standard iTasks run-time architecture consists of a single server that serves multiple clients. These clients are commonly browsers that end-users use to perform their tasks.

2.3.1 The iTasks Server. The core components of the iTasks server are the following. The task pool administration contains a list of MSG a = {time :: Time, user :: User, message :: a} deriving iTask Msg

Figure 2: An instance of the generic chat specification

Figure 3: Invoking chats recursively

createChatSession :: (Task a) (User a -> Task b) -> Task [b] | iTask a & iTask b

createChatSession enter update

= get currentName

>> \ me -> enterMultipleChoiceWithShared ["select_people"] [] users

>> \ others -> withShared []

= withChat enter update [me:others]

startChats :: (Task a) (User a -> Task b) [User] (Shared [b])

= allTasks [ (person, "chat") @:

| person <- people ] >> get currentUser

= viewSharedInformation ("Chat␣History:") [] chatStore

| oneChat

where

oneChat = enter

>>> [ OnAction (Action "Send") (hasValue send)

, OnAction (Action "Quit") (always (return [])) ]

send new_msg

= update me new_msg

>>> \ new -> upd (msgs ++ [new]) chatStore

>>| oneChat

Figure 1: The dynamic, customizable, chat application
of all the tasks someone can work on. It can be regarded as a to-do list. When a new parallel task is created for someone to work on, it is added to this pool together with the task attributes. The set of attributes is not fixed but can be freely defined by the programmer. Common attributes are: the title of the task, 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. The task instance administration keeps track of the task instances of tasks 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, the owner of a task may change, however (e.g. someone else 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 SDS administration contains which tasks currently depend (subscribed) on which SDS’s. This is used to inform those tasks when an SDS changes. A web server is built in, 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 SDS’s, TCP connections and the like to coordinate the tasks.

The iTasks system contains several plug-in components as extensions, 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 commonly HTML 5 compatible browsers. When a browser makes a connection with the server, initially a small client-side application written in JavaScript is loaded. The client sends client-events such as changes made with an editor or OnAction buttons pressed to the iTasks server that processes its responses. The response may result in a dynamic update of (a part of) the HTML code, or a new piece of JavaScript code needed for the evaluation of a specific closure on the client. Hence new JavaScript code is automatically added to the client code pool when needed. It consists of just-in-time compiled Clean code needed for the evaluation of a specific closure on the client. For that purpose Clean is compiled twice at compile time: once to native code (Intel) x86-64 code for Windows, Mac, and Linux platforms, and once to SAPL code [10]. SAPL is a core functional language that is dynamically just-in-time translated to JavaScript code. SAPL has as advantage that from the code one can relatively easily determine which functions are additionally needed for the evaluation of a given closure on a specific client.

This scheme eliminates the need to define browser computations in JavaScript, and use the pure and type safe Clean host language instead. We can define custom editors in Clean (see [4]), e.g. for making drawings in the browser, or working with maps, or for interfacing with JavaScript WebAPIs like getting the current location or access the camera of a client device.

3 DISTRIBUTED EVALUATION OF CLEAN CODE

As stated above, iTask programs are compiled to native (Intel) code and to JavaScript via SAPL. To unlock TOP for ARM-based platforms, such as Raspberry Pi and Android, we need to generate code for ARM processors as well. To realize distributed evaluation of Clean functions, two additional facilities are needed. First, the corresponding code needs to be available on the remote processors in the network. Second, any Clean application on any platform must be able to create any closure in a symbolic format such that it can be serialized, shipped, de-serialized, and remotely executed on any of the other platforms.

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.

ABC code is the intermediate code of the Clean compiler that is compiled in a later phase to machine specific code. One option is to ship the ABC code and let the remote machine perform the last phase of the compilation for its platform. We have decided to generate all images for all platforms at once when the application is being developed because it is easier in this way to guarantee that the corresponding generated images have been made from the same source code using the same compiler, yet different code generators and linkers. As said before, the Clean compiler is very fast, so the developer is not hampered by this. To give an idea: on a smart phone like the Samsung Galaxy Note 4 the Clean compiler compiles itself from scratch in 11 seconds.

Next we have to ensure that the required code is available on the (remote) machine before we ship a closure for evaluation to it. It is important to realize that statically it is undecidable what kind of closures are constructed. Because we are dealing with a lazy language, a closure might contain unevaluated function calls. So, one cannot know on beforehand which code is needed for the evaluation of a closure. Code can be distributed in different ways, eagerly or lazily. When shipped eagerly, all code, the complete image, is stored on the target machine in advance. With the image any possible closure received can be evaluated. Eager code distribution has as disadvantage that one perhaps does not want to show all code on the remote machine for security reasons. With the lazy shipment method only the code which is really needed for the evaluation of the closure is shipped. It has as disadvantage that one needs to be able to dynamically extend the running application (dynamic linking). For each closure one has to determine which code, not present yet, needs to be shipped and added to the application.

Currently, for x86-64 (Intel) and ARM processors the eager code distribution approach is used. One has to ensure manually (download or upload) that the image is indeed available on the remote machine. For browsers, code is shipped lazily to the browser on demand using push technology. It should be noted that for Windows a dynamic lazy linking facility already exists, via Clean Dynamics.
We intend to add a lazy shipping and linking option for the other platforms and operating systems as well.

Of course, no matter how code is distributed, proper security measures have to be taken to ensure that the code can be trusted, as usual. This is beyond the scope of this paper.

### 3.2 Distributed Evaluation of Functions in Native Code

To send a closure for evaluation from one platform to some other platform means that we have to be able to serialize any function or expression at any time for 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 sent back in the same way as well.

Using native code has the drawback that serialization and deserialization 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 layout, and heap layout. At run-time therefore a closure has to be reconstructed symbolically from the actual content of the stacks and the heaps, serialized, and on the other machine the information has to be stored in the right way such that evaluation can be done as usual. Closures 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 at another platform.

For every platform we have implemented (de-)serialization operations that work for any closure of any type. These operations use the symbol table of the executable to look-up the memory addresses of the descriptors. One 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 and reconstruct the function call in such a way that it can be evaluated on the platform.

When generating code, one can choose to generate code for 32-bit or 64-bit machines. We provide the ability to mix these formats, and translate the data of 32-bit machines to a 64-bit format and vice versa. Of course, one has to be careful because down scaling a 64 bit integer to a 32 bit integer can create incorrect numbers, and causes 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 distributed iTasks system. Instead of having one central server, we support an arbitrary number of iTask servers, called controllers from now on to avoid confusion, since these new iTask servers can run on clients as well as on servers. Actually, an arbitrary network topology of controllers can be made which may change dynamically over time. All controllers are programmed in Clean, and can therefore run on any platform, i.e. any Intel or ARM processor. We assume that all devices on which controllers are running have all the necessary code at their disposal and are able to evaluate any closure shipped to them for evaluation (see Section 3). Hence we focus on the issues related to the distribution of iTask controllers and iTask tasks.

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 (Section 4.1) and explain what the consequences are for iTask applications in this setting and how the distribution of controllers affects the implementation. To be able to evaluate tasks in a distributed fashion, we have to deal with the fact that an SDS can now be located on some other machine instead of the same central server (Section 4.2). The evaluation of a task on another device differs from the evaluation of an ordinary Clean function because the corresponding task value has to be made observable by the shipping controller (Section 4.3). We show how to reduce the workload of a serving device by splitting the task coordination over several serving controllers (Section 4.4). We introduce local iTask controllers (Section 4.5). They allow to work offline on a tablet or other device. We show examples of tasks running in a distributed fashion on a network of (domain and local) controllers running as Android app, or located on a Raspberry Pi (Section 4.6). Finally, we discuss the consequences of having a distributed architecture versus a centralized one (Section 4.7).

#### 4.1 Distributed Domain Controllers

Domain controllers for iTasks are inspired by domain servers for handling e-mail traffic. The iTask setting is more complicated because working on a task can have direct effects for other tasks of other users located anywhere, while answering an e-mail has no global effect at all until the e-mail is actually sent. 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. Domain controllers can still be added and removed dynamically. For usability we assume that all the domain controllers have a domain name that can be resolved using DNS.

2. There is at least one domain controller in any iTask network. If there is only one domain controller, the architecture behaves just like the old standard configuration with one central server. The new architecture is a real extension of the old one.

3. As is the case with an e-mail server, a domain controller serves a group of users who are administrated in that domain and therefore can login into the controller. A domain controller takes care of the authentication process. By default the user and roles administration of iTasks is used that stores users and roles in a SDS. However, any other back-end such as an LDAP server or Single Sign On server [3] can be used. A server has to support two operations. First, at authorization, given a user name and password, results in a unique user identification, including the roles the user can fulfill.
Second, it must provide a list of all currently administrated users, which can be used to (interactively) select a user to assign a task to.

(4) 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 e-mail addresses. The operator \( \theta \) combines them user \( \theta \) .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.

(5) 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. When a controller fails the controller can be restarted. Information of all users, all tasks, and all tasks instances are constantly stored in persistent memory, no information is lost when a machine on which a controller is running, has stopped execution. When the controller is restarted, a user can continue the work by re-opening the task.

We generalize the chat example of Section 2.2 such that we can chat with any user administrated in any domain known in the network. We only need to alter one line of code in the task function `createChatSession` (Figure 1), all other code of the previous chat example remains the same. The call to `enterMultipleChoiceWithShared` has to be replaced by a call to `selectUsersFromDomain` (Figure 4). It calls the recursive task `select` that accumulates a list of persons selected from the domains. The SDS domains (6) contains the names of all known domains. The function `usersOf` (8) returns an SDS containing the names of all users administrated in the given domain. The end-user who wants to start a chat session recursively selects a domain (6), and the users to chat with from that domain (7–8). Next the end-user can decide by pressing `add` to add more people, or to stop the selection process by pressing `done`.

Although the iTask specification is changed only in one place, the possibility that there are several (domain) servers has quite some consequences for the implementation. Consider Figure 5 where we assume a small distributed network configuration, consisting of two domain controllers, for the domains A and B. Assume that Alice on domain A initiates a chat session with Dave on domain B. The chat-history SDS of type \([\text{Msg String}]\) which is displayed to both persons is located on the domain controller of Alice. So, the task shipped to Dave running on controller B needs to have access to the SDS remotely located on A. Moreover, if Dave stops the chat session, this will have consequences for the parallel task allTasks (Figure 1) located on A. Hence, the current value of tasks need to be observable on other controllers.

### 4.2 Accessing Remote Shared Data Sources

In the original setting, all SDS’s are 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 locally stored on that device. We call 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, remote versions of the basic SDS operations `get`, `upd`, and `watch` (Section 2.1.4) are implemented as `r_get`, `r_upd`, `r_set`, and `r_watch`. The programmer does not need to be aware where an SDS is located: at run-time an application of `get`, `upd`, `set`, and `watch` is automatically redirected to a call of `r_get`, `r_upd`, `r_set`, and `r_watch` in the case of a remote SDS. The `r_` versions 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.

To access a remote SDS we use the (de-)serialization mechanism (Section 3.2) to send a serialized closure to the SDS-host, and let it de-serialize the closure and apply it to the SDS. This allows us to access any SDS no matter where and how it has been remotely created and stored. It also works for parameterized SDS’s [5], SDS projections which allow a task to access a specific part of an SDS enabling a more fine grained and more efficient access to SDS’s.

The result of the remotely applied closure is sent 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 ensures that the operators are applied atomically to the latest value of the SDS, so we do not have to deal with synchronization and version conflicts.

The `r_watch` operation is handled differently, because a straight adaption of the above scheme for this operation results in a lot of network traffic and unnecessarily blocks the requesting controller. Instead we implement a `notify_me` request which evaluates 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 `r_get` and store a copy in a new local

---

**Figure 4: Selecting domains and users**

Known domains. The function `usersOf` (8) returns an SDS containing the names of all users administrated in the given domain. The end-user who wants to start a chat session recursively selects a domain (6), and the users to chat with from that domain (7–8). Next the end-user can decide by pressing `add` to add more people, or to stop the selection process by pressing `done`.

Although the iTask specification is changed only in one place, the possibility that there are several (domain) servers has quite some consequences for the implementation. Consider Figure 5 where we assume a small distributed network configuration, consisting of two domain controllers, for the domains A and B. Assume that Alice on domain A initiates a chat session with Dave on domain B. The chat-history SDS of type `[Msg String]` which is displayed to both persons is located on the domain controller of Alice. So, the

---

**Figure 5: Two mutually connected domain controllers.**
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 explained how any Clean function can be serialized, shipped to another processor, de-serialized, and evaluated. However, if we want to evaluate a task function remotely, it is not enough to send over the task function. Also all instances of the generic iTask functions specifically generated for the task function are required as well. For this purpose we define the following container type Remote_Task for shipping a task with its generic functions:

\[
\begin{align*}
\text{:: Remote_Task} = & \text{E.a: Remote_Task (Task a) TaskAttr InstanceNo & iTask a} \\
\text{| NoTask}
\end{align*}
\]

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 the class Task a is automatically added to the constructor of the ADT when a value of this type is created. Hence, when such a 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 that cannot be de-serialized to a proper task as described, the container is not accepted. This can only happen when a container is corrupted somehow due to external communication failures.

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, emits observable task values that change over time (see 2.1.1). If observed by for instance the step task combinator, \( \gg \), this implies that the shipping controller must be able to observe the current value of the remote task.

To support this, we create two special tasks, a proxy task on the sending controller and an evaluating task on the remote controller. The proxy task maintains a local copy of the task value in an SDS. The remote evaluator maintains the current task value and an instance of the task to be evaluated in another SDS. Every time the remote evaluator evaluates the task due to an action of the end-user, it may change the task value. The changed task value is communicated to the sending controller to store it in the SDS of the proxy task for synchronization. In this way other tasks in the sending controller can observe the task value of the remote task.

4.4 Distributing Tasks Server-Side
The ability to have multiple controllers can be used to decrease the workload of a domain controller on the server. Another server-side controller can take over 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 it is willing to take over via a claim filter, which is a predicate of type \( \text{:: TaskAttributes -> Bool} \) that defines which tasks are wanted. The claim filter is sent 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 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, or resource. In this way, any algorithm can be realized to spread the workload over servers. 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.

\[
\begin{align*}
\text{claimUsersAM :: TaskAttributes -> Bool} \\
\text{claimUsersAM attrs = let name = readAttr "createdFor" attrs in} \\
\text{("a" <= name && name < "n") || ("A" <= name && name < "N")}
\end{align*}
\]

The domain controller can redirect end users after they have logged in to the address (e.g. URL) of the controller that is claiming their tasks. The work and users are divided over the controllers.

4.5 Distributing Tasks to Local Controllers on a Client
One can also have additional controllers on clients. Such a local controller might be located anywhere, on a pc, laptop, tablet, or smart-phone. An administrated user can log into a domain controller with a local controller and claim a subset of his or her tasks to be downloaded to the local machine. Claiming tasks to be handled locally is realized in the same way as described in section 4.4. The local controller can for example automatically pull all the task for the current active user. Another option is that the user manually chooses the tasks that need to be pulled to the local controller by

![Figure 6: Network with two domain controllers, each with a local controller attached.](image_url)
opening them. With a local controller one can work on tasks locally without disturbing the server side domain controller one is administrated on. This makes sense when a task requires a lot of work for which no or limited interaction with other end-users or systems is needed. Handling task events can be done faster because the controller is located on the same machine and one does not need to share the controller with other users. As long as no interaction with other end-users or remotely located SDS’s is needed, one can also work off-line.

A local controller is an iTasks server, 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 the specific end user logged in can be seen and claimed via the claim mechanism. One can subscribe for specific tasks, e.g. that need a camera.
- Any number of local controllers can make a connection to their domain controller. The domain controller has a local controllers administration to keep track of the currently connected local controllers. Local controllers can connect or disconnect themselves from the domain controller at any time. If a local controller is temporarily disconnected from the net, then, after re-establishing the connection, the deferred communication continues as if nothing happened.
- 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 a local controller can subscribe to the task pool of the domain controller. When new tasks are added to the domain controller, the local controller is notified. 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 remote SDS’s and its task value can be observed remotely.
- When a task is downloaded from a (domain) controller, it is stored in the task instance administration which local controller is working on it. This prevents other controllers or end-users to work on the same task. In iTasks only one person or system is allowed to work on a certain task at the same time.
- As in the original iTask system, it is possible to explicitly steal a running task. 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. The task instance can in general not be completely lost when a controller becomes unavailable for some reason. 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 e-mails are presented in an e-mail client. The same end-user can log in as many times as desired, directly on the domain controller, or indirectly via a local controller connected to a domain controller. In Figure 6 we show a possible set-up in which end-user Alice is logged-in twice. However, Alice is not allowed to work on the same task on both machines. If she starts to chat with someone on one of her machines, then this task is blocked on all others. Still, she can steal the task onto the other machine by asking the iTasks system to move it. The system warns her that local state information is lost. For the chat example this concerns only the current sentence that is edited in the browser. The chat-history SDS remains 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. CPU-intensive computations on the browser can also cause slow-downs 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.

### An iTasks Controller as Android app

An Android iTasks app consists of three components:

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

2. Additional functionality as Clean functions which are compiled to native ARM code stored in the app. We 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 [9]. 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. We can optionally include a browser component in the app, and connect it to the local controller. In Android we use the WebView component [7] that is part of the Android platform as a browser.
where

\[ \text{chatTextOrPicture} = \text{createChatSession} \ \text{enterPictureChat} \]

\[ \text{chatTextOrPicture} :: \text{Task} \ [\text{Msg} \ \text{TextPicture}] \]

\[ \text{enterPictureChat} \ n \rightarrow \text{Task} \ (\text{Maybe} \ (\text{Msg} \ \text{TextPicture})) \]

\[ \text{enterPictureChat} :: \text{User} \rightarrow \text{Task} \ (\text{Maybe} \ (\text{Msg} \ \text{TextPicture})) \]

Device.Location

DeviceInfo
Nothing
All other code remains the same.

(10–11) via the task

\[ \text{mkPict} n \rightarrow \text{Task} \ (\text{Maybe} \ (\text{Msg} \ \text{TextPicture})) \]

\[ \text{mkPict} :: \text{User} \rightarrow \text{Task} \ (\text{Maybe} \ (\text{Msg} \ \text{TextPicture})) \]

\[ \text{takePicture} >>= \text{maybe} \ (\text{enterPictureChat} n) \ (\text{newMsg} n \ o \ P) \]

\[ \text{chatTextOrPicture} :: \text{Task} \ [\text{Msg} \ \text{TextPicture}] \]

\[ \text{chatTextOrPicture} = \text{createChatSession} \ \text{enterPictureChat} \]

---

**Figure 7:** Chat example with support for sending pictures

As an example we have added a library, Device.Camera, that can be used to create a task with which an end user can take a JPEG picture using the device’s camera. The task function returns a Maybe Base64 that contains the picture encoded as Base64 string or Nothing in the case the end-user canceled the picture (e.g. by pressing the back button). A JPEG picture can be shown to the user using the viewInformation editor because there is an editor defined for JPEG images in iTasks.API.Extensions.Pictures.JPEG module. The Device.Location API allows users to share the location using the location service of Android and can 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.

**An iTask Controller on a Raspberry Pi**

Since a Raspberry Pi uses ARM code, we can also run an iTask controller on a Raspberry Pi. In this way we gain access to the resources available on small IoT devices such as the Raspberry Pi. By providing an interface to its resources, one can simply write tasks to be executed on the Pi. Interaction with the other tasks and SDS’s is obtained for free.

**4.6 Examples**

**4.6.1 Extended Chat Example on Android.** The first example extends the chat example of Section 2.2 with an option to take a picture. Figure 7 shows the code of the extended enterChat, enterPictureChat. 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 device SDS. This information is read (6), and used to find out if this device has a camera (10). The task function enterPictureChat extends enterChat with an action to take a Picture (10–11) via the task mkPict (13–15). The takePicture task activates the camera for taking a picture. If the end-user decides not to take a picture, enterPictureChat is called recursively to offer all chat options again. If the end-user has taken a picture then it is returned as Just (P p). The infix operator o is function composition. We use the altered createChatSession described in section 4.1.

This extended chat example can run on any iTask configuration. Figure 8 shows a screen-shot of Alice and Bob, where Bob has taken a picture with the camera on his Android tablet running an iTask App, and has sent this picture to Alice.

**4.6.2 Measuring Temperature on a Raspberry Pi.** In Figure 9 we show an example of a task that must be executed on a Raspberry Pi that is equipped with a temperature sensor connected to its GPIO pins (for the nitty-gritty Raspberry Pi details how that works we refer to [11]). On the Raspberry Pi, the measured temperature is stored in a file. However, the file system of the Raspberry is 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 the new distributed iTasks system it suffices to install a local controller on the Raspberry Pi, and ship tasks to it for execution.
With the distributed, dynamic, iTask architecture TOP programs which no progress is made. In the new architecture, a remote device when human processors are involved, tasks ensure progress by
ware systems), and instead emits (lack of) progress via observable
value. Because the TOP event model is defined only in terms of
nal system tasks need to be designed in such a way that they can
step task combinators, not alter the meaning of a program.

Tasks that exhibit lack of progress cannot hamper the progress of the task it is part of, unless this has been defined explicitly. Certainly when human processors are involved, tasks ensure progress by observing its sub tasks and provide a way out of a situation in which no progress is made. In the new architecture, a remote device might become temporarily unavailable. As soon as it is operational, it can resume its work and emit new task values. Again, this does not alter the meaning of a program.

Even though sub tasks might be executed remotely, the two core task combinators, step and parallel, that coordinate the compositional behavior of these sub tasks always run on a single controller in an atomic way. As a consequence, they will not behave differently from the original system. A key difference of the distributed, dynamic, architecture with the original system is that in the new system one can create arbitrary long latency. When designing a TOP application, the developer always needs to be aware of this. It must be emphasized that this is not new because also in the original system tasks need to be designed in such a way that they can handle situations in which end-users or tasks that refer to external

organizations do not respond in a timely manner. This also holds for tasks and shared data sources that have been downloaded on remote devices that fail permanently.

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 [18]. They support a standard error recovery with the 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 a lost internet connection, 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 no longer responding. If the disconnected controller returns after a while, 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 [20] 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 [6]. The system uses a list of all the peers that are available in the 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 [14]. 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.

Using a single source to generate code for clients and servers is also done in the Eliom [17] 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 sent to any other server or client and this decision can be made at run-time. 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.
6 CONCLUSION AND FUTURE WORK

In this paper we have presented a new distributed architecture for the evaluation of iTask applications and discussed its implementation. The new architecture is a generalization of the old iTask system that had only one central server. In the new architecture we can have multiple iTask servers (controllers) running distributed over a network of platforms, on servers as well as on clients. Tasks can be pushed (assigned to someone somewhere) or pulled (using a subscription mechanism) from one controller to another, regardless of the platform they are running on.

We solved the disadvantages of the original 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, thus 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 generate state-of-the-art native code. The implementation of the distributed platform 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 requires that the native code images (Intel or ARM) are available on all controlling devices beforehand, either as executable or as app. In the future we want to be able to dynamically extend a running controller with the code needed for the evaluation of a closure. This means that one has to be able to extend a running Clean application with new code while it is running. We already have the infrastructure for Windows, but need to port it for the other platforms. 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 development of the distributed, dynamic, version of the iTask system made us even more enthusiastic about the underlying concepts than we already were. The specification of iTask applications is not affected by the changed architecture and remains elegant and high level. The implementation of the distributed version turned out to be very elegant as well. The major technical hurdle to take was the technological ability to ship closures to distributed Clean applications running on different platforms. Once this was achieved, it was relatively easy to turn the existing iTask system into a distributed, dynamic, version. The new technology enables the system to push tasks to a remote controller, pull tasks that satisfy a certain criteria to a local controller, and accomplish remote access to task values and SDS’s. We plan to use the new distributed, dynamic, system for developing real world applications, to start with in the challenging domain of Command and Control.

ACKNOWLEDGMENTS

We thank Bas Lijnse, Tim Steenvoorden and the anonymous reviewers for their valuable comments that helped improve this paper.

REFERENCES


