Computing Science: Master's Thesis

Radboud University Nijmegen

Implementing Asynchronous I/O in iTasks

January 2021 - July 2021

Author:
ing. Gijs Alberts

Supervisor:
dr. Pieter Koopman

Second supervisor:
dr. Bas Lijnse

Second reader:
Mart Lubbers MSc

July 7, 2021

Abstract

iTasks is a general-purpose framework for developing web applications. It is implemented in the pure functional programming language Clean. iTasks allows to develop web applications in a task-oriented manner. The idea is that end-users that develop web applications using iTasks should only have to specify what tasks should be accomplished using the web application. The iTasks framework takes care of the technical realization of these tasks.

iTasks uses network I/O to communicate with other programs (e.g: browsers) over a network connection. IPC (inter-process communication) is used by iTasks to be able to communicate with programs that are executed on the same computer. This thesis focuses on the network I/O and IPC functionality that is provided by iTasks. This is a small part of the functionality that is provided by the iTasks framework as a whole.

More specifically, the network I/O functionality that is provided by iTasks concerns the iTasks HTTP server. The iTasks HTTP server is used to serve the web applications that are developed using iTasks. Furthermore, it involves the functionality that iTasks provides to perform network I/O as an end-user.

The IPC functionality of iTasks enables the end-user to execute an external process through an iTasks program. An external process can be seen as any executable computer program. iTasks provides functionality that allows communicating with the external process that was executed through IPC. Like the network I/O functionality, the iTasks IPC functionality involves performing I/O as well.

In the existing iTasks implementation, network I/O and IPC are implemented through two separate concepts. Network I/O is based on the concept of I/O multiplexing. IPC is based on a time-based polling concept. This architecture has evolved as iTasks was extended over time but is unnecessary. Having network I/O and IPC be implemented through the same concept makes the iTasks framework more consistent and easier to maintain.

In addition, the existing iTasks implementation makes use of blocking I/O operations. This is a limitation because having I/O operations block results in iTasks applications (temporarily) becoming unresponsive. Furthermore, this limits iTasks applications from being horizontally scaled. Horizontally scaling iTasks applications is a future goal of the iTasks project. This would allow iTasks to be used in large-scale projects that require serving a large number of users simultaneously.

As part of this thesis project, the iTasks network I/O and IPC functionality was re-implemented such that IPC and Network I/O functionality are both provided through the I/O multiplexing concept. This resulted in a replacement implementation. Providing IPC and network I/O through this concept increases the maintainability and consistency of the implementation as a whole. The replacement implementation is inspired by the libuv library. Libuv provides asynchronous I/O and is used by Node.js and various other projects.

In the replacement implementation, the blocking I/O operations are implemented in a non-blocking manner instead. As a consequence, iTasks applications do not block when performing I/O. This is beneficial because iTasks applications can no longer become (temporarily) unresponsive when performing I/O. Furthermore, this is a necessary step for being able to horizontally scale iTasks servers, which is a future aim of the iTasks project.

As a result of this thesis, the iTasks network I/O implementation was optimized with the goal of increasing the scalability of iTasks applications. Implementing these optimizations lead to benchmarking the existing iTasks HTTP server and the replacement HTTP server. The results of this benchmark show that the replacement HTTP server scales significantly better than existing one.

Acknowledgement

I would like to thank Pieter Koopman and Bas Lijnse for their supervision throughout this project. The weekly meetings and feedback were very valuable and helped me a great deal. I would also like to thank them again for supervising me during my research internship, which was a precursor to this thesis. I would like to extend my thanks to Mart Lubbers for agreeing to be the second reader of the thesis. Furthermore, I would like to thank the members of the iTasks/Clean community in general for allowing me to participate in discussions that were relevant to this thesis. This helped me understand how I could best make a positive impact. I would like to thank Job Cuppen for performing the benchmark measurements on macOS and providing feedback on the reproducibility chapter. Finally, I would like to express my gratitude to my parents for their support.

Contents

Chapter 1

Introduction

Almost every useful computer program makes use of Input/Output (I/O) to some degree. There are many different forms of I/O. For instance, think of humans interacting with computer programs through a CLI (command-line interface) or a GUI. In addition, there are forms of I/O that allow computer programs to interact with other computer programs through I/O. For example, the internet consists of web servers that provide websites to clients. Communication between clients visiting a website and the web server responding to their requests happens through network I/O. Visiting a website leads to the browser establishing a connection to the web server which serves the website. After establishing the connection, requests and responses may be exchanged. Often, the client will request a web page that is served by the web server. The web server processes this request and provides the web page as a response. The response is then interpreted by the browser of the client. This leads to the browser displaying a web page. The browser and web server are both computer programs that rely on network I/O to communicate. Programs running on a single computer may also communicate through inter-process communication (IPC). This form of communication is also based on I/O. In addition, programs may make use of file I/O to store and access data.

Operating systems provide abstractions which allow to perform these forms of I/O. An example of such an abstraction is a socket. Sockets are used to abstract from network I/O channels. Computer programs may use these abstractions and perform operations on them. For example, a computer program may write to a socket to send data over a network connection. Websites are primarily provided through a client-server model. When using this model, a socket is created to represent the server. For each client that connects to the server, a socket is created on the server-side. The server may then interact with the created socket to communicate with the connected client. Likewise, the client itself obtains a socket as a result of connecting to the server, this socket may be used to communicate with the server. As a result, a server may communicate with possibly many clients.

This thesis concerns itself with the way network I/O and IPC (inter-process communication) mechanisms are provided by iTasks. iTasks is a general-purpose framework for developing web applications. It is implemented in the functional programming language Clean. iTasks provides a means to write a program as a composition of tasks and then execute it as web application. Executing such an iTasks program leads to a web server being started. The web server serves the web application. As a result, clients may then interact with the web application. This interaction is provided through the client-server model. The iTasks framework is platform-independent. Concretely, this means that iTasks applications run on the Windows, macOS and Linux operating systems. iTasks provides a DSL (domain-specific language) that allows developers to write concise specifications of web applications.

For any iTasks program, iTasks:

- Sets up a web server which serves the web application that was specified to clients.
- Provides network I/O between clients and the web server to be able to respond to requests made by clients. For example, such a request may occur when a client requests a web page on the web server.
- Generates (responsive) forms that allow users to interact with the web application. The content of the forms that are generated are based on the types that were specified/inferred through the specification of the web application.
- Checks that the types of the user input provided by the clients are correct and properly notifies the user if the input is not of the correct type.

It follows that even for simple programs, iTasks takes care of several technical details for the developer. iTasks tries to remove the need for the developer to think about things that are not related to what the users should accomplish by using the web application [20].

To summarize, iTasks provides a DSL that allows to develop web applications through concise programs. In doing so, iTasks minimizes the need for the user to concern themselves with details that are not related to the task at hand.

From the foregoing it follows that there is an internal implementation that takes care of these details for the developer. A simplified overview of the internal implementation is included below.

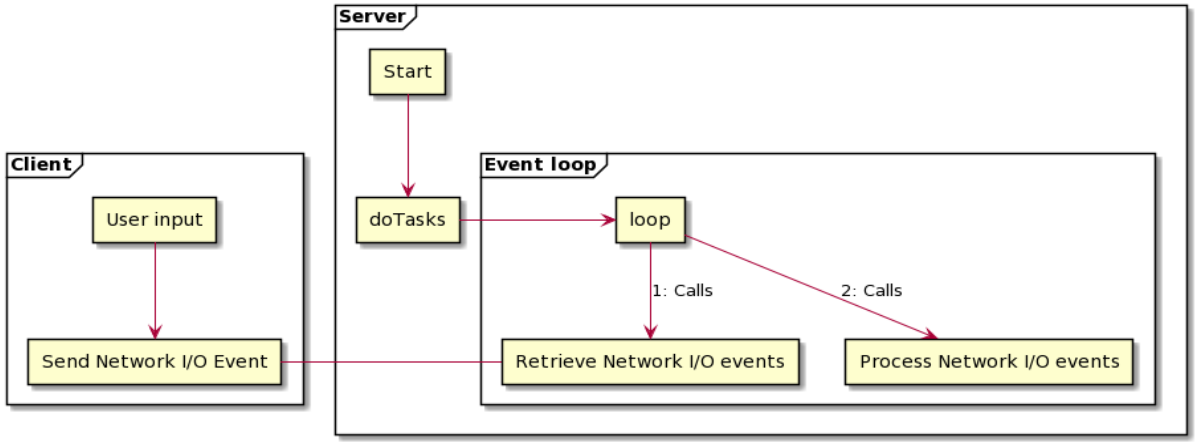


Figure 1.1: Simplified overview of iTasks internal implementation.

The internal implementation of iTasks revolves around an event loop. The event loop is implemented as a function which recursively calls itself. This function is called `loop`. As a result, the `loop` function is repeatedly evaluated while an iTasks program is being executed. The event loop is used to react to events. For instance, an event could be that the user clicked the submit button on a web page or entered a value in an input field. The browser of the user notifies the server of such events using network I/O. These network I/O events are then processed by the server through the event loop.

As a consequence of iTasks providing a web server on which the generated web pages are hosted, it internally relies on network I/O. As the above figure shows, processing and retrieving network I/O events is done within the event loop. Therefore, the event loop plays an important role in providing network I/O as well. Network I/O is one of the topics this thesis focuses on. Another topic this thesis focuses on is improving the way IPC (inter-process communication) is provided in iTasks. IPC is used to provide communication between processes (programs) running on a computer. There are some drawbacks to the existing IPC implementation of iTasks, which are described in the problem statement. One of these drawbacks is that IPC events are not being processed and retrieved within the event loop.

1.1 Problem description

It might not always be possible to perform certain I/O operations. For example, a program may not read data from a socket if there is no data available. Not knowing whether operations may be performed or not may result in problems. A web server is often connected to multiple clients. As a consequence, the server must read from multiple sockets to receive the requests of each connected client. Provided a single thread is used, it is only possible to receive data from a single socket at a time. At the time of writing, iTasks is single-threaded.

Consider the possibility of no data being available to read when the server attempts to read from a socket. A possibility to handle this situation is to block waiting for data to be received on the socket. In this case, the thread waits until data is received and only then returns from the function call [4]. The communication with other clients that are connected to the server would be halted while the thread is waiting on data to arrive. If a web server has to block waiting on operations that may not be performed this could lead to the web server not being available for requests any longer. It is also possible to receive data in a non-blocking manner. In this case, an error is returned if there is no data available to read [4]. This approach causes problems because it is difficult to decide how often the socket should be read. Reading too often results in many obsolete function calls. Reading infrequently increases the time it takes for the server to respond to requests.

Various solutions allow avoiding the drawbacks of the approaches mentioned above. During my research internship, I investigated which solution would be best suited for iTasks and developed a proof of concept that indicated that the solution could be used for iTasks. The solution is based on I/O multiplexing mechanisms. The approach taken was inspired by the approach of the libuv library. The libuv library is used to provide asynchronous I/O in Node.js [10]. The research goal of the thesis and preceding research internship is to investigate whether the approach that is taken by libuv could fit the existing iTasks architecture.

iTasks relies on network I/O and also provides IPC (inter-process communication) I/O operations to end-users. The existing iTasks implementation makes use of an I/O multiplexing mechanism to avoid the problem described above as well. However, this I/O multiplexing mechanism has some drawbacks. The concept of I/O multiplexing is introduced below. This is followed by an outline of the drawbacks of the I/O multiplexing mechanism that is used by the existing implementation.

Operating systems unify the abstractions for Network, IPC and File I/O through file descriptors (Linux, macOS) and file handles (Windows). This means that for Linux and macOS, sockets, pipes (IPC), pseudoterminals (IPC) and files can be represented through file descriptors. Similarly, for Windows, sockets, pipes and files can be converted to the `HANDLE` type. This is the type representing file handles within Windows. In this thesis, file handles are referred to as file descriptors for readability. In the context of this thesis, the difference between a file handle and a file descriptor is not important.

Operating systems provide mechanisms to perform I/O operations without having the program block in the case that the operation may not be performed. Some of these mechanisms can be categorized as I/O multiplexing mechanisms. I/O multiplexing mechanisms allow monitoring which I/O operations are possible on a set of file descriptors (e.g sockets, pipes). In the case of a web server, the sockets (file descriptors) of the connected clients could be monitored for readability. Using an operation, it is then possible to retrieve the subset of file descriptors that are readable out of a set of monitored file descriptors. What to monitor for (readability, writability, exceptions) can be configured for each individual file descriptor that is monitored.

When the program retrieves that a file descriptor is in a state that it is being monitored for, it may react by performing an operation that depends on the file descriptor being in this state. In most cases, the operation will then immediately succeed. This does not happen in all cases, however. For example, when reading it is not guaranteed that a read succeeds even though the I/O multiplexing mechanism indicated readability [2] (see "Bugs" section). Similarly, when writing data it is not guaranteed that all data may be transferred at once. These situations may lead the program to block if the operations are performed in a blocking manner. Having the program block when performing I/O operations should be avoided, as it will leave the application unable to respond to other events. Therefore, I/O operations

should be performed in a non-blocking manner. This means that if an operation may not immediately be performed, the operation will immediately return a specific error that can be handled.

iTasks currently makes use of the `select` I/O multiplexing mechanism to monitor sockets. The `select` I/O multiplexing mechanism is a portable mechanism. It is implemented on Windows, Linux and macOS. However, `select` does not scale well. `select` was introduced to (BSD) Unix in 1983 [19] and has a long history of use. Retrieving the subset of file descriptors of a set of file descriptors on which operations may be performed has an $\mathcal{O}(n)$ time complexity, with n being the number of file descriptors being monitored [1]. Furthermore, monitoring over 1023 file descriptors at once using `select` should not be attempted on Linux [2] (see "Bugs" section). The `select` I/O multiplexing mechanism is therefore limited in terms of scalability.

There are more recent I/O multiplexing mechanisms that can retrieve such a subset with a $\mathcal{O}(1)$ time complexity [1]. As a result, using such I/O multiplexing mechanisms can improve the performance of iTasks applications that perform I/O operations as the number of sockets being monitored increases. In addition, these I/O multiplexing mechanisms can easily monitor larger numbers of file descriptors and do not have a practical limit on the number of file descriptors that can be monitored. The best performing I/O multiplexing mechanisms are `epoll` (Linux), `kqueue` (macOS) and `IOCP` (Windows) [5]. However, the performance advantage of these mechanisms only becomes significant as thousands or more file descriptors are monitored [9]. The scalability advantage provided by the replacement I/O multiplexing mechanisms is worth mentioning but it is not the main goal of the thesis.

The Windows IPC implementation does not use an I/O multiplexing mechanism for IPC as `select` does not allow to monitor pipes on Windows [3]. As a result, IPC and network I/O are provided through conceptually different implementations. However, pipes may be monitored on Windows through the `IOCP` I/O multiplexing mechanism. Using `IOCP` allows to monitor for IPC through the event loop as well. This allows to provide IPC and network I/O through a conceptually equivalent model. This simplifies the iTasks implementation and makes it more consistent. In addition, the Windows IPC implementation is improved as the named pipes may be monitored through an I/O multiplexing mechanism. This allows to perform the operations using the indications that are provided by the I/O multiplexing mechanism. As a result, a goal of the thesis is to replace the `select` I/O multiplexing mechanism for the `IOCP` I/O multiplexing mechanism on Windows.

Currently, the internal implementation may retrieve and process I/O events in several locations within the internal implementation. In this thesis we aim to unify all retrieval and processing of events in a single location. This increases the maintainability of the implementation. To accomplish this, it must be possible to monitor IPC and network I/O using the same approach. This is made possible by using `IOCP` on Windows.

Another reason for retrieving and processing I/O events in multiple locations is that several operations within the existing implementation may block. Providing operations in a blocking manner results in a simpler implementation compared to providing them in a non-blocking manner. This is a result of blocking operations guaranteeing that the operation succeeded before the program continues execution. In the case of using completely non-blocking operations, operations may have to be delayed to a later point in time, which complicates matters. However, performing operations in a blocking manner has the drawback of making it possible for the program to block. This leaves the iTasks program (temporarily) unresponsive to other events.

To give an example, sending and receiving network I/O data may block in the existing implementation. When sending or receiving data, initially a non-blocking operation is used. However, the non-blocking operation may fail. In this case, the existing implementation uses the `select` I/O multiplexing mechanism to wait for the file descriptor to be in the required state. Instead of providing a set of file descriptors, a single file descriptor is provided. `select` is used as a blocking call. As a result, it only returns once the socket being monitored is in the required state. Therefore, sending and receiving data in the existing implementation may block. Certain other operations like `connect` (network I/O) are performed in a blocking manner as well. A goal of the thesis is to provide all operations in a non-blocking manner. This paves the way to have a non-blocking event loop. This is useful because iTasks depends on having the event loop being regularly evaluated. If the event loop blocks, iTasks is unable to respond to events. Using non-blocking operations ensures the iTasks event loop does not block when performing

network I/O/IPC operations. In addition, implementing I/O operations in a non-blocking manner makes it possible to retrieve I/O events within a single location instead of several locations.

Furthermore, providing I/O operations in a non-blocking manner is a necessary step for horizontally scaling iTasks applications. A future aim of the iTasks project is to make it possible to serve iTasks applications in a distributed manner. This would enable iTasks applications to use all the threads of the CPU of the computer which provides the iTasks application. In addition, this would allow traffic to iTasks applications to be distributed over multiple servers. Horizontal scaling is beneficial as it allows iTasks to be used for developing web applications that require serving large numbers of simultaneous users.

As stated, a goal of the thesis is to replace the `select` I/O multiplexing mechanism for the `IOCP` I/O multiplexing mechanism. There are significant differences between the `IOCP` and `select` I/O multiplexing mechanisms and how they are used. As a consequence, `select` is replaced by `kqueue` on macOS and `epoll` on Linux. The reasoning for this replacement is that `epoll` and `kqueue` are very similar I/O multiplexing mechanisms and they differ less from `IOCP` than `select` does. Furthermore, these mechanisms are not limited in terms of scalability. As stated, `select` has the limitation of not being able to monitor over 1023 file descriptors.

The drawback of using the `kqueue`, `epoll` and `IOCP` I/O multiplexing mechanisms is that the mechanisms are operating system specific. `kqueue` and `epoll` are more similar to `IOCP` than `select`. Nonetheless, significant differences remain between the `IOCP` and `epoll` I/O multiplexers. Despite these differences, the network I/O/IPC abstractions that are provided to the end-users should be platform-independent. This is required because iTasks is a platform-independent framework. Accomplishing this requires providing abstractions to the end-users that abstract away from the differences within the internal implementation.

To summarize, this thesis focuses on implementing network I/O/IPC through `epoll` (Linux), `kqueue` (macOS) and `IOCP` (Windows) in iTasks. The `select` I/O multiplexing mechanism that is currently being used is replaced. Replacing `select` for `epoll`, `kqueue` and `IOCP` is beneficial because it allows to monitor IPC file descriptors on Windows. This enables IPC and network I/O to be provided through a conceptually equivalent approach. Furthermore, the thesis focuses on unifying the retrieval of I/O events in a single location instead of several locations throughout the code. This increases the maintainability of the implementation. In addition, a goal of the thesis is to provide network I/O and IPC solely through non-blocking operations. This is useful because this makes sure iTasks remains able to respond to events. In addition, implementing I/O operations in a non-blocking manner is required for horizontally scaling iTasks applications. This is a future goal of the iTasks project.

1.2 Document structure

- Chapter 1 contains the introduction, problem statement and document structure.
- The existing implementation is introduced in chapter 2.
- Chapter 3 contains an introduction of the replacement implementation. The replacement implementation replaces the existing implementation.
- The implementation of the existing HTTP server is discussed in chapter 4.
- The implementation of the HTTP server in the replacement implementation is described in chapter 5.
- Chapter 6 provides an overview of how network I/O functionality aimed at end-users is provided in the existing implementation.
- Chapter 7 examines the internal implementation of network I/O for the replacement implementation.
- The implementation of IPC (inter-process communication) functionality within the existing implementation is described in chapter 8.
- The internal implementation of IPC within the replacement implementation is discussed in chapter 9.
- Chapter 10 describes the process of benchmarking the iTasks HTTP server and the obtained results. For the benchmark, the existing HTTP server implementation was compared to the replacement implementation in terms of performance and scalability.
- Chapter 11 contains the conclusion of the thesis.
- Chapter 12 provides suggestions for future work that is related to this thesis.

Chapter 2

Existing Implementation - Introduction

iTasks is a general-purpose framework for developing web applications. iTasks is implemented in the pure functional programming language Clean. A part of the implementation of iTasks provides network I/O and IPC functionality. This part of the iTasks implementation is used by iTasks to communicate with external programs, possibly over a network connection. The focus of this thesis is on this part of the iTasks implementation. Hereafter, this part of the iTasks implementation is referred to as the existing implementation.

This chapter introduces the network I/O and IPC functionality that is provided by iTasks. Furthermore, this chapter introduces the internals of the existing implementation. This is done with the goal of providing a general idea of how the existing implementation provides Network I/O and IPC functionality. Furthermore, the main drawbacks of the existing implementation are identified. To provide solutions for these drawbacks, the network I/O and IPC implementation was revisited. This resulted in a replacement implementation. The replacement implementation is introduced in the subsequent chapter.

2.1 Network I/O

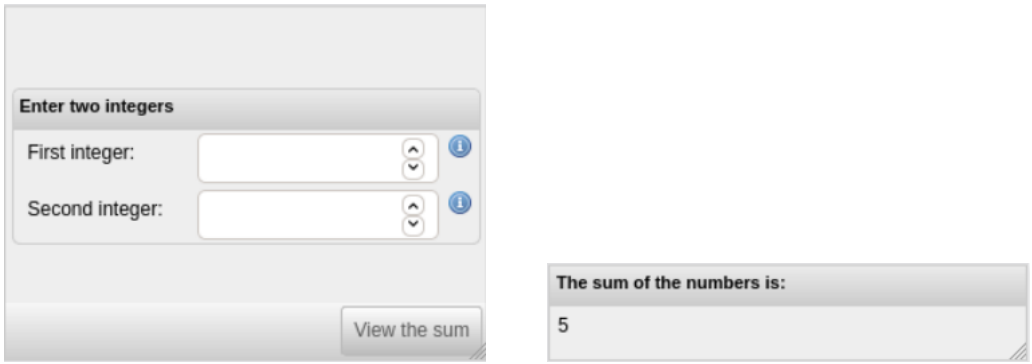
iTasks provides Network I/O functionality, which naturally is concerned with providing I/O communication over a network. iTasks provides a built-in HTTP server that is used to serve the web applications that are developed using iTasks. The HTTP server relies on network I/O to communicate with the clients that are connected to the HTTP server. Usually, the clients are browsers that are used by people that interact with the iTasks web application.

As an example of how network I/O may be used by iTasks, consider the following simple iTasks program:

```
module example

import iTasks

Start w = doTasks addTwoNumbers w

:: IntPair = {firstInteger :: Int, secondInteger :: Int}
derive class iTask IntPair

addTwoNumbers :: Task Int
addTwoNumbers =
    Title "Enter two integers" @>>
    enterInformation [] >>*
        [   OnAction
            (Action "View the sum")
            (
                hasValue \{IntPair|firstInteger, secondInteger} ->
                viewInformation [] (firstInteger + secondInteger)
```

```
            <<@ Title "The sum of the numbers is:"
        )
    ]
```

The program included above allows the user to enter two numbers, confirm their choice and then displays the sum of the numbers on the screen. Even though the program is rather concise, iTasks generates a complete web application that is used to serve the program. When the program is executed, iTasks starts a web server that allows clients to perform the specified workflow. This program provides a (responsive) form to the end-user. In this form, the end user may input two integers and press a button to submit the input. This is followed by the sum of the numbers being displayed to the end-user.



(a) Entering two integers

(b) Viewing the sum

Figure 2.1: The generated user workflow.

Network I/O is used to notify the server of the numbers that were entered by the user and the fact that the user submitted the form. Furthermore, network I/O is used to retrieve the web page that is shown to the user. The server makes use of sockets as a communication channel for interacting with the connected clients. The existing HTTP server relies on the `select` I/O multiplexing mechanism. The HTTP server makes use of `select` to detect connection requests made by clients. In addition, `select` is used to react to user input, such as submitting the form in the above example. The role of the `select` I/O multiplexing mechanism is described in more detail in section **2.3**

2.2 IPC

The IPC functionality provided by iTasks involves being able to execute an external process as an end-user from an iTasks program. An external process can be seen as any executable computer program. iTasks provides a means to communicate with the external process that was executed. This makes it possible to provide input to the external process and read the output of the external process.

As an example, consider the following program, which will be executed as an external process.

```
#include <stdio.h>

// addtwonumbers.c: Reads two numbers as input and prints the sum as output
    .
int main() {
    int a, b;
    scanf("%d", &a);
    scanf("%d", &b);
    printf("%d\n", a + b);
    return 0;
}
```

The program included below executes the C program specified above and provides two integers (2 and 3) as input to the external process. It reads the sum of the integers that is written as output by the external process once the external process terminates. The sum of the numbers provided as input is then displayed to the user. This happens because the external process writes the sum as output.

```
module process

import iTasks
import System.Time

Start w = doTasks process w

process :: Task [String]
process = withShared ["2\n3\n"] \inputStorage ->
          withShared ([], []) \outputStorage ->
          externalProcess {Timespec|tv_sec=1, tv_nsec=0}
                          "/path/to/addtwonumbers.exe" [] ?None 0
                          ?None inputStorage outputStorage
          >-| get outputStorage >>~ \(output,errorOutput) ->
          viewInformation [] output
```

The communication between the iTasks program and the external process is made possible by using pipes or a pseudoterminal. Pipes and ptys (pseudoterminals) are abstractions which are used to represent an inter-process I/O channel. The iTasks program reads from a pipe/pty to receive the output of the external process. Similarly, the iTasks program writes to a pipe/pty to send input to the external process. The developer can store input that should be written to the external process in a SDS (Shared Data Source). Likewise, the output of the external process is stored in a SDS as well. A SDS is used to read and write shared data.

The existing iTasks IPC functionality does not rely on the `select` I/O multiplexing mechanism. The reason for this is that the Windows implementation of `select` does not allow to monitor pipes (IPC) [3]. Instead iTasks uses a time-based non-blocking approach to monitoring the pipes that are involved in communication. This means that iTasks will attempt to read from the pipe/pty periodically to read the output of the external process. This is done regardless of whether there is data available to read or not.

2.3 Internal implementation

The existing network I/O implementation relies on the `select` I/O multiplexing mechanism. I/O multiplexing mechanisms such as `select` provide a means to monitor a set of file descriptors.

Operating systems provide abstractions which represent network I/O and IPC communication channels. In the case of network I/O, such an abstraction is a socket. For example, a program may write data to a socket to send data over a network connection. In the case of IPC, examples of such abstractions are pipes and pseudoterminals (ptys). A side note is that using a pseudoterminal for IPC is not supported on Windows.

Pipes, sockets and pseudoterminals are all file descriptors. Therefore; pipes, sockets and pseudoterminals may be monitored by the `select` I/O multiplexing mechanism. There is an exception, the Windows `select` implementation does not allow to monitor pipes. This drawback is described in more detail in section **2.3.3**.

I/O multiplexing mechanisms allow to retrieve which file descriptors out of a monitored set of file descriptors are in a specific state (e.g: readable, writable). A file descriptor being in a state may indicate that a certain I/O operation may be performed. For example, a socket being readable implies that a connection request may be accepted when monitoring a socket that is configured to listen for connections. The existing HTTP server relies on the `select` I/O multiplexing mechanism to be able to react to the activity of clients that interact with the HTTP server.

There is more to the existing implementation than just the `select` I/O multiplexing mechanism itself. The I/O multiplexing mechanism will not start monitoring the file descriptors involved in I/O by itself, for instance. The I/O multiplexing mechanism plays a small yet vital role within the implementation as a whole. This is shown by the overview that is included below.

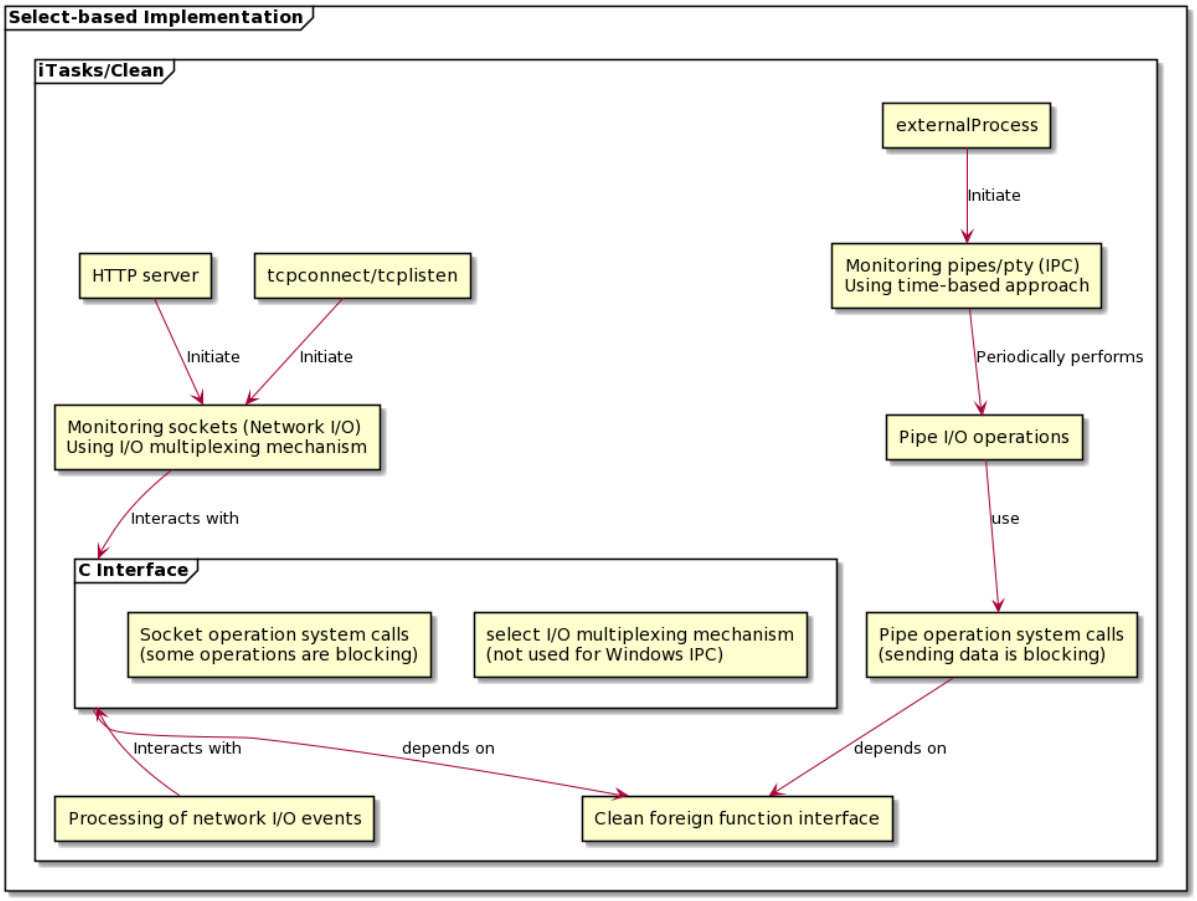


Figure 2.2: Abstract overview of the existing implementation.

The above overview illustrates the general idea of how network I/O and IPC is provided by iTasks in the existing implementation. In addition, the overview accentuates the main drawbacks of the existing implementation.

2.3.1 iTasks/Clean

As the above overview shows, a part of the implementation is implemented in Clean/iTasks. The iTasks HTTP server relies on network I/O. `tcplisten`, `tcpconnect` are the tasks which are used to perform network I/O as an end-user. Likewise, `externalProcess` is the task that is used to execute an external process as an end-user. The `externalProcess` task is then able to communicate with this external process using IPC (inter-process communication). The internal implementation of these tasks and the HTTP server ensure that the file descriptors (sockets, pipes, ptys) that are involved in communication are monitored.

As stated, pipes and pseudoterminals (ptys) are abstractions that are used to represent an inter-process communication channel. Pipes and pseudoterminals allow to send data to the stdin of an external process and receive the output that was written to the stdout or stderr of an external process. An external process can be seen as any kind of executable computer program. stdin is the standard input channel of programs. For example, when a user executes a program in a terminal and then provides user

input to the terminal, this input is written to stdin by default. Similarly, stdout is the standard output channel of a program. For example, a program prints data, this data is written to stdout by default. stderr follows the same approach as stdout, however it is used for error data instead. It is possible to communicate with an external process by interacting with the stdin and stdout/stderr of the external process. This is accomplished by redirecting the stdin/stdout/stderr of the process to pipes/a pty. The pipes/pty are then interacted with.

A socket is an abstraction that is used to represent a network I/O communication channel. Sockets allow to communicate with a peer over a network connection. This involves establishing a connection to the peer, which is done making use of sockets as well. In the case of the iTasks HTTP server, sockets are used for communication between clients and the HTTP server. For example, when a browser is used to access a web page served by an iTasks HTTP server, the browser establishes a socket connection to the HTTP server. The browser then uses the socket connection to communicate with the HTTP server.

In the existing implementation, a different approach is used to monitor the file descriptors involved in I/O depending on whether IPC or network I/O functionality is used. This is illustrated by the overview.

In the case of using network I/O, the sockets that are involved in I/O communication are monitored using the `select` I/O multiplexing mechanism. This allows to monitor the sockets for being in a certain state (e.g readable, writable). When a socket is in the state that it is being monitored for, an event is returned. The internal implementation of network I/O processes the events that the `select` I/O multiplexing mechanism returns. For example, if a socket is monitored for readability and the socket is readable once events are retrieved, `select` returns an event for this socket. This event is processed using Clean/iTasks. In the case of the socket being a listening socket, a readability event indicates that a connection request is available. This event is processed by performing an `accept` operation, which is used to accept the connection request. In the case of the socket being a client socket, a readability event indicates that data is available to be read. In this case, the `read` operation used to read the data that is available to be read and process this data. To perform network I/O operations, monitor sockets and interact with the `select` I/O multiplexing mechanism, the iTasks/Clean implementation depends on a C interface. The C interface is described in section **2.3.2**.

In the case of using `externalProcess` (IPC), a time-based approach is used for monitoring the file descriptors due to limitations of the `select` I/O multiplexing mechanism on Windows. This is a drawback of the existing implementation, which is described in more detail in section **2.3.3**. The end-user can provide a time interval to the `externalProcess` task and as a result, the pipes/pty involved in communicating with the external process are read periodically. That means that every given time period, the implementation will attempt to read from the pipes/pty in a non-blocking manner. On Windows this is done regardless of whether there is data available or not. By reading the pipes/pty, the output of the external process may be processed. The IPC implementation makes use of system calls to be able to perform I/O operations on pipes and pseudoterminals. To be able to perform these system calls, the IPC implementation relies on the Clean foreign function interface. The Clean foreign function interface is described in section **2.3.4**.

2.3.2 C interface

As the overview that is included above illustrates, the iTasks/Clean implementation depends on a C interface. The C interface – that can be seen as a set of C functions that are called by Clean – provides interaction with the `select` I/O multiplexing mechanism. As stated, the `select` I/O multiplexing mechanism is used to retrieve which file descriptors out of the monitored set of file descriptors are in a specific state (e.g: readable, writable). In addition, the C interface provides functions which perform I/O operations (e.g `send`, `connect` and `accept`). Clean is able to call the C functions in the C interface through the Clean foreign function interface, which is described in section **2.3.4**.

2.3.3 General improvements

The Windows implementation of the `select` I/O multiplexing mechanism has the limitation of not being able to monitor pipes [3]. A pipe is an abstraction that is used to represent an inter-process communication channel. The processing of network I/O events relies on the `select` I/O multiplexing

mechanism. However, pipes may not be monitored using the `select` I/O multiplexing mechanism on the Windows platform. As a result, processing I/O events for IPC and network I/O could not be done using the same approach when using `select`.

This means that the internal implementation of IPC takes a different approach to performing I/O operations. As stated, IPC I/O is performed periodically based on a time interval that is specified by the end-user. This means that the pipes involved in IPC are periodically read in a non-blocking manner in an attempt to receive data. This is done regardless of whether data is actually available or not. Similarly, the implementation periodically checks whether there is data to be sent to the external process. If there is data available to be sent, the data is sent to the external process.

This time-based approach is less optimal than performing I/O operations using a indication the the operation may actually be performed. Such indications are provided by I/O multiplexing mechanisms. Moreover, this approach leads to performing I/O operations according to two different concepts. However, both pipes and sockets may be monitored on Windows using the `IOCP` I/O multiplexing mechanism. If the `IOCP` multiplexing mechanism were to be used instead, I/O events for both IPC and network I/O could be processed using the same approach. Furthermore, monitoring pipes could be done using an I/O multiplexing mechanism instead of the current time-based approach. As a result, a goal of the thesis is to unify the approaches for providing IPC and network I/O through a single concept instead. This requires replacing the `select` I/O multiplexing mechanism for the `IOCP` multiplexing mechanism on Windows.

In addition, I/O operations may be performed in a blocking manner in the existing implementation. This is illustrated by the overview included above (figure **2.2**). For example, data is sent in a blocking manner for both the IPC and network I/O implementations. The problem with performing operations in a blocking manner is that the iTasks program may block when performing I/O. When the iTasks server blocks, it may not perform other operations or respond to events. Therefore, performing I/O in a blocking manner should be avoided. As a consequence, performing all I/O in a non-blocking manner is a general goal when replacing the existing implementation. This is also required for serving iTasks applications in a distributed manner. Doing so is a future aim of the iTasks project as this is required for using iTasks for large-scale applications that require serving a large number of users simultaneously.

2.3.4 Clean foreign function interface

Clean provides a foreign function interface (FFI) which may be used to perform system calls and evaluate C functions in Clean [6]. The FFI allows to pass Clean values to C functions or system calls and to pass return values back to Clean. The Clean foreign function interface is used to:

- Perform pipe I/O operations.
- Perform network I/O operations.
- Interact with the I/O multiplexing mechanism.

The types of the values that may be used are restricted. The following table gives an overview of:

- The types that may be used.
- How the Clean and C types that may be used correspond.
- How the types are represented in the type string that must be provided when calling the C function through Clean.

The type string that has to be provided is described in more detail further on in this section.

Clean type	C type	Typestring character
Int	int	I
Char	char	I
Bool	int	I
Real	double	R
String	CleanString	S
{#Char}	CleanCharArray	A
{#Int}	CleanIntArray	A
{#Real}	CleanRealArray	A
Int	anytype*	P

The `CleanString`, `CleanCharArray`, `CleanIntArray` and `CleanRealArray` C types are provided through the `Clean.h` header file. The operations that may be performed on values of these types are described in more detail in [6]. Any pointer within C may be represented within Clean as an Int. A simple example of the use of the Clean FFI is included below:

```
// C function accessing/updating global variable.
int global = 3;

int divGlobal(int n, int* err) {
    global--;

    // Division by 0.
    if (global == 0) {
        *err = -1;
        return 0;
    }
    *err = 0;
    return n / global;
}

// Corresponding Clean function.
:: Err :== Int
divGlobal :: !Int !*World -> (!(!Int, !Err), !*World)
divGlobal x state = code {
    ccall addGlobal "I:II:A"
}
```

A strict unique state (`!*World`) is required to be passed to the Clean functions that use the foreign function interface. This is required since the order of evaluation of C functions should be specified [7] [6]. This is necessary because evaluating C functions may lead to side effects occurring [6].

Not passing on the strict unique `World` state results in problems determining the order of evaluation because Clean has a lazy evaluation strategy by default. This means that, by default, the Clean compiler delays evaluating expressions until the result of the expression is needed. The order of evaluation is therefore not usually defined. If the functions that are used are pure and thus do not produce any side effects, this does not result in problems. However, when the functions used are impure, like the `divGlobal` function, problems arise. It should **not** be possible to be able to specify a program such as:

```
incorrect :: [Int]
incorrect = [x, y]
    where
    (x, err) = divGlobal 2
    (y, err) = divGlobal 3

// No unique World state passed through divGlobal.
:: Err :== Int
divGlobal :: !Int -> (!Int, !Err)
```

Since the compiler has a lazy evaluation strategy by default, it would not be defined whether `divGlobal 2` or `divGlobal 3` should be evaluated first. Since the `divGlobal` function results in side effects (e.g: it updates a global variable), this could have very undesirable consequences. Luckily, the compiler is able to determine the order of evaluation through the unique state [7]. Requiring this state to be passed on enforces the order of evaluation to be specified thanks to the uniqueness type system. If a value of a certain type is unique, there can be at most one reference to the value while it is being inspected by a function [8]. Hence specifying a program like

```
:: Err :== Int
incorrect :: *World -> [((Int,Err), *World)]
incorrect world = [divGlobal 2 world, divGlobal 3 world]

divGlobal :: !Int !*World -> (!(!Int, !Err), !*World)
```

leads to an uniqueness type error as the world has two references to it while it is being inspected by the `divGlobal` function. This uniqueness error can be prevented by specifying an order of evaluation like so:

```
:: Err :== Int
correct :: *World -> ([(Int, Err)], *World)
correct world0
    # (r0, world1) = divGlobal 2 world0
    # (r1, world2) = divGlobal 3 world1
    = ([r0,r1], world2)

divGlobal :: !Int !*World -> (!(!Int, !Err), !*World)
```

Because `divGlobal 3 world1` needs `world1` to be evaluated, the compiler will evaluate `divGlobal 2 world0` first. There is at most one reference to the world at all times so there are no uniqueness type errors in this case.

It is not necessary to give the world variables a different name (world0, world1, ...) but this was done to make the state updates more explicit. A program that is equivalent up to renaming of variables is shown below:

```
:: Err :== Int
correct :: *World -> ([(Int, Err)], *World)
correct world
    # (r0, world) = divGlobal 2 world
    # (r1, world) = divGlobal 3 world
    = ([r0,r1], world)

divGlobal :: !Int !*World -> (!(!Int, !Err), !*World)
```

It should be noted that the iTasks developer only has access to a single unique `World` state. As a consequence, it is not possible to avoid specifying the order of evaluation by using two or more different unique `World` states.

In addition, all of the types that are contained within the function type of the Clean function which performs the `ccall` have a strictness annotation (`!`). This makes sure the values that are provided are strictly evaluated. As a result, the values are reduced before they are passed to the C function, which is required [6]. Similarly, return values from C functions have a strictness annotation as well.

C functions and system calls are evaluated in Clean using the `ccall` instruction. `ccall` takes a system call or C function as an argument alongside a typestring.

ccall divGlobal `"I:II:A"` evaluates the divGlobal C function. The first characters in the provided type string are the types of the arguments provided to the C functions from Clean. In this case it is a single integer argument, hence the type string starts with `I:` (see the type conversion overview included above). The state is not provided to the C function itself. Therefore, it is not part of the first characters in the typestring.

The characters after the first colon separator indicate the types of the return values of the C function. In this case, the C function itself returns an `int` and the `int* err` is used to return an extra `int` to Clean. Therefore, this part of the typestring contains `II`. The characters after the second colon separator of the typestring are used to return values which are not returned or used by the C function that is called. Usually this part of the typestring indicates the type of the unique state as it is not used by the C function. `World` is an Array type so this part of the typestring contains `A`. The types of the return values are put in left-to-right order. The type for the return value of the function itself is always the first character of this part of the typestring. Consider the following example:

The type string "I:IIIS:A" indicates that the C function could have the following form:

```
int f (int cleanValue, int* extraRet0, int* extraRet1, CleanString*
    extraRet2)
{
    // Assign to return values.
    *extraRet0 = 2;
    ...
    return 0;
}
```

A corresponding Clean function could be:

```
:: CleanValue  :== Int
:: ReturnValue :== Int
:: ExtraRet0 :== Int
:: ExtraRet1 :== Int
:: ExtraRet2 :== String
f :: !CleanValue !*World ->
     (!ReturnValue, !ExtraRet0, !ExtraRet1, !ExtraRet2, !*World)
f = code {
    ccall f "I:IIIS:A"
}
```

Something to be aware of is that using the FFI may be unsafe. For example, the Clean type system may be broken through the FFI. The C function above could be made to return a `double` instead of an `int`. If the typestring added to the `ccall` and the return type of the Clean were not adjusted, Clean would still be expecting an `int` and this would not lead to problems at compile-time. This could lead to problems at run-time, however. Moreover, functions that use the FFI may be impure, as is the case for the example included above. A requirement for a function being pure is that the function, when provided the same arguments, always produces the same output. The output of the C function above depends on a global variable which is being updated as a side effect of calling the function. As a result, the example function above does not always produce the same output when provided the same input argument. The function is thus impure.

2.4 Summary

iTasks provides network I/O and IPC functionality to end-users and relies on an HTTP server for serving web applications that are developed using iTasks. This thesis focuses on revisiting the internal implementations of network I/O and IPC functionality with the intent of improving on the existing implementation. In the existing implementation, the end-user makes use of the `tcpconnect` and `tcplisten` tasks to perform network I/O. The HTTP server relies on network I/O as well. The `externalProcess` task is used to execute an external process and allows to communicate with this external process through IPC (inter-process communication).

Evaluating these tasks or making use of the HTTP server results in file descriptors (e.g: sockets, pipes) being monitored for being able to perform I/O operations. This monitoring may lead to performing I/O operations on the file descriptor, such as `accepting` a connection. The existing implementation depends on the Clean foreign function interface, the foreign function interface is used to call C functions and

system calls which are used to perform I/O operations and interact with the `select` I/O multiplexing mechanism.

File descriptors involved in IPC are monitored using a different approach than file descriptors involved in network I/O. The thesis aims at replacing the `select` I/O multiplexing mechanism for the `IOCP` multiplexing mechanism on Windows. This makes it possible to monitor IPC and network I/O using the same approach, which simplifies the implementation. In addition, this allows to unify the retrieval and processing of I/O events in a single location instead of several locations throughout the code. Furthermore, the existing implementation performs several I/O operations in a blocking manner. A goal for the replacement implementation is to implement all I/O operations in a non-blocking manner. As a result, the iTasks server will never block when performing I/O operations. This is beneficial because this ensures the iTasks implementation is always able to respond to events in a timely manner. In addition, performing I/O operations in a non-blocking manner is required for horizontally scaling iTasks applications. This is a future goal of the iTasks project. Being able to serve an iTasks application using multiple servers would make it possible to use iTasks for large-scale applications that require serving large numbers of users simultaneously.

Chapter 3

Replacement Implementation - Introduction

This thesis focuses on revisiting the existing iTasks network I/O and IPC implementation. The existing network I/O and IPC implementation have been introduced in the preceding chapter. The existing implementation depends on the `select` I/O multiplexing mechanism. Revisiting the existing implementation lead to a replacement implementation, which is introduced in this chapter. In the replacement implementation, `select` is being replaced for the `IOCP` (Windows), `kqueue` (macOS) and `epoll` (Linux) I/O multiplexing mechanisms. This is done to be able to provide IPC and network I/O through a single approach instead of two separate approaches. Providing network I/O and IPC through a single approach simplifies the implementation and makes it more consistent. On Linux and macOS, `kqueue` and `epoll` are used because these mechanisms are more similar to `IOCP` than `select`. Furthermore, like `IOCP`, `kqueue` and `epoll` are not limited in terms of scalability. The `select` I/O multiplexing that is used by the existing implementation may monitor at most 1023 file descriptors at once on the Linux platform [2] (see "Bugs" section).

This chapter introduces how network I/O and IPC is provided in the replacement implementation. The replacement implementation provides solutions to the drawbacks of the existing implementation. The drawbacks of the existing implementation are described in section **2.3.3**. `IOCP` takes a significantly different approach to I/O multiplexing compared to the other I/O multiplexing mechanisms. In this chapter, the approach of `IOCP` is introduced and compared to the approach of `kqueue`, `epoll` and `select`. As the replacement implementation provides operations in a non-blocking manner, data is sent in a non-blocking manner as well. The existing implementation sends data in a blocking manner. This chapter describes the process of sending data in a non-blocking manner as it applies to both network I/O and IPC.

The overview included below provides an abstract overview of how the replacement implementation provides Network I/O and IPC. It is similar to the overview of the existing implementation (figure **2.2**) and illustrates the main changes to the implementation.

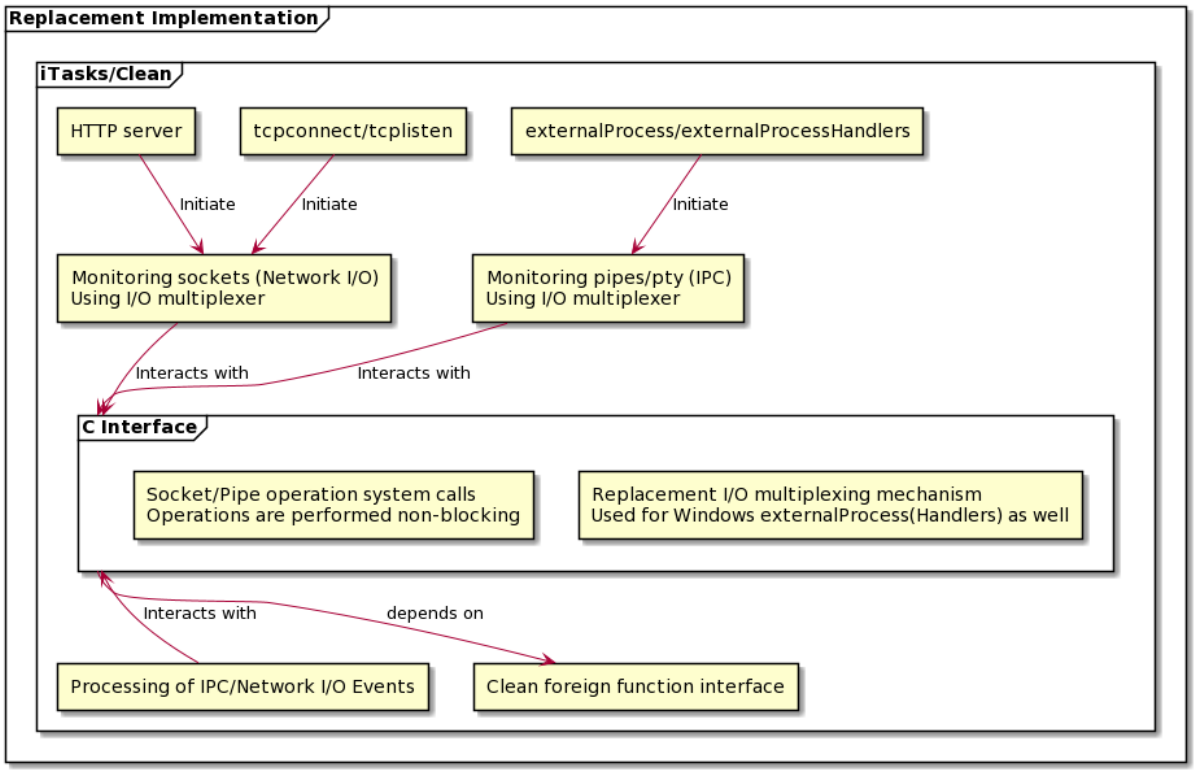


Figure 3.1: Abstract overview of the replacement implementation.

The replacement implementation makes use of the same approach when monitoring network I/O and IPC unlike the existing implementation. To achieve this, `select` was replaced for the `IOCP` multiplexing mechanism on Windows. The replacement implementation internally makes use of the `IOCP`/`kqueue`/`epoll` I/O multiplexing mechanisms. Which mechanism is used depends on the target platform of the iTasks application. `IOCP` is used on Windows, `kqueue` is used on macOS and `epoll` is used on Linux.

Like the existing implementation, the replacement implementation consists of a Clean/iTasks implementation and a C interface. The iTasks/Clean implementation depends on the C interface for interacting with the I/O multiplexing mechanism and performing I/O operations. In the replacement implementation, all I/O operations are provided in a non-blocking manner, unlike the existing implementation. The events returned by the I/O multiplexer are processed through Clean/iTasks. Processing IPC and network I/O events is done using the same approach.

The `tcplisten` and `tcpconnect` tasks are used to provide network I/O to end-users. The type of `tcplisten` and `tcpconnect` remained the same as the type that is defined for the existing implementation. The replacement implementation makes it possible to execute an external process and communicate with the external process. This is done using the `externalProcess` and `externalProcessHandlers` tasks. The `externalProcessHandlers` task is new and is described in chapter **9**. The `externalProcess` task has the same type as the `externalProcess` task featured in the existing implementation. The network I/O/IPC tasks and the HTTP server make use of the replacement I/O multiplexing mechanisms and perform operations in a non-blocking manner. Since the `externalProcess`, `tcplisten` and `tcpconnect` tasks are still of the same type, old programs using these tasks do not have to be altered.

3.1 C interface

Like the existing implementation, the replacement implementation depends on a C interface. The C interface can be seen as a set of C functions that may be evaluated through Clean. In this case, the functions within the C interface provide interaction with the I/O multiplexing mechanism. In addition,

functions within the C interface are used to perform operations on I/O abstractions (e.g sockets, pipes). The I/O multiplexing mechanism and operations that are used differ depending on the target platform. Hence, the functions within the C interface may have different implementations as well, depending on the target platform. It is possible to evaluate functions within the C interface using the Clean Foreign Function Interface (FFI), which is described in section **2.3.4**. The functions in the C interface have the same types on all platforms. The replacement I/O multiplexing mechanisms are similar enough to have a single iTasks/Clean implementation for all supported platforms.

3.2 Proactive and reactive I/O multiplexing mechanisms

An important goal of the thesis is to provide network I/O and IPC through a single approach instead of two separate approaches like the existing implementation. To realise this goal, the `select` I/O multiplexing mechanism was replaced for the `IOCP` (Windows), `kqueue` (macOS) and `epoll` (Linux) I/O multiplexing mechanisms. These mechanisms all have a distinct API. Consequently, the implementation of network I/O and IPC has a distinct internal implementation for each operating system. Nonetheless, the functionality that the implementation provides is exposed to the end-user through platform-independent abstractions. For example, the `tcplisten` task will behave the same regardless of which operating system is used.

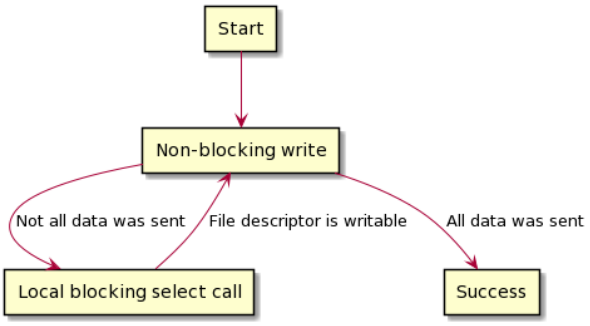
Even though the `select`, `kqueue` and `epoll` mechanisms make use of a different API they use the same concept for handling I/O events. However, `IOCP` (Windows) takes a different approach.

The difference between the two approaches can **generally** be summarized by stating that for `IOCP`, an event being returned means that something has happened. For `kqueue` and `epoll`, an event being returned means that something may happen at the moment in time that the event was returned. In the context of a connection request, that means that returning a connection event for `IOCP` means that the connection request has been accepted. For `epoll`, `kqueue` and `select` it provides an indication (not a guarantee) that a connection request may successfully be accepted. `IOCP` takes a proactive approach and operations are generally performed in an asynchronous manner. `epoll`, `select` and `kqueue` take a reactive approach and generally use synchronous operations. Operations that are performed on Windows (IOCP) may also end up completing synchronously. Furthermore the `connect` operation, which is used by the reactive I/O multiplexing mechanisms may be performed in an asynchronous manner. Unifying both the proactive and reactive approaches through a single platform-independent abstraction resulted in challenges. These challenges were related to abstracting away from the differences between the proactive and reactive approach within the internal implementation. The differences between the proactive and reactive approach is the root cause of most of the platform-specific differences within the internal implementation. Several of the I/O events returned to Clean have to be processed differently depending on whether `IOCP` is used or not, for instance. Different processing is necessary because the events have a different meaning depending on the platform that is used.

3.3 Sending data

A goal in regard to the replacement implementation is to provide I/O operations in a non-blocking manner. As a result, the replacement implementation takes a different approach to sending data. First, this section describes how data is sent within the existing implementation. This is followed by a description of the approach of the replacement implementation. The approach to sending data in the replacement implementation has been inspired by the libuv library. Libuv is a library which is used to provide asynchronous I/O operations in Node.js [10].

The existing implementation uses a local select call which blocks to guarantee that the data is sent or an error is returned before the program continues. The following figure illustrates how data is sent in the existing implementation:



Note that the use of local means that a single file descriptor is being monitored instead of all the file descriptors that are being monitored by the program.

One of the goals of the thesis is to retrieve I/O events in a single location within the code instead of several locations. This location is the same for IPC I/O events and network I/O events. Retrieving I/O events in a single location makes the implementation more maintainable. Furthermore, it enables code reuse when processing the events. In addition, an important goal of the thesis is to perform all I/O operations in a non-blocking manner. The internal implementation of iTasks depends on the event loop being evaluated regularly. Without the event loop being evaluated, it is not possible to react to events. Performing all I/O operations in a non-blocking manner makes sure that the event loop is regularly evaluated. Furthermore, this is required for horizontally scaling iTasks applications, which is a future goal of the iTasks project. The way data is sent has been changed to be able to realise these goals.

The replacement implementation sends data differently depending on whether a reactive or proactive I/O multiplexing is used. First, the process of sending data for reactive I/O multiplexing mechanisms is described. The reactive mechanisms that are used are `kqueue` (macOS) and `epoll` (Linux). This is followed by an explanation to how data is sent by `IOCP`. `IOCP` (Windows) is a proactive I/O multiplexing mechanism. The difference between proactive and reactive I/O multiplexing mechanisms is described in section **3.2**.

The overview included below illustrates how data is sent by the reactive I/O multiplexing mechanisms. As stated, the reactive mechanisms are `kqueue` (macOS) and `epoll` (Linux).

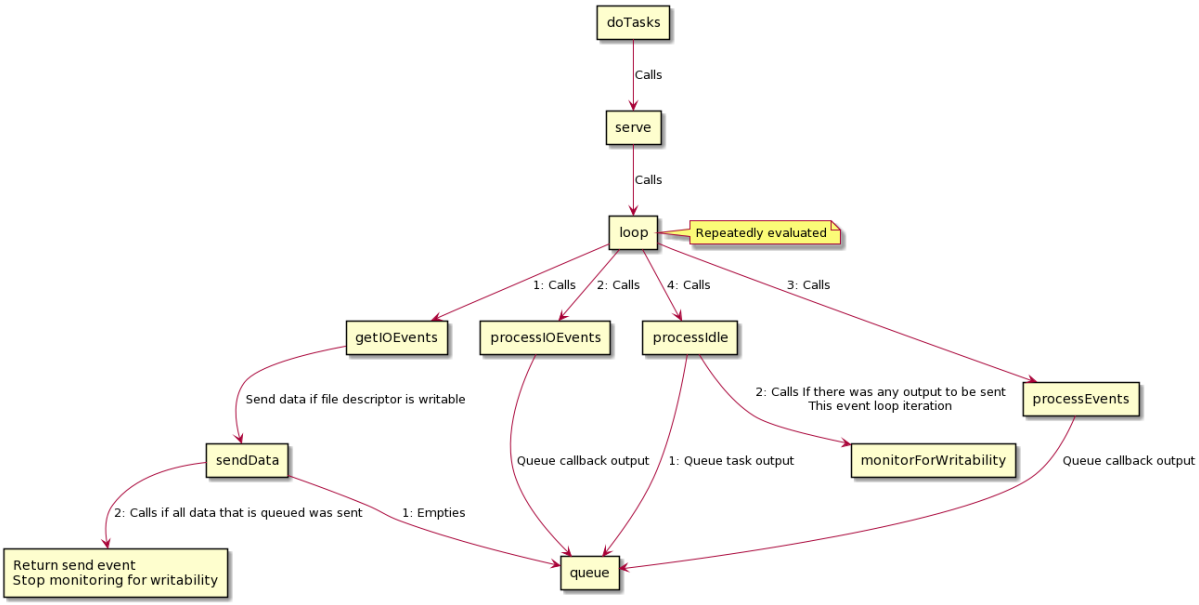


Figure 3.2: Sending data on Linux/macOS

When using reactive mechanisms, storing the data to be sent is required. This is a result of the replacement implementation not attempting to send data unless the file descriptor is writable. As a result of evaluating callbacks or evaluating tasks, output to be sent over a socket or pipe is queued.

In the replacement implementation, events are retrieved once every iteration of the event loop. Retrieving the I/O events is the first thing that happens in an event loop iteration. As a result, detecting that a file descriptor is writable may only happen in iterations that follow the iteration where the data itself is queued.

It may be possible that a file descriptor is not writable during the next iteration of the event loop but more data is queued. To handle this situation, the data that has to be sent is stored. To store the data, a write queue is used. The reasoning behind using a queue is that the operations that are used (enqueuing, dequeuing, checking whether the queue is empty) can be performed in constant time. Furthermore, using a queue naturally allows to maintain the order of the data which is to be sent.

Once the file descriptor becomes writable, the write queue is emptied and the data it contains is sent in a non-blocking manner. This happens in order. When the queue is empty, the file descriptor is no longer monitored for writability until there is more data to be sent. The `processIdle` function is used to monitor the file descriptor for writability if any data was queued during an event loop iteration.

If the file descriptor stops being writable while data is being sent, the file descriptor remains being monitored for writability. During further evaluations of the event loop, the file descriptor may become writable again. When this happens, another attempt is made at sending the data contained in the queue.

The overview included below illustrates how data is sent when `IOCP` (Windows) is used. `IOCP` is a proactive I/O multiplexing mechanism.

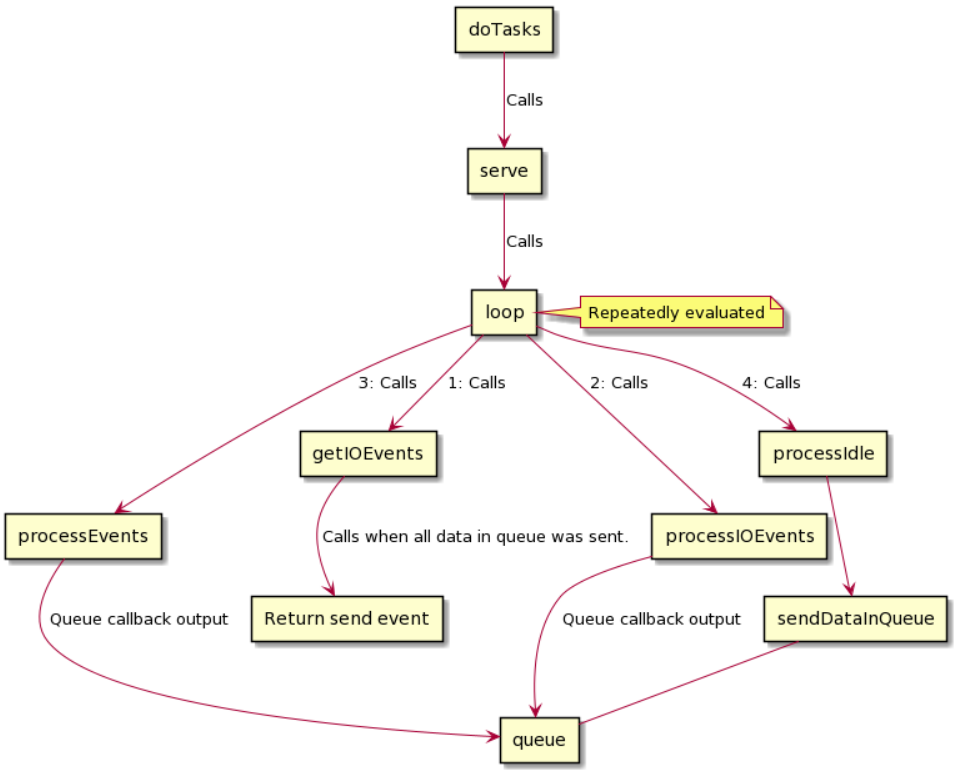


Figure 3.3: Sending data on Windows

When using IOCP, there is no notion of readability/writability as IOCP is a proactive I/O multiplexing mechanism. In the case of using IOCP the data would not have to be queued. However, doing so makes the internal implementation more harmonious. On Windows, `processIdle` initiates asynchronous send

operations for the data contained within the write queue. As a consequence, the data is sent over to the peer. The data is sent over the connection in order.

In conclusion, sending data may be delayed until a later moment thanks to both approaches for sending data. Being able to delay the sending of data is required for implementing non-blocking I/O. Performing I/O in a non-blocking manner is beneficial because it becomes impossible for an iTasks to block in regard to performing network I/O and IPC operations. As a result, the event loop is always evaluated regularly. In addition, this is necessary step for being able to horizontally scale iTasks applications, which is a future goal of the iTasks project. Furthermore, the new approach does not require retrieving events for an individual file descriptor. As a result, events can be retrieved in a single location.

The trade-off of this approach is that it complicates the process of sending data. As iTasks is a platform-independent library, the reactive and proactive approach required to be unified to a single platform-independent implementation. In addition, requiring the sending of data to be delayed to a later point in time naturally makes the implementation more complicated. As a result, the implementation of sending data is more complicated than the existing one.

3.4 Discussion

Like the existing implementation, the replacement implementation provides an HTTP server implementation. Furthermore, the `tcplisten` and `tcpconnect` tasks are provided through the same types as the existing implementation. The aforementioned tasks are used to provide network I/O to the end user. The replacement implementation also features an `externalProcess` task that allows to execute an external process and communicate with it. The replacement implementation also introduces a new task called `externalProcessHandlers`. This task allows to define how communication should occur in a similar style as the `tcplisten` and `tcpconnect` tasks.

Unlike the existing implementation, the replacement implementation provides IPC and network I/O through a single approach. To make this possible, the `select` I/O multiplexing mechanism was replaced for the `IOCP`, `kqueue` and `epoll` I/O multiplexing mechanisms. The network I/O and IPC functionality that is provided by the replacement implementation rely on the replacement I/O multiplexing mechanisms. Pipes (IPC) are now monitored using an I/O multiplexing mechanism instead of the time-based approach used by the existing implementation (see section **2.3.3**). The retrieval and processing of I/O events is unified to a single location instead of a several locations throughout the code. Furthermore, all I/O operations are performed in a non-blocking manner. This is beneficial as this ensures that iTasks applications will not block when performing I/O. This is required for horizontally scaling iTasks applications, which is a future goal of the iTasks project. Events are retrieved in a single location for both the IPC and network I/O functionality.

A drawback to the replacement implementation is that different I/O multiplexers are used on different operating systems instead of a single mechanism. This complicates the implementation. Furthermore, implementing operations in a non-blocking manner complicates the internal implementation as well. This is especially the case for sending data.

Chapter 4

Existing Implementation - The HTTP Server

iTasks is a general-purpose framework for developing web applications. To host web applications built using iTasks, it is not necessary to explicitly setup an HTTP server (e.g: nginx, Apache) to serve the website content. The internal implementation of iTasks sets up an HTTP server itself when executing an iTasks application. The HTTP server makes sure that client requests are properly processed. The processing of requests may lead to sending responses to the involved clients. As a result, the HTTP server and connected clients are able to communicate.

Usually, clients will make use of a browser to request web pages that are served by the HTTP server. In this case, the browser can be seen as the client. Clients and the HTTP server rely on network I/O (TCP) to communicate. The internal implementation of the HTTP server depends on an I/O multiplexing mechanism to perform network I/O. In the existing implementation this is the `select` I/O multiplexing mechanism.

A goal of this thesis is to replace the `select` I/O multiplexing mechanism for the `kqueue`, `epoll` and `IOCP` I/O multiplexing mechanisms. Since the I/O multiplexing mechanism is replaced, the internal implementation of the HTTP server is changed as well. This is a result of the HTTP server relying on the I/O multiplexing mechanism. This chapter provides an overview of how the HTTP server is implemented in the existing implementation. Furthermore, the chapter aims to describe how the website content is provided to the end-user when accessing an iTasks web application using a browser. Some documentation on this particular topic already existed when writing this thesis [14]. This chapter benefited from this documentation. By examining the HTTP server, the drawbacks to the existing HTTP server are identified. The replacement implementation of the HTTP server attempts to provide solutions for these drawbacks.

4.1 Communication between clients and the HTTP server

The overview included below illustrates how a browser may communicate with the HTTP server.

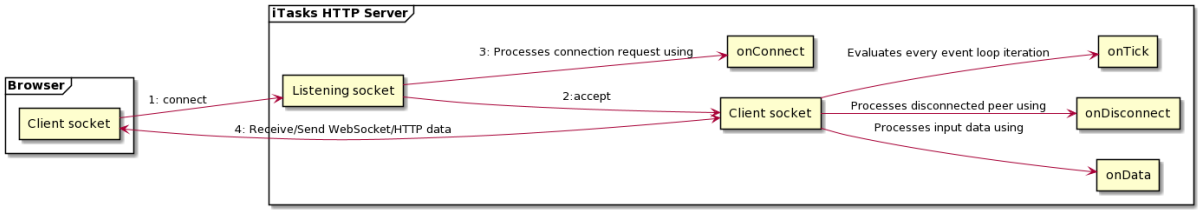


Figure 4.1: Communication between a browser (client) and the iTasks HTTP Server.

As stated, the iTasks HTTP server may communicate with clients through network I/O. The listener socket and client sockets that are involved in this communication are monitored by the I/O multiplexing mechanism. The `onConnect`, `onData` and `onTick` callbacks are used to process events and possibly

send responses (data) back to the client for which the event occurred. Note that the `onDisconnect` callback may not send data back to the client. Clients that disconnect from the HTTP server are cleaned up. The HTTP server makes use of the HTTP and WebSocket protocols to send and receive data. WebSocket and HTTP are both communication protocols which function on top of the socket network I/O communication channel.

WebSocket is the communication protocol which is used to send task events. In addition, WebSocket is used to send task event responses from the server. Task events are any events related to executing a task which is served by the iTasks web application. For example, a task event may occur because the user entered a value in a text field or submits a form.

HTTP is the communication protocol which is used to request/provide static resources (e.g: .js and .css files, icons). In addition, HTTP is used to establish the WebSocket connection. This is standard for the WebSocket protocol [15] (see section 1.3 of the specification). The different protocols are mentioned because the performance metrics for responding to task events and for requesting static resources were separately benchmarked when benchmarking the HTTP server.

The `onConnect`, `onData` and `onTick` callbacks are part of a callback record which is used for processing network I/O events. The type of this callback record is provided below. This is followed by an explanation of what processing is performed by the callbacks in the context of the HTTP server.

```
:: Output :== String
:: Close :== Bool
:: PeerAddress :== String
:: Data :== String
:: FD :== Int
:: ConnectionHandlersIWorld l r w =
    { // Used by HTTPServer.
      onConnect      :: FD -> PeerAddress -> r -> *IWorld
        -> *(MaybeErrorString l, ? w, [Output], Close, *IWorld)
    , onData         :: Data -> l -> r -> *IWorld
        -> *(MaybeErrorString l, ?w, [Output], Close, *IWorld)
    , onTick         :: l -> r -> *IWorld
        -> *(MaybeErrorString l, ? w, [Output], Close, *IWorld)
      // Unused by HTTPServer.
    , onShareChange  :: l -> r -> *IWorld
        -> *(MaybeErrorString l, ? w, [Output], Close, *IWorld)
    , onDisconnect   :: l -> r -> *IWorld
        -> *(MaybeErrorString l, ? w, *IWorld)
    , onDestroy      :: l -> *IWorld
        -> *(MaybeErrorString l, [Output], *IWorld)
    }
```

Some functions in the callback record are not actively used by the HTTP server. The unused callbacks are present to allow end-user network I/O events to be processed in the same way as HTTP server I/O events. End-user Network I/O that is provided by iTasks may use the callbacks that are not used by the HTTP server. The HTTP server implementation defines callbacks that do not have any effects for the callbacks that are not actively used. This does not mean that disconnected clients do not get cleaned up since there is no `onDisconnect`, however. The callbacks are used to define specific behavior and disconnected clients are cleaned up by default. The callbacks in the callback record provided to the HTTP server are processed in the same manner as the callbacks provided to the end-user network I/O tasks. This is possible because the callbacks provided to the end-user network I/O tasks are converted to a callback record of the type included above. As a result, the internal implementation of the HTTP server is very similar to the internal implementation of end-user Network I/O and a significant amount of code is reused. Chapter **6** describes how end-user network I/O is provided in the existing implementation.

As stated, the `onConnect`, `onData` and `onTick` callbacks are used by the HTTP server to process events. The processing that is performed by these callbacks for the HTTP server is described below:

1. When a client connects to the HTTP server, a connection event is returned by the I/O multiplexing mechanism. This connection event is processed by evaluating the `onConnect` callback. In the case of the HTTP server, the `onConnect` callback verifies that the IP address of the client is on the allowlist of the HTTP server. The allowlist defines which IP addresses may access the HTTP server.

2. An I/O event may occur when data is received from a client. Receiving data results in evaluating the `onData` callback. The `onData` callback is used to process client requests. This could be a request for a web page that is served by the HTTP server. Furthermore, the `onData` callback is used to process task events. A task event may occur when the clients enters a value in a text field on the website or submits a form, for instance. The `onData` callback parses the HTTP/WebSocket requests that are sent by clients. Furthermore, the `onData` callback may provide output data. As a result, a response – containing the output data – may be sent back to the client. To give an example, this response could consist of the web page which the client requested. Essentially, the response is text which is interpreted by the browser, leading the browser to visualize a web page.

3. The `onTick` callback is evaluated every event loop iteration. As a reminder, iTasks makes use of an event loop which is repeatedly evaluated throughout the execution of the program. The event loop is used to retrieve and process events. The onTick callback ensures that the connected clients are still active. The clients that are connected to the HTTP server regularly send keep-alive data to the HTTP server and the `onTick` callback is used to close the connections of clients that did not send their keep-alive data in time.

   Furthermore, the processing of task events of a client may lead to task output. Task output is stored in a SDS (Shared Data Source). During the `onTick` function, the Task output for the given client is sent. In the context of this thesis, it is not necessary to completely understand this process.

   In addition, clients use the HTTP server to work on tasks. Multiple clients may work on different instances of the same task. During the `onTick` it is verified that the instances of the task the client is working on are the instances of the task started by the client. This is done to prevent clients affecting task instances of other clients. Each task instance may only have a single client working on the instance.

   The way the `onTick` callback is processed by the existing implementation could be improved. The improvements that could be made are described in more detail in section **4.2**.

A more concrete example is provided below. This example is included to show how the callbacks are used when interacting with clients. Requesting a page hosted by an iTasks web application using the browser generally leads to the following sequence of actions:

1. The browser of the client establishes a connection to the listener socket of the HTTP server.

2. The HTTP server evaluates the `onConnect` callback, which leads to verifying that the IP address of the client is contained within the allowlist of the HTTP server. If this is not the case, the server sends an HTTP 403 error page to the client and closes the connection.

3. If the client is part of the allowlist, the client will proceed by requesting a web page from the HTTP server.

4. The request will lead to the `onData` callback being evaluated. The `onData` callback processes the request and responds with the web page.

5. The client receives the web page and the browser will detect that the requested web page depends on static resources. Examples of static resources are javascript/css files and icons. The browser detects which static resources are required and requests the static resources from the iTasks HTTP Server using HTTP requests. The iTasks HTTP server processes the requests and responds with the static resources that were requested, using the `onData` callback.

6. A static resource dependency included for any iTasks web page is a javascript file called `itasks-core.js`. This javascript file is obtained from the HTTP server. When the requested web page loads, javascript is used to establish a WebSocket connection to the iTasks HTTP Server. The WebSocket request is processed by the `onData` callback. WebSocket makes use of regular network

I/O sockets as a underlying communication channel. The WebSocket communication protocol is used to process task events.

7. Using the WebSocket connection, the client attaches to a particular instance of a task by sending the server a WebSocket message, which is processed by the `onData` callback. As a result, the server is aware of the particular task the client is working on.
8. As a result of attaching to the task, the servers sends data to the client which is used to determine which UI components should be included on the web page. The UI components are then visualized on the web page by the browser (client) using javascript. As a result, the browser will visualize the input fields, submit buttons and other UI components that are related to performing the task. In the existing implementation, this data is sent through the `onTick` callback. The replacement implementation takes a different approach to the processing that is done by the `onTick` callback in the existing implementation. This approach is described in chapter **5**.
9. Furthermore, javascript is used to monitor the web page for user interaction, such as the user editing an input field or submitting a form. The WebSocket connection is used to send events indicating such user interaction to the iTasks HTTP server. The iTasks HTTP server then responds to these events using the WebSocket connection. This response may then lead to changes to the UI of the web page.

4.2 Internal implementation

The internal implementation of the HTTP server closely resembles the internal implementation of the `tcplisten` task. The internal implementation of the `tcplisten` task is described in section **6**. The call graph below illustrates how the iTasks HTTP server is created and monitored for I/O:

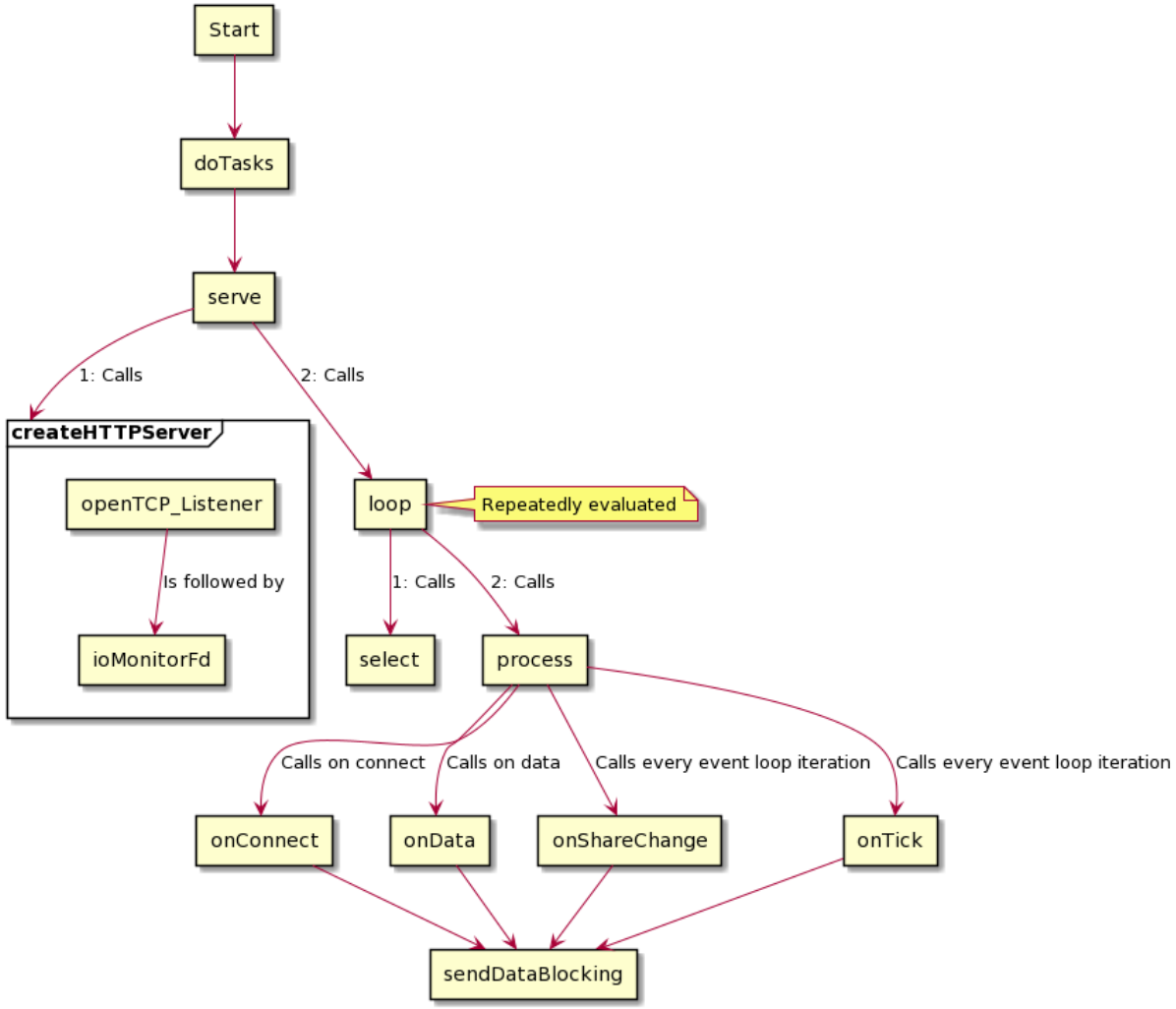


Figure 4.2: Internal implementation of the iTasks HTTP server

If the specification of the iTasks program involves a web-based task, an HTTP server is created to serve the task. A web-based task is a task that relies on the HTTP server to be executable. Setting up an HTTP server happens through the `serve` function. The `serve` function:

1. Creates a listener socket for the HTTP server.
2. Monitors the listener socket for incoming connections using the I/O multiplexing mechanism.
3. Starts the event loop by calling the `loop` function.

The `loop` function:

1. Retrieves the I/O events on the file descriptors that are being monitored using `select`. This will include the listener socket of the HTTP server. Furthermore, this includes the sockets of the clients that are connected to the HTTP server.
2. Processes the I/O events that were retrieved using `process`. In the case a client connected to the HTTP server, the `onConnect` callback is processed. In the case data was received on a client socket that is connected to the HTTP server, the `onData` callback is processed. The `onConnect` and `onData` callbacks may produce output which is sent in a blocking manner.
3. Performs the processing that should happen every iteration of the event loop for the sockets that are being monitored using `process`. This means that, for every connected client, the `onTick` and

`onShareChange` callback are processed. This happens every event loop iteration for every client that is connected to the HTTP server. The `onShareChange` and `onTick` callback may send data in a blocking manner.

The `onShareChange` callback is not actively used by the HTTP server. However, it may be used by end-user network I/O. Since I/O events are processed the same way for the HTTP server and end-user network I/O, the callback is still evaluated. In the existing implementation, the `onShareChange` callback of the connected clients is evaluated for every client every event loop iteration. This can be considered a bug, as the `onShareChange` callback name implies that it should only be evaluated when the SDS (Shared Data Source) that is associated with the HTTP server changes. A SDS is a means to share data within an iTasks application. This does not pose problems for the HTTP server as the `onShareChange` callback defined by the HTTP server does not have any effect. However, evaluating the `onShareChange` callback involves reading from a SDS once every event loop iteration per client. This read is unnecessary in the case of the HTTP server. Reading from a SDS is a relatively expensive operation. As a result, not evaluating the `onShareChange` callback should increase the scalability of the HTTP server.

The `onTick` callback is evaluated every event loop iteration. In the onTick callback, it is checked whether the clients are still active. The clients that are connected to the HTTP server regularly send keep-alive data to the HTTP server and the `onTick` callback is used to close the connections of clients that did not send their keep-alive data in time.

Furthermore, the processing of task events of a client may lead to task output. Task output is stored in a SDS. The `onTick` function is used to the task output of the connection clients. In the context of this thesis, it is not necessary to completely understand this process. This is mentioned because this involves reading the SDS which is used to store Task Output every event loop iteration for every client. After sending the task output for a client, the SDS is written to again to clear the Task output. Reading and writing to a SDS are expensive operations. This means that the operations take a relatively long amount of time to perform.

In addition, clients use the HTTP server to work on tasks. Multiple clients may work on different instances of the same task. During the `onTick` it is verified that the instances of the task the client is working on are the instances of the task started by the client. This is to prevent clients altering task instances of other clients. This process involves reading a SDS every event loop iteration for every client. Again, reading a SDS is a relatively expensive operation.

It was discovered that instead of using the `onTick`, the task output and verification of task instances could be processed at once for all clients that use the HTTP server instead of individually through the `onTick` callback. As a result, two SDS reads and 1 SDS write would have to be performed in total per event loop iteration. The existing HTTP server uses 2 SDS reads and 1 SDS write per event loop iteration per connected client. This results in having to perform significantly less SDS reads and writes as more clients are connected to the HTTP server. As a result, this discovery lead to the goal of optimizing this process for the HTTP server in the replacement implementation.

4. Evaluates the `loop` function again, repeating the above.

Sending data is done in a blocking manner in the existing implementation. A goal when implementing the replacement implementation is to send data in a non-blocking manner. This is beneficial because having the event loop block results in the iTasks server becoming unresponsive and this should be avoided. Sending data is done using a non-blocking call. However, if this call fails the socket is individually monitored for writability using a blocking `select` call. As a result, the implementation effectively sends data in a blocking manner.

Reading data may also block in exceptional circumstances. The existing implementation waits for a readability event indicating that data may be read. After this, a non-blocking read operation is performed to read the data. However, if this non-blocking read fails, a blocking `select` call is performed to wait for readability on the involved socket. This call leads to either being able to read data or the server becoming unresponsive. As mentioned in the manual for `select` [2] (see the "Bugs" section), it is advised to use a non-blocking approach to reading instead to be safer. In the current situation, the non-blocking read could fail even though readability was indicated. At that point there is no guarantee that file descriptor

becomes readable again, making it possible that the iTasks server blocks indefinitely waiting for the file descriptor to become readable using `select`. Of course, this is a very exceptional circumstance but this could lead the iTasks server to become unresponsive. Reading in a non-blocking manner would rule out this possibility.

Another goal of the thesis is to retrieve I/O events in a single location. By using a non-blocking approach to reading and sending data, the `select` calls that are currently being performed are not necessary anymore and I/O events can be retrieved in a single location.

4.3 Summary

To summarize, the iTasks HTTP server relies on the `select` I/O multiplexing mechanism. Clients are able to communicate with the HTTP server by connecting to a listener socket. After connecting, the HTTP server uses the `onConnect`, `onData` and `onTick` callbacks to communicate with the connected client.

The following points of improvement were identified by examining the existing HTTP server:

- The processing done by the `onTick` callback should be optimized. This is done with the goal of improving the scalability of the HTTP server.
- The `onShareChange` callback should not be evaluated for the HTTP server as it is not used. This is useful as it further increases the scalability of the HTTP server.
- Reading and sending data should be performed in a non-blocking manner. This is required for being able to horizontally scale iTasks web applications, which is a future goal of the iTasks project.
- Events should be retrieved in a single location for all forms of I/O. As a result, retrieving events on individual file descriptors when non-blocking operations fail should not be done.
- The HTTP server should use the replacement I/O multiplexing mechanisms. The `select` I/O multiplexing mechanism is replaced to be able to provide IPC and network I/O through a single approach.

Chapter 5

Replacement Implementation - The HTTP server

This chapter describes the implementation of the HTTP server within the context of the replacement implementation. In addition, this chapter describes to what extent the replacement HTTP server improved upon the existing HTTP server.

The HTTP server communicates with connected clients in the same way as in the existing implementation. As a result, section **4.1** applies for the replacement implementation as well. There is one exception, which is that the callback record has been changed. The callback record used by the replacement implementation is of the following type:

```
:: Output :== String
:: Close :== Bool
:: PeerAddress :== String
:: Data :== String
:: FD :== Int
:: ConnectionHandlersIWorld l r w =
    { // Used by HTTPServer.
      onConnect      :: FD -> PeerAddress -> r -> *IWorld
        -> *(MaybeErrorString l, ? w, [Output], Close, *IWorld)
    , onData         :: Data -> l -> r -> *IWorld
        -> *(MaybeErrorString l, ?w, [Output], Close, *IWorld)
    , onTick         :: l -> *IWorld
        -> *(MaybeErrorString l, Close, *IWorld)
      // Unused by HTTPServer.
    , onShareChange :: l -> r -> *IWorld
        -> *(MaybeErrorString l, ? w, [Output], Close, *IWorld)
    , onDisconnect   :: l -> r -> *IWorld
        -> *(MaybeErrorString l, ? w, *IWorld)
    , onDestroy      :: l -> *IWorld
        -> *(MaybeErrorString l, [Output], *IWorld)
    }
```

The type of the `onTick` callback has been changed. This poses no problems for end-user network I/O and the rest of iTasks as the `onTick` callback is only used by the HTTP server. Due to the optimizations that were made to the `onTick` , the task output SDS value is no longer provided as it is no longer needed. These optimizations are described in more detail in section **5.1**. As the `onTick` callback is now only used for checking that the connected clients are still active, there is no need to send output anymore. The new implementations of the end-user network I/O tasks now convert the provided callback record to a callback record of this type instead. As a result, processing events for the HTTP server and end-user network I/O can be done in the same manner in the replacement implementation as well.

5.1 Internal implementation

The internal implementation of the HTTP server closely resembles the internal implementation of the `tcplisten` task. The internal implementation of the `tcplisten` task is described in section **7**.

The call graph below illustrates how the iTasks HTTP server is created and monitored for I/O:

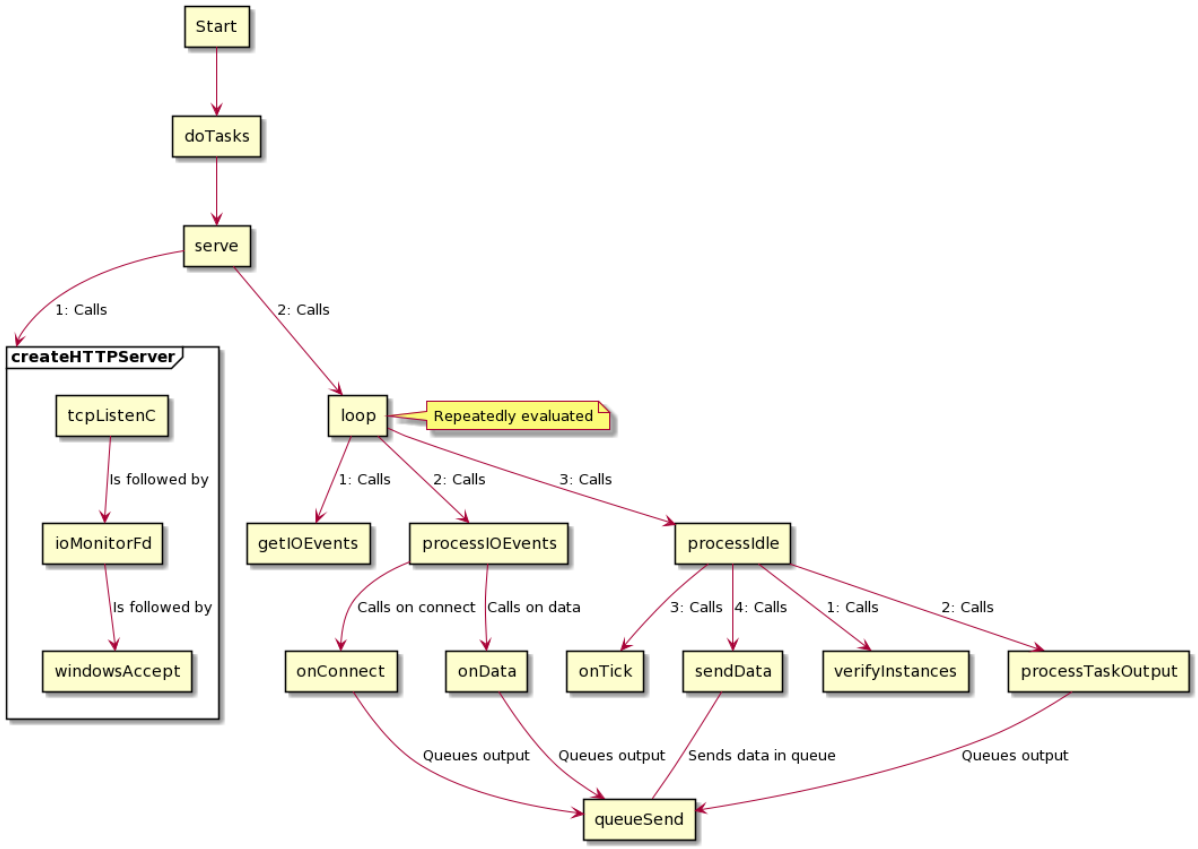


Figure 5.1: Internal implementation of the iTasks HTTP server

If the specification of the iTasks program involves a web-based task, an HTTP server is created to serve the task. This happens through the `serve` function. The `serve` function:

1. Creates a listener socket for the HTTP server.
2. Monitors the listener socket for incoming connections using the I/O multiplexing mechanism.
3. Proactively accepts a connection request on the listener socket when using the `IOCP` I/O multiplexing mechanism (Windows). This is required because `IOCP` is a proactive I/O multiplexing mechanism. The difference between reactive and proactive I/O multiplexing mechanisms is described in section **3.2**.
4. Starts the event loop by calling the `loop` function.

The `loop` function:

1. Retrieves the I/O events on the file descriptors that are being monitored. This is done using a replacement I/O multiplexing mechanism depending on the target platform. `kqueue` is used on macOS, `epoll` is used on Linux and `IOCP` is used on Windows. The set of monitored file descriptors contains the listener socket of the HTTP server. In addition, the set of monitored file descriptors contains the sockets of the clients that are connected to the HTTP server.

2. Processes the I/O events that were retrieved on the monitored file descriptors. When a client connects to the HTTP server, the `onConnect` callback is processed. In the case of receiving data on a client socket that is connected to the HTTP server, the `onData` callback is processed. Data is now read in a non-blocking manner. Furthermore, instead of sending the data in a blocking manner, the `onConnect` and `onData` callbacks queue output that should be sent. This approach allows to delay sending data. This is useful when sending data in a non-blocking manner.

3. Performs the idle processing that is related to the sockets that are being monitored. Idle processing involves the processing that is done every event loop iteration, independent of the I/O multiplexing mechanism returning events. This means that, for every connected client, the `onTick` callback is processed. This happens every event loop iteration for every client that is connected to the HTTP server. In the replacement implementation, the `onTick` callback is used to verify that the connected clients sent their keepalive data in time, indicating that the client is still alive. Sending the task output – which used to happen through the `onTick` callback for each individual client – is now done at once for all clients using the `processTaskOutput` function. Similarly, verifying that the task instances which the clients are attached to are theirs is also done at once. The new approach to processing the `onTick` has the same effect but scales significantly better. This is shown in the benchmark of the HTTP server (see figure **10.2.1**).

   As stated, in the replacement implementation, output that is to be sent is queued. The `processIdle` function makes sure the queued output is sent over to the peer in a non-blocking manner. The process of sending data within the replacement implementation is described in more detail in section **3.3**.

4. Evaluates the `loop` function again, repeating the above.

5.2 Summary

The replacement implementation provides solutions for the points of improvement that were listed for the existing implementation of the HTTP server (see section **4.3**). This means that, in the replacement implementation of the HTTP server:

- The processing that used to be done by the `onTick` callback has been optimized. This lead to a significant scalability improvement as is shown by the benchmark of the HTTP server, see figure **10.2.1**.
- The `onShareChange` callback is not evaluated for the HTTP server as it is not used but does involve an expensive SDS read for every connected client for every event loop iteration. This increases the scalability of the iTasks HTTP server as well.
- Reading and sending data is performed in a non-blocking manner. This ensures that the event loop does not block when handling network I/O and IPC events. This is useful because iTasks may not respond to other events when the implementation does block. This is required for horizontally scaling iTasks applications.
- Events are retrieved in a single location for all forms of I/O that are provided by the replacement implementation (network I/O and IPC). This simplifies the implementation.
- The HTTP server makes use of the replacement I/O multiplexing mechanisms. That means that `epoll` is used on Linux, `kqueue` is used on macOS and `IOCP` is used on Windows. This allows to provide a conceptually equivalent model for providing IPC and Network I/O.

Chapter 6

Existing Implementation - Network I/O

This chapter aims at giving an overview of how the existing implementation provides network I/O functionality to the end-user. In iTasks, the ability to perform network I/O is provided to the end-users through the `tcplisten` and `tcpconnect` tasks. This chapter provides an end-user perspective which describes how the `tcplisten` and `tcpconnect` tasks can be used. Furthermore, this chapter provides an overview of the internal implementation of the `tcplisten` and `tcpconnect` tasks. The drawbacks to the existing implementation of the aforementioned tasks are identified. This is done with the aim of providing solutions for these drawbacks in the replacement implementation.

The `tcplisten` task is used to setup a listening socket which listens for connection requests over TCP. This essentially sets up a TCP server. The end-user provides the `tcplisten` task a callback record which defines how the TCP server should communicate with the clients that connect to the listening socket. As a result, the `tcplisten` task is conceptually similar to the HTTP server. The HTTP server has been discussed in the two preceding chapters. The HTTP server makes use of predefined callbacks while the `tcplisten` task makes use of callbacks that are defined by the end-user.

The `tcpconnect` task is used to connect to a TCP server as a client. The end-user provides this task a callback record which defines how the client communicates with the TCP server it connected to.

The end-user may make use of a custom communication protocol when using `tcplisten` and `tcpconnect`. For example, an iTasks extension which allows to send email using the SMTP protocol makes use of `tcpconnect`.

For the most part, the `tcplisten` and `tcpconnect` tasks abstract from the internal implementation. As a consequence, the end-user does not need to be aware of the internal implementation to make use of network I/O. For instance, to use network I/O operations, the end-user does not need to be aware of the I/O multiplexing mechanism at all. Nonetheless, the `tcplisten` and `tcpconnect` tasks rely on the `select` I/O multiplexing mechanism internally. The `select` I/O multiplexing mechanism is used to monitor the sockets that are involved in network I/O communication. As a result, the program can react to a listening socket receiving a connection request, for instance.

6.1 End-user perspective

The `tcplisten` task sets up a TCP listener which listens for connections. The `tcpconnect` task connects to a TCP listener. An overview of how these tasks could interact is included below.

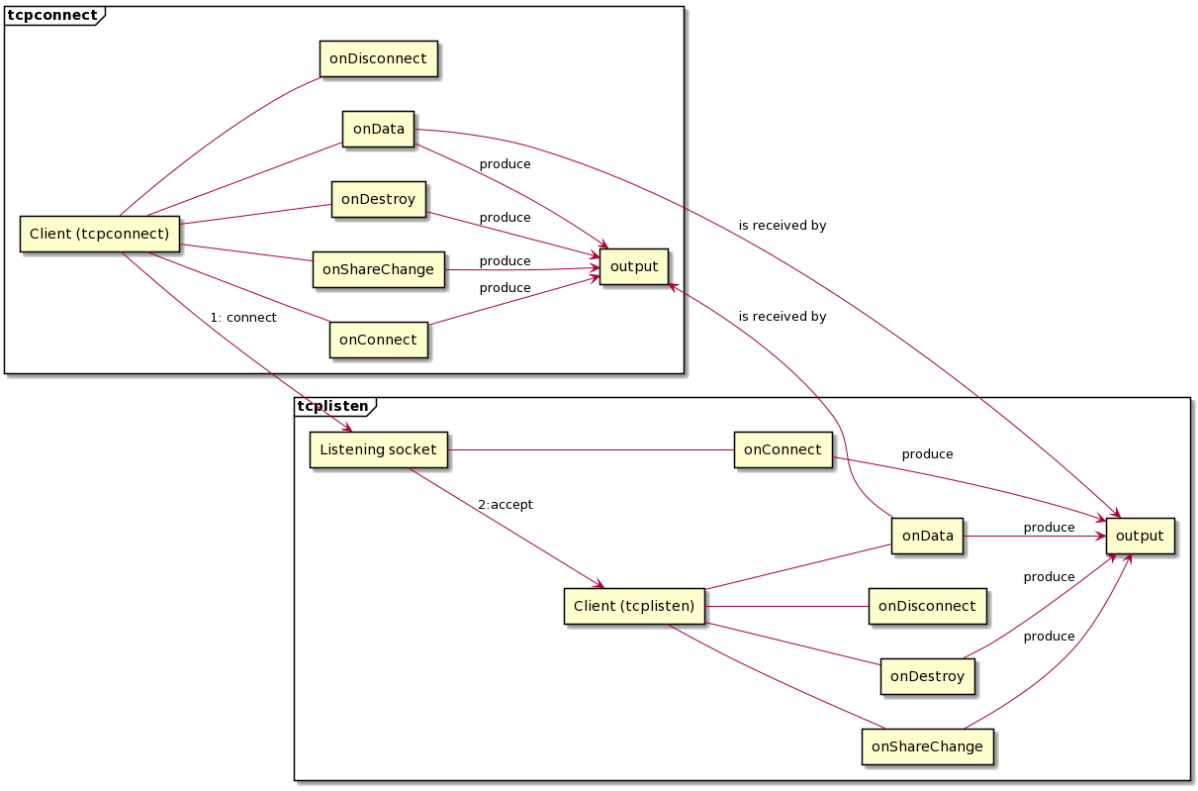


Figure 6.1: How iTasks provides network I/O to end-users.

The above figure gives an overview of how the `tcplisten` and `tcpconnect` tasks could interact. Note that `tcplisten` does not require that clients that connect use the `tcpconnect` task, any TCP connection request may be accepted. Likewise, `tcpconnect` does not require that the server that is connected to uses the `tcplisten` task, it can connect to any TCP server. Similarly to how `tcplisten` and `tcpconnect` abstract from the internal implementation, this overview hides the internal implementation as well.

When a client successfully connects to the server, accepting the connection will return a socket that can be used to communicate with the client that connected. This is illustrated by the `accept` operation leading to the creation of a client socket. Both the `tcpconnect` and `tcplisten` tasks take a record of callback functions that are provided by the end-user. The record contains the following callbacks:

- `onConnect`, which is a callback that is evaluated when a TCP connection is established. In the case of using `tcpconnect`, this callback is evaluated after successfully connecting to the server. In the case of using `tcplisten`, this callback is evaluated whenever a client successfully connected to the server.
- `onData`, when the socket receives data, this callback is evaluated.
- `onShareChange`, this callback is invoked every event loop iteration (which is considered to be a bug).
- `onDisconnect`, when a peer (socket on the other end of the connection) disconnects, this callback is evaluated.
- `onDestroy` when the task is destroyed, this callback is evaluated. This happens when the iTasks program terminates, for instance.

Note that the `onTick` callback – which is used by the HTTP server – is not defined by the end-user. `tcplisten` and `tcpconnect` convert the callback record provided by the end-user to a callback record of the same type as the HTTP server. This is done to be able to process events for the HTTP server and

end-user network I/O in the same manner. As a result, `tcplisten` and `tcpconnect` internally define a callback that has no effect for the `onTick` callback.

As the overview illustrates, evaluating certain callbacks in the figure results in output. This output is sent over the network connection. The I/O multiplexing mechanism monitors the clients for readability. When a client socket becomes readable and data is received, the `onData` callback is evaluated. The `onData` callback may produce output and send data back to the peer. This enables clients and servers to communicate. At this point, the general idea of how the `tcplisten`/`tcpconnect` tasks provide network I/O to the end-users has been described. This is followed by a description of how the `tcplisten` and `tcpconnect` tasks may be used by the end-user.

The `tcplisten` task has the following type:

```
tcplisten :: Int Bool (sds () r w) (ConnectionHandlers l r w) -> Task [l]
          | iTask l & iTask r & iTask w & RWShared sds
```

The `tcplisten` task takes the following arguments:

- A port to listen on.
- A boolean indicating whether the connection state of the client should be removed when a client disconnects.
- A shared data source (SDS) which is used to give the callbacks access to a custom value of a type determined by the end-user. The callbacks may modify this value.
- A record of callbacks (`ConnectionHandlers`) which are used to react to certain events occurring.

```
:: Output :== String
:: Close :== Bool
:: PeerAddress :== String
:: Data :== String
:: ConnectionHandlers l r w =
   { onConnect      :: ConnectionId -> PeerAddress -> r
       -> (MaybeErrorString l, (? w), [Output], Close)
   , onData         :: Data -> l -> r
       -> (MaybeErrorString l, (? w), [Output], Close)
   , onShareChange  :: l -> r
       -> (MaybeErrorString l, (? w), [Output], Close)
   , onDisconnect   :: l -> r -> (MaybeErrorString l, (? w))
   , onDestroy      :: l -> (MaybeErrorString l, [Output])
   }
```

The `tcpconnect` task has the following type.

```
tcpconnect :: String Int (?Timeout) (sds () r w)
              (ConnectionHandlers l r w) -> Task l
              | iTask l & iTask r & iTask w & RWShared sds
```

The `tcpconnect` task takes the following arguments:

- An IP address to connect to.
- A port to connect to.
- Optionally, a timeout value. If there is no timeout value given, the connection attempt will block until it is successful. Otherwise, it will block for the duration of the timeout value.
- Similar to `tcplisten`, `tcpconnect` takes a SDS and `ConnectionHandlers` record.

An example program which evaluates the `tcplisten` task on startup is included below:

```
module listen

import iTasks

Start w = doTasks (onStartup listen) w

listen :: Task [String]
listen = tcplisten port removeOnClose null handlers
where
    port = 1234
    removeOnClose = False

connect :: Task String
connect = tcpconnect "localhost" port timeout null handlers
where
    port = 80
    timeout = ?None

null :: SDSSource () () ()
null = nullShare

handlers :: ConnectionHandlers String () ()
handlers = {ConnectionHandlers|
            onConnect = \cid h r -> (Ok "initConState", ?None, [h], False)
           ,onData = \data conState r -> (Ok conState, ?Just r, ["out"],
              True)
           ,onShareChange = \conState r -> (Ok conState, ?None, [], False)
           ,onDisconnect = \conState r -> (Ok "newConState", ?None),
           ,onDestroy = \conState -> (Ok conState, ["out"])
           }
```

In the above program, the `listen` task is evaluated on startup. This leads to the `tcplisten` task being evaluated. The `tcplisten` task starts listening for TCP connections on the specified port, 1234 in this case. The handlers record provides callback functions that are used to communicate with the clients that connect to the TCP listener. Similarly, the `connect` task could have been evaluated on startup instead to connect to a TCP listener that listens on port 80. The `tcpconnect` task also takes a `ConnectionHandlers` record as a parameter. This record is used to define how the iTasks program should interact with the server it connected to in the case of using `tcpconnect`.

6.2 Internal implementation - Overview

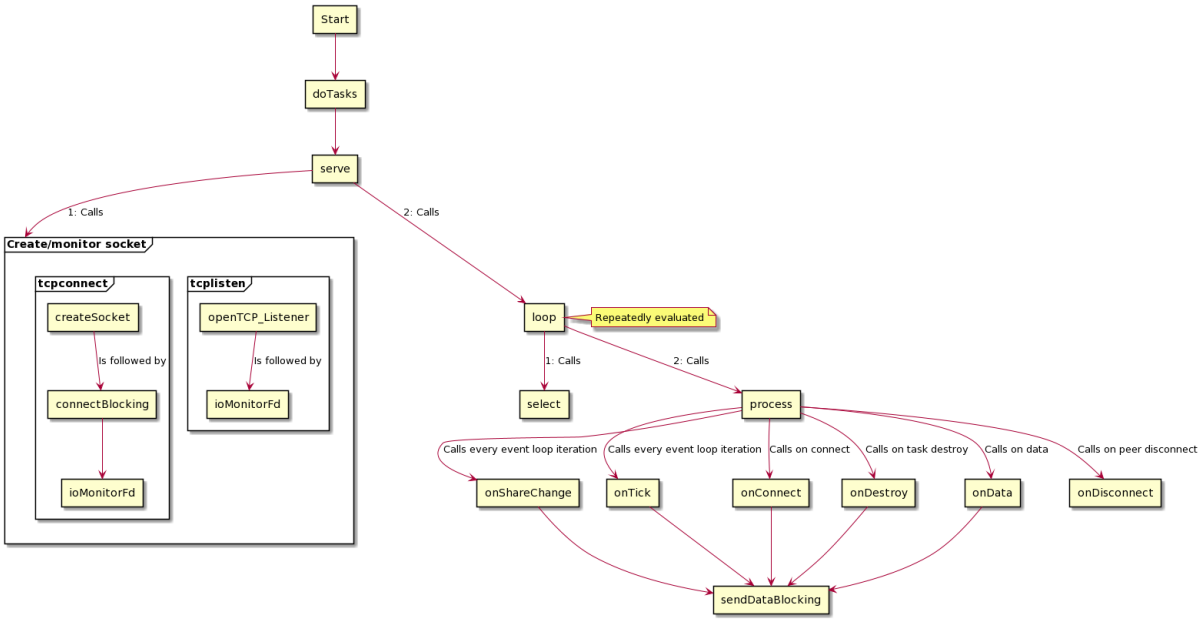


Figure 6.2: Overview of existing network I/O implementation.

The internal implementation of the tcplisten task closely resembles the internal implementation of the HTTP server. The HTTP server has been described in chapter **4**. Like the HTTP server, the internal implementation of end-user network I/O relies on the `select` I/O multiplexing mechanism.

The internal implementation is described according to the example program included in section **6.1**. Evaluating the `doTasks` function from the `Start` function signifies that an iTasks program is used. The `doTasks` function will call the `serve` function. The `tcplisten` or `tcpconnect` tasks are executed `onStartup`. As a result, the `serve` function starts a process which leads to evaluating the `tcpconnect` or `tcplisten` task, depending on which one is used.

Evaluating the `tcpconnect` task results in a socket being created. This socket is used to establish a connection to the server that is specified by the end user. The connection is established in a blocking manner. As a reminder, one of the goals of the thesis is to implement all of the I/O operations in a non-blocking manner in the replacement implementation. As a result, a goal for the replacement implementation is to establish connections in a non-blocking manner instead. When the connection attempt is accepted, the socket is monitored for I/O activity using the `select` I/O multiplexing mechanism.

Evaluating the `tcplisten` task results in a listening socket being created. This listening socket starts listening for TCP connections on the port that was specified by the end-user. Afterwards, this file descriptor is monitored for readability using the `select` I/O multiplexing mechanism. As a result, whenever a client attempts to connect to the server, the socket becomes readable. The existing implementation reacts to the socket being readable by accepting the connection request. This results in a client socket being created, which is monitored by the `select` I/O multiplexing mechanism. Client sockets created through `accept` (`tcplisten`) and `connect` (`tcpconnect`) are monitored in the same manner. Events on such client sockets are processed in the same manner as well.

At this point, the sockets involved in I/O communication are created and monitored by the `select` I/O multiplexing mechanism. This is the case regardless of whether `tcplisten` or `tcpconnect` is used. The `serve` function proceeds by starting the event loop. This happens by evaluating the `loop` function.

The `loop` function calls `select` to retrieve I/O events on the monitored sockets. It then performs processing for the monitored sockets through the `process` function. The `process` function evaluates the

callbacks that are provided to `tcplisten` and `tcpconnect` by the end-user. The `onShareChange` and `onTick` are evaluated every event loop iteration for every client that is being monitored.

In the case of `onShareChange` this is considered to be a bug. The name of the `onShareChange` callback implies that the callback should only be evaluated when the SDS provided to `tcplisten` or `tcpconnect` is modified. As a result a goal of the replacement implementation is to implement an `onShareChange` that is only evaluated when the SDS provided to `tcplisten` or `tcpconnect` changes.

The `onTick` callback does not have any effect when using `tcplisten` and `tcpconnect`. It may not be specified by the end-user. However, evaluating this callback involves reading a SDS in the existing implementation. This is a relatively expensive operation. The `onTick` callback is only used by the HTTP server. The alterations to the HTTP server resulted in the `onTick` not longer requiring to read the SDS. As a result, this read is also saved for the clients that are monitored as a result of the `tcplisten` and `tcpconnect` tasks.

The `onConnect` callback is evaluated when a connection has successfully been established. The `onData` callback is evaluated when data is received on a socket that is monitored.

The `onDisconnect` callback is evaluated when the peer closed the connection. By default, the socket used to communicate with the peer is closed.

The callbacks that are provided to `tcplisten` and `tcpconnect` may return output to be sent to the peer. The output is sent in a blocking manner. This could lead to the iTasks server blocking which is undesirable as it render iTasks unable to respond to other events. As stated, a goal of the thesis is to implement all I/O operations in a non-blocking manner. As a consequence, a goal for the replacement implementation is to send the data in a non-blocking manner instead.

6.3 Summary

iTasks provides the `tcplisten` and `tcpconnect` tasks that allow end-users to perform network I/O.

The `tcplisten` task is used to setup a TCP server which listens for connections. The end-user may define how the server communicates with the clients that connect with it through a callback record.

The `tcpconnect` task is used to setup a connection to a TCP server. The end-user provides `tcpconnect` a callback record. The callbacks in this callback record define how the TCP server that was connected to should be communicated with.

When examining the `tcplisten` and `tcpconnect` tasks, the following drawbacks were identified:

- The `onShareChange` callback is evaluated every event loop iteration for every monitored client. This is considered to be a bug as `onShareChange` implies that the callback should only be evaluated when the SDS changes.
- The callbacks provided to the `tcplisten` and `tcpconnect` tasks may return output. This output is sent to the peer in a blocking manner. A general goal of the thesis is to perform all I/O operations in a non-blocking manner. As a result, the implementation for sending data should be altered such that data is sent in a non-blocking manner.
- Data may also be read in a blocking manner. Since the existing implementation of network I/O only reads if the socket is readable, this will usually not cause trouble. However, in exceptional circumstances reading data may block even though the socket is readable [2] (see "Bugs" section). Therefore, reading data should be implemented in a non-blocking manner as well to be safe.
- The `tcpconnect` task establishes a connection the TCP server in a blocking manner. A goal of the thesis is to implement all I/O operations in a blocking manner. Therefore, the connection should be established in a non-blocking manner in the replacement implementation.
- The `tcplisten` and `tcpconnect` implementation rely on the `select` I/O multiplexing mechanism. A general goal of the thesis is to replace the `select` I/O multiplexing mechanism to be able to provide network I/O and IPC through a single concept. The existing implementation uses two separate concepts for providing network I/O and IPC. As a result, the `tcplisten` and `tcpconnect`

tasks should be re-implemented such that they make use of the replacement I/O multiplexing mechanisms instead.

Chapter 7

Replacement Implementation - Network I/O

This chapter discusses the implementation of end-user network I/O in the context of the replacement implementation. The existing implementation of end-user network I/O has been described in the preceding chapter. Like the existing implementation, the replacement implementation provides the `tcpconnect` and `tcplisten` tasks for performing network I/O. The `tcplisten` task sets up a TCP server which listens for connections. The `tcpconnect` task is used to connect to a TCP server as a client.

First, the chapter recaps how `tcpconnect` and `tcplisten` may be used by the end-user. This is followed by a description of the internal implementation of `tcplisten` and `tcpconnect`. The internal implementation of `tcplisten` and `tcpconnect` is very similar to the internal implementation of the HTTP server that was discussed in chapter **5**. This chapter contains a more technical and detailed description of the internal implementation of network I/O compared to the explanation of the HTTP server that is provided in chapter **5**. The internal implementation is described by first providing a simple overview of the internal implementation of end-user network I/O. Afterwards, the chapter zooms in on this overview and explains some of the underlying technical details that are important for making the implementation work. Furthermore, this chapter discusses to what degree the replacement implementation improved upon the existing implementation.

7.1 Providing network I/O to the end-user

Network I/O is provided to end-users in the exact same manner as in the existing implementation (with one exception). As a consequence, almost everything that is described in section **6.1** applies for the replacement implementation. The only exception is that an attempt was made at only evaluating the `onShareChange` when the SDS provided to the `tcplisten` or `tcpconnect` task is changed. This used to happen every iteration of the event loop. The approach used by the replacement implementation applies an interesting idea but the solution that is used is considered to be fragile. The solution is fragile in the sense that the `onShareChange` could still be evaluated more often than is necessary. This is discussed in more detail in chapter **12**.

An objective while developing the replacement implementation was to provide network I/O to end-users in the same manner. The reasoning behind this being an objective is that it makes it easier to integrate the replacement implementation within iTasks. Conceptually, it is only necessary to perform changes to the internal implementation. The replacement I/O multiplexing mechanisms are all able to provide similar indications for network I/O operations as the indications that `select` provides. As a result, it is possible to provide network I/O to the end-users in the same manner.

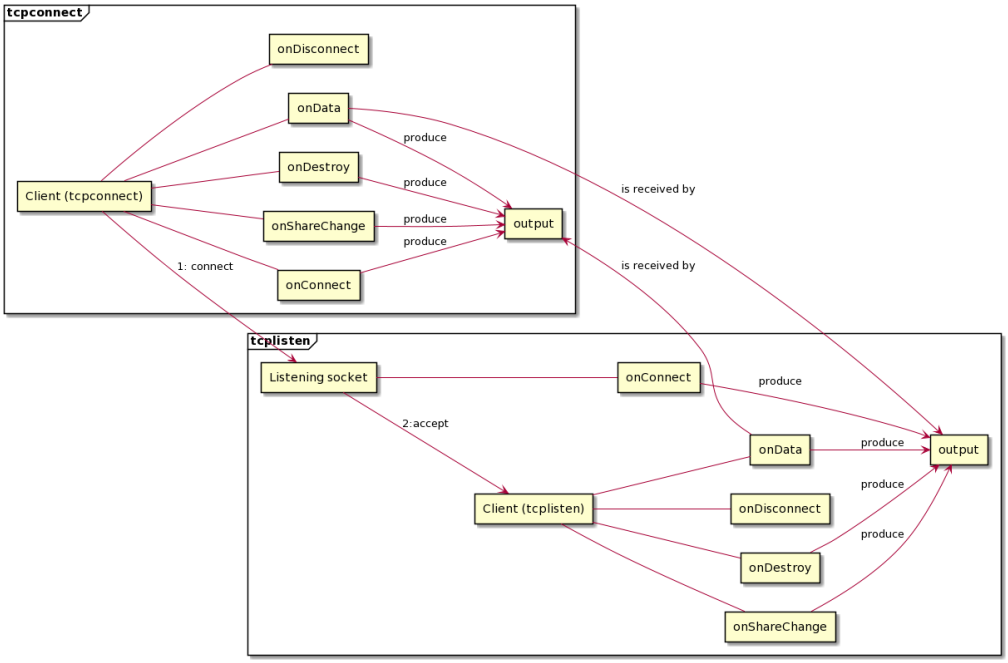


Figure 7.1: How iTasks provides network I/O to end-users.

As stated in section **6.1**, The end-user defines callback functions and provides them to the `tcplisten` and `tcpconnect` tasks. The callback functions are used to react to events occurring (e.g: data being received). With the exception of the `onDisconnect` callback, the callback functions allow to send data back to the peer as a response to a given event occurring. The data which is sent back is provided by the end-user.

7.2 Internal implementation - overview

This section gives a general overview of the internal implementation of end-user network I/O. The following sections proceed by zooming in on this general overview and provide a more detailed description. This is done with the aim of describing some of the technical details that make the replacement implementation work.

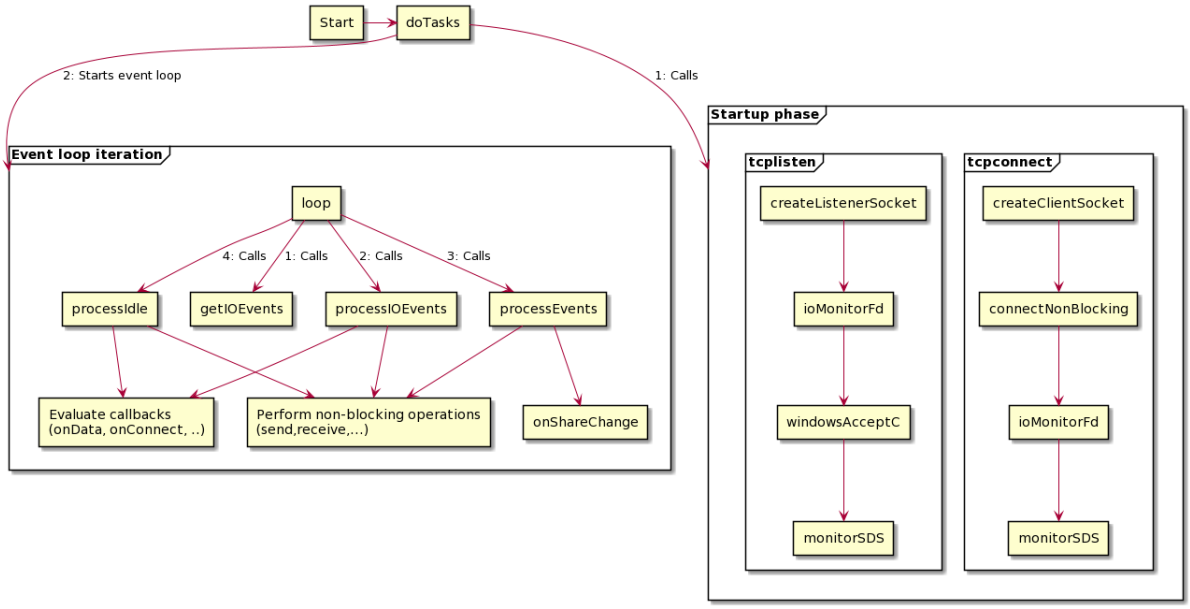


Figure 7.2: Simplified overview of the internal implementation of network I/O in the context of the replacement implementation.

On this level of abstraction, the internal implementations of `tcplisten` and `tcpconnect` are conceptually the same. `tcplisten` creates a listener socket and accepts connections from clients. `tcpconnect` creates a client socket and connects to the listener socket. All of the sockets that are involved in the network I/O communication are monitored by the I/O multiplexing mechanism. Activity on the sockets may result in evaluating callbacks and performing I/O operations. The internal implementation can be split into two phases, a startup phase and a monitor phase. These phases are described separately to make the implementation as a whole easier to understand.

7.3 Startup phase

This section describes the startup phase, the startup phase involves performing the following tasks:

- Creating a socket and either listening for connections (tcplisten) or connecting to the server (tcpconnect).
- Starting the iTasks event loop.
- Monitoring the SDS that is provided to `tcplisten` or `tcpconnect` for changes in order to handle the `onShareChange` callback.

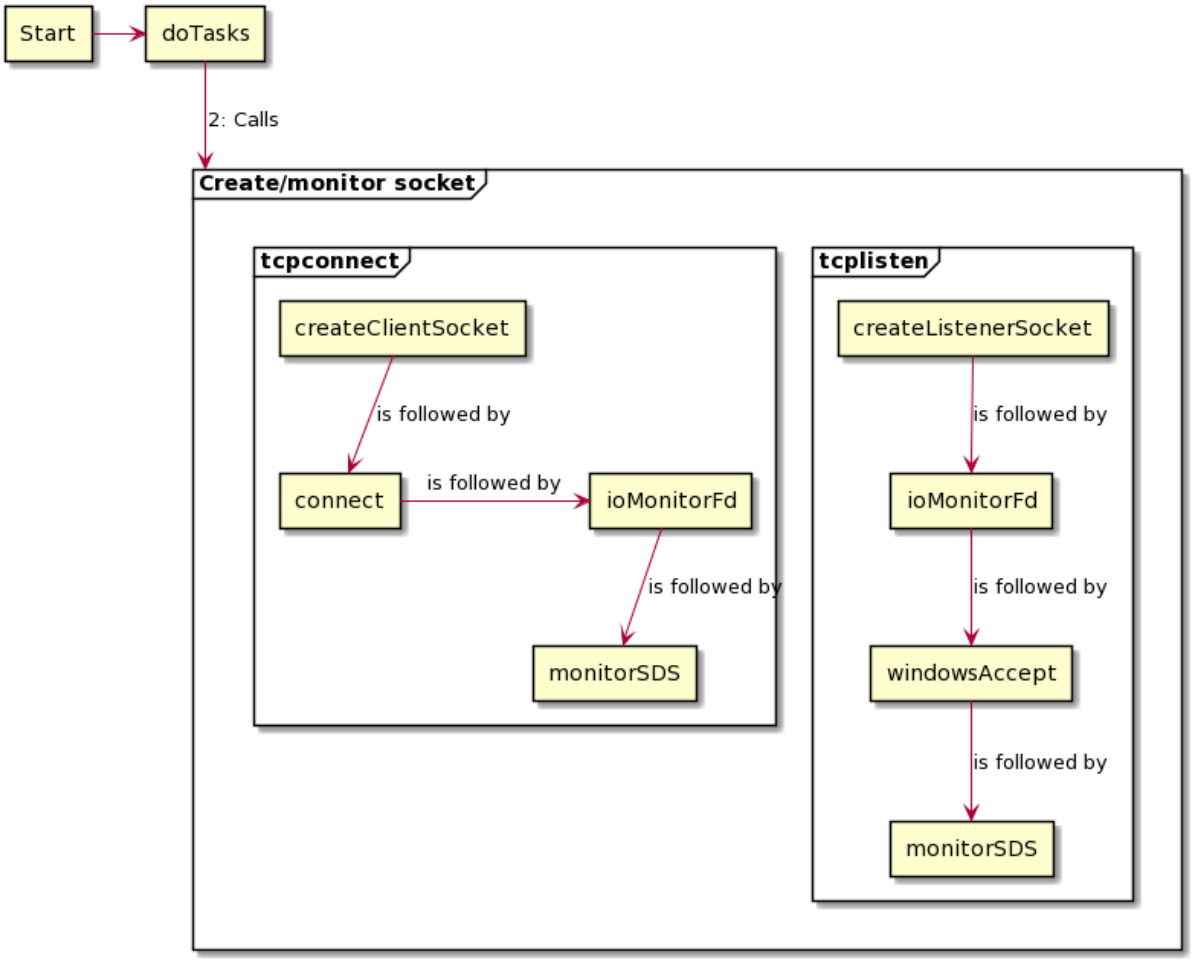


Figure 7.3: Callgraph for the startup phase of the `tcpconnect` and `tcplisten` tasks.

The `tcpconnect` and `tcplisten` tasks are evaluated on startup (see section **6.1**). As a result, the `tcplisten`/`tcpconnect` task are evaluated before starting the event loop. For either task, this leads to a socket being created and monitored.

In the case of `tcplisten`, the created socket is a socket which is used to listen for connections. On Linux/macOS, which use reactive I/O multiplexing mechanisms, this socket is monitored for readability. A listening socket being readable means that a connection requested may be accepted. As a result, the listening socket is monitored for connection requests. In the case of `IOCP` (Windows), an asynchronous `accept` operation is initiated which leads to proactively accepting a connection request from a client. The difference between proactive and reactive I/O multiplexing mechanisms is described in section **3.2**. Either way, when a client successfully connects to the server this leads to an event occurring. The event is processed during the monitor phase.

In the case of `tcpconnect`, the created socket is a client socket. The program attempts to connect to the server – which is specified by the end user – in a non-blocking manner. The file descriptor is then monitored for writability. After the socket becomes writable, it is verified that the connection attempt succeeded. This approach is based on the manual for `connect` [11] [12] (see `EINPROGRESS`). Technically, if the connection attempt succeeded immediately it is not necessary to monitor for writability. The socket will become writable either way so monitoring for writability is always done. This was done to simplify the implementation. The connection attempt succeeding leads to an `onConnect` event being returned to Clean, which is processed during the monitor phase.

Furthermore, the callback SDS which was provided to `tcplisten`/`tcpconnect` is monitored for changes.

When the SDS changes, the `onShareChange` callback which is provided by the end user is evaluated and processed. It should be noted that this approach improves on the approach of the existing implementation. The `onShareChange` in the existing implementation is evaluated and processed once every event loop iteration, which can be considered a bug. In addition, the new approach may save a significant number of SDS reads (1 read per event loop iteration per client if the SDS is never changed). However, it is should be noted that this solution is fragile in the sense that the `onShareChange` callback may still be evaluated more often than is necessary. This is described in more detail in **12**.

When the startup phase ends, the program is in a state where:

- The I/O multiplexing mechanism is initialized.
- The event loop is started.
- In case of using `tcplisten`, a listening socket is created and monitored for connection requests.
- In case of using `tcpconnect`, a client socket is created and an attempt is made to connect to the TCP server in a non-blocking manner.
- Interest has been registered in changes to the callback SDS that was provided by the end-user. This is used to process the `onShareChange` callback in the monitor phase.

The startup phase is followed by the monitor phase.

7.4 Monitor phase

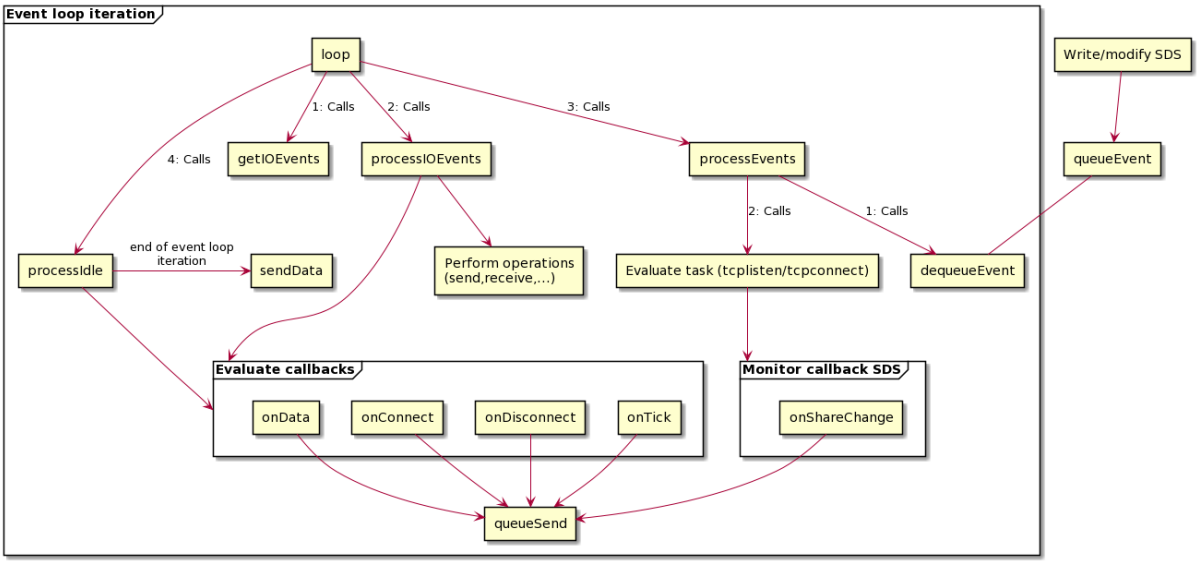


Figure 7.4: Callgraph for the monitor phase of the `tcplisten` and `tcpconnect` tasks.

During the monitor phase, the following tasks are performed:

- The callback SDS is monitored for changes. This is done with the goal of evaluating and processing the `onShareChange` callback when the SDS is written to.
- The I/O Events that occured on the monitored sockets are retrieved and processed. This involves evaluating and processing callbacks and performing I/O operations.
- The idle processing of the monitored file descriptors is done. Idle processing is performed every event loop iteration for every monitored client. Idle processing is independent of the I/O multiplexing mechanism.

The tasks are described separately for clarity.

7.4.1 Monitoring the callback SDS for changes

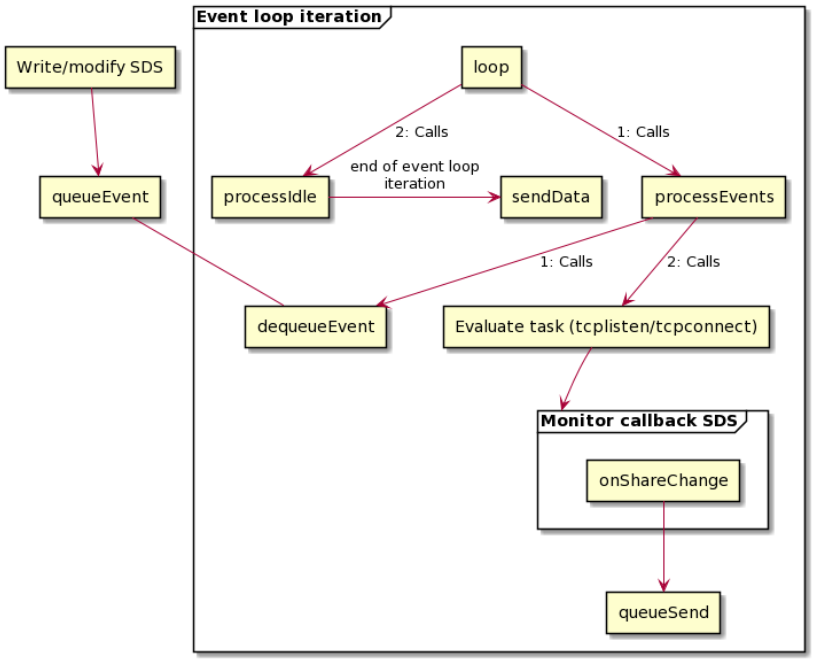


Figure 7.5: Callgraph which illustrates how the callback SDS is monitored for changes.

During the monitor phase, the callback SDS is monitored for changes. As the program is in the monitor phase, the event loop has been started. In the startup phase, interest has been registered in changes to the callback SDS. As a result, whenever the SDS is modified, a task event is queued. During each iteration of the event loop, task events are processed by the `processEvents` function. A task event leads to the task being evaluated. As a result of evaluating the task, the `onShareChange` callback is processed. The `onShareChange` callback may produce output, which is queued for sending. The approach to sending data in the replacement implementation is described in section **3.3**. In conclusion, changes to the callback SDS result in the `onShareChange` callback being processed. Note that if the same value is written to the SDS twice, the `onShareChange` callback will end up being evaluated twice. Changed therefore means "written to" in this context. It is not checked whether the value written to the SDS is actually different because the equality check may be very expensive to perform.

This approach is deemed to be fragile as the task event it relies on is the refresh event. A refresh event may also end up being queued for the task for different reasons. This then leads to evaluating the `onShareChange` even though the SDS did not change.

The existing implementation reads the callback SDS every event loop iteration, regardless of whether the SDS was modified. In conclusion, the `onShareChange` is still evaluated and processed when the SDS is changed. Furthermore, the SDS is no longer read every event loop iteration, which may involve obsolete reads.

7.4.2 Retrieving and processing I/O Events

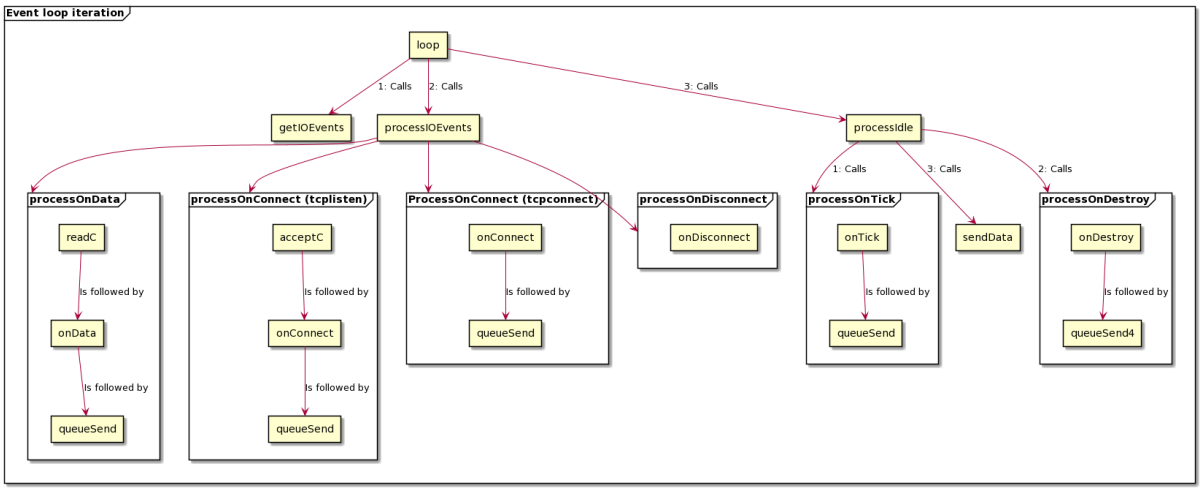


Figure 7.6: Callgraph which illustrates how I/O events are retrieved and processed.

The program starts monitoring for I/O events in the monitor phase. This means that the event loop has been started. The `getIOEvents` function is called every iteration of the event loop. `getIOEvents` is used to retrieve the I/O events that occurred on the file descriptors (sockets) that are being monitored. The `processIOEvents` function is used to process the I/O events that were retrieved by `getIOEvents`. `processIdle` is used to perform the operations that should happen every iteration of the event loop.

The `processIOEvents` function processes the following I/O Events:

- The `onConnect` I/O event in the context of `tcplisten`. In the context of `tcplisten`, the `onConnect` I/O event has a different meaning depending on whether the I/O multiplexing mechanism is reactive or proactive. The difference between reactive and proactive I/O multiplexing mechanisms is described in section **3.2**.

  For `IOCP` (Windows), the `onConnect` I/O event signals that a connection request has successfully been accepted. This is followed by initiating another asynchronous accept operation through `acceptC` which proactively handles the next connection request. Furthermore, data is proactively read on the client socket that resulted from accepting the connection. As a result, the program is able to respond to the peer sending data over the connection. Afterwards, the `onConnect` callback is evaluated and processed. The `onConnect` callback may return output, which is queued to be sent. The process of sending data has been described in section **3.3**.

  For `kqueue` (macOS) and `epoll` (Linux), The `onConnect` event indicates that a connection request may be accepted. The `acceptC` function is used to accept the connection. After this, the `onConnect` callback is evaluated. The output returned by `onConnect` is queued to be sent. The client socket that resulted from accepting the connection is monitored for readability to be able to react to data being sent over the connection.

- The `onConnect` I/O event in the context of `tcpconnect`. This event being returned means that the connection attempt that was performed by `tcpconnect` succeeded. As a result, the `onConnect` callback is evaluated. Furthermore, the socket will be monitored for readability on Linux and macOS. Data is proactively read from the socket on Windows instead (see section **3.2**). The output returned by the `onConnect` callback is queued to be sent.

- The `onData` I/O event, which has a different meaning depending on whether a proactive or reactive I/O multiplexing mechanism is used. The difference between proactive and reactive I/O multiplexing mechanisms is described in section **3.2**.

  In the case of `IOCP` (Windows), which is a proactive mechanism, the `onData` I/O event occurring

confirms that data has been read. When the `onData` I/O event occurs on IOCP, a preceding asynchronous read operation has been completed. This is always the case because the `onData` event is always preceded by an `onConnect` I/O event. This makes sense because a connection must be established before data can be sent/retrieved. Processing the `onConnect` I/O event leads to performing an asynchronous read operation on the connected socket. The completion of an asynchronous read always results in an `onData` I/O event occurring. When the `onData` I/O event occurs, the data from the preceding asynchronous read operation is retrieved. This is followed by initiating another asynchronous read operation, which will lead to another `onData` event. As a result, data is continuously read from the socket because the process repeats itself. There is a small amount of time where there is no outstanding asynchronous read operation. However, this is not a problem because the received data is buffered. If there is no space in the buffer this is handled by the peer. After retrieving the data, the `onData` callback is evaluated and processed. If the `onData` callback returned output, it is queued to be sent.

In practice, retrieving the data from the preceding asynchronous read operation and performing the next read operation has been split into two functions. The reasoning for this is technical. One buffer per file descriptor is used to store the received data for that file descriptor. Clean copies strings that are returned to Clean by a given C function on the heap at the end of the function call. Therefore, when data is retrieved, the data in the receive buffer of the file descriptor is copied to Clean at the end of the function call used to retrieve the data. This is useful because the same buffer can then be reused for the next asynchronous read operation. However, an asynchronous read operation may also complete immediately in some circumstances in the case of using `IOCP` [13] (see the Overlapped Socket I/O section). In this case, the receive buffer is immediately overwritten. Therefore, retrieving the data from the buffer and initiating the next read operation is split. If this were not the case, it would be possible that the the receive buffer would be overwritten by the next read operation if the next read operation completed synchronously. Since Clean only copies the data in the receive buffer to the heap at the end of the function call, this would lead to incorrect data being returned. Splitting the function up makes sure that the data in the receive buffer is retrieved/copied by Clean before the next read operation is initiated.

In the case of `epoll` (Linux) and `kqueue` (macOS), the `onData` I/O event indicates that data may be read. In this case, the `readC` function is used to synchronously read the data in a non-blocking manner. The `readC` function may detect a disconnected peer and this possibility is correctly handled. It is possible to detect a disconnect when reading because the peer may disconnect after the file descriptor becomes readable but before the file descriptor is actually read. If data was successfully read, The `onData` callback is evaluated using the received data and the output returned by the callback is queued to be sent.

The way the `onData` I/O event is processed internally is significantly different depending on whether the used I/O multiplexing mechanism is proactive or reactive. However, the end-user does not need to be aware of the differences. This is a consequence of the `tcplisten` and `tcpconnect` tasks abstracting away from these internal differences.

- The `onDisconnect` I/O event is returned when the peer closes the connection. The event is processed by evaluating the `onDisconnect` callback. The `onDisconnect` callback may not send output as the connection has been closed when it evaluated. At this point, the socket file descriptor is closed and the memory associated with the socket (e.g the receive buffer) is freed. In addition, the `onDisconnect` callback is evaluated when the connection is closed locally.

The `processIdle` function is used to:

- Evaluate and process the `onTick` callback for the connected file descriptors. In the case of `tcpconnect` and `tcplisten`, it is not possible to provide a `onTick` callback. A dummy `onTick` callback is used in the case of using `tcpconnect` and `tcplisten`. However, the `onTick` callback is used by the HTTP server, which is described in section **4**.
- Handle the task being destroyed. In this case, the sockets associated with the task have their `onDestroy` callback evaluated and processed. The `onDestroy` callback may return output, which is queued to be sent. The sockets associated with the task are closed once all data that was queued

for the socket has been sent. Of course, this is done in a best-effort manner. If the peer closes the connection in the meantime it becomes impossible to send the queued data, for example.

- Send the data that has been queued to be sent. IPC and network I/O now share the same approach to sending data. The approach to sending data has been described in section **3.3**.

7.5 Summary

Like the existing implementation, the replacement implementation offers the `tcplisten` and `tcpconnect` tasks for performing network I/O as an end-user. The `tcplisten` task is used to setup a TCP server, listen for incoming connections and communicate with the connected clients. The `tcpconnect` task is used to connect to a TCP server and communicate with it. The `tcplisten` and `tcpconnect` tasks are of the same type as the `tcplisten` and `tcpconnect` tasks that are provided by the existing implementation. As a result, existing programs and extensions that make use of `tcpconnect` and `tcplisten` do not have to be altered.

For the most part, the drawbacks of the existing implementation no longer apply to the replacement implementation. This means that, in the replacement implementation of network I/O:

- The `onShareChange` is no longer evaluated every event loop iteration. Generally, the `onShareChange` callback is only evaluated when the SDS changes but the approach that is taken is fragile (see chapter **12**). This especially increases the scalability of the `tcplisten` task.
- Data is sent and read in a non-blocking manner. As a result, the iTasks program may no longer block when sending or reading data. This is required for horizontally scaling iTasks web applications. Horizontally scaling iTasks web applications is a future goal of the iTasks project.
- Establishing a connection to a TCP server using the `tcpconnect` task is now done in a non-blocking manner. This means that the iTasks program will no longer block when establishing a connection to a TCP server. This is required for being able to horizontally scale iTasks applications.
- The replacement I/O multiplexing mechanisms are used instead of `select`. This allows to provide IPC and network I/O using the same approach.

Chapter 8

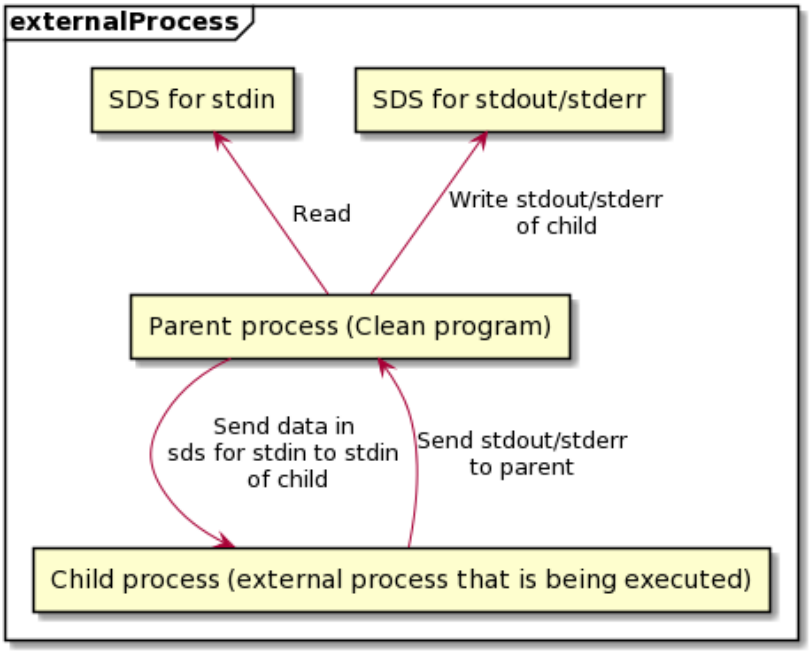
Existing Implementation - IPC

iTasks provides a task that allows to execute an external process. This allows to execute any executable through an iTasks program. Such executables may produce output data and process input data.

For example, consider the `su` command. The `su` command allows a user to perform commands with root (superuser) privileges. The location of the executable for the `su` command is normally located in `/bin/su` on Linux systems. If the `su` command/executable is used without parameters, it takes input (the password of the superuser account) and may return output. For instance, if the authentication attempt failed produces output indicating that the authentication failed.

An iTasks program may be interested in providing input to executables that are executed as an external process. Similarly, iTasks programs may also be able to read output from the program that is executed as an external process. This involves inter-process communication. This chapter discusses how the existing implementation provides inter-process communication.

The task that allows the end-user to execute an external process and communicate with it is called `externalProcess`. An overview of how the `externalProcess` allows the end-user to execute external processes and interact with these processes is included below.



When the `externalProcess` task is executed, the currently running Clean process is duplicated using `fork` to create a new Clean process. After this, `exec` is used to execute the external process the user

wishes to run on this new process, the duplicated Clean process is thereby replaced. The new process, on which the external process is executed, is called the child process since it was created by the parent process. The standard input channel of the external process is called stdin. Likewise, the standard output and standard error output channels of the external process are called stdout and stderr. The data written to the stdout/stderr channels of the child process is sent to the parent process. The parent process then writes this data to the SDS (Shared Data Source) that stores the stdout/stderr data. SDSs provide a means to read and write shared data. A SDS is used to abstract from various means for storing data, such as a file, shared memory or a database. The end-user may then read the data in this SDS and process it.

The end-user may write data to the SDS for stdin. This SDS is then read by the parent process. The data that is stored within the SDS for stdin is written to the stdin of the child process. This is followed by the contents of the SDS for stdin being emptied.

Communication between the child and parent process is provided either through pipes or through a pseudoterminal, depending on the choice of the end-user. Pseudoterminals are only supported on the Linux/macOS platforms.

The `externalProcess` task has the following type:

```
externalProcess :: Timespec FilePath [String] (?FilePath) Int
                   (?ProcessPtyOptions) (Shared sds1 [String])
                   (Shared sds2 ([String],[String])) -> Task Int
                   | RWShared sds1 & RWShared sds2

externalProcess pollInterval executablePath parameters mbStartDir exitcode
                mbPtyOptions stdinsds stdouterrsds = ...
```

`externalProcess` takes the following arguments:

- A poll interval, which determines how often data is read from and written to the external process.
- The path of the executable of the process to be executed.
- The command line arguments (parameters) that should be provided to the executable.
- The starting directory (useful if providing a relative path for `executablePath`)
- The exitcode to send when the task is destroyed.
- Possibly pseudoterminal options. If this is `?None`, pipes are used to provide the inter-process communication. If this is `?Just` then a pseudoterminal will be used to provide inter-process communication. Pseudoterminals are not supported on Windows. The choice of using a pseudoterminal or pipes needs to be accounted for within the internal implementation. It also has impact on the end-user. For example, it is not possible to execute certain processes externally using pipes. For example, the `su` command may only be communicated with through a (pseudo)terminal. Pipes allow to separate the stdout and stderr output of the external process while the pseudoterminal does not.

  When using a pty, the stdout and stderr output of the external process is not properly separated. This is considered a bug and it is described in more detail in section **8.1**. Furthermore, stdout is fully buffered when using pipes. In the case of using a pseudoterminal, stdout is line-buffered [21]. This can lead to different behavior when executing external processes. As a process terminates the stdout buffer is automatically flushed. In the case of using pipes, it is the responsibility of the external process to flush the stdout/stderr buffer to send data to the parent process. Otherwise, the data that is sent to stdout/stderr by the external process might not arrive to the parent process.
- A shared data source (SDS) in which the input to be provided to the external process is stored. This shared data source is read once every `pollInterval`. If there is data, it is sent to the stdin of the process. The data that was sent is removed from the SDS afterwards.

- A shared data source in which the output of the external process is stored. The task attempts to read data from the external process once every `pollInterval`.

Below, the use of the `externalProcess` task is explained according to a simple example. The example involves a Clean/iTasks program which executes an external process and provides it with input. Afterwards, it receives output from the external process. The behavior of the external process is specified through a C program.

The C program included below is the external process that is executed. It reads two integers from stdin and writes the sum to stdout. Note that any kind of executable may be executed as an external process.

```
// addtwonumbers.c: Reads two numbers from stdin and prints the sum to
    stdout.
int main() {
    int a, b;
    scanf("%d", &a);
    scanf("%d", &b);
    printf("%d\n", a + b);
    return 0;
}
```

The program included below executes the C program specified above and writes two integers to the stdin of the external process. It reads the sum of the integers that is written to stdout by the external process. The sum is then written to the SDS used for storing stdout and stderr.

```
module process

import iTasks
import System.Time
import Text
import StdDebug

Start w = doTasks (onStartup process) w

process :: Task Int
process = withShared ["2\n3\n"] \input ->
          withShared ([], []) \outputStorage ->
          externalProcess {Timespec|tv_sec=1, tv_nsec=0}
                          "/path/to/addtwonumbers.exe" [] ?None 0
                          ?None input outputStorage
         >>- \exitcode ->
          get outputStorage >>~ \(stdout,stderr) ->
          trace_n (concat stdout) return exitcode

externalProcess :: Timespec FilePath [String] (? FilePath) Int
                   (? ProcessPtyOptions) (Shared sds1 [String])
                   (Shared sds2 ([String],[String])) -> Task Int
                   | RWShared sds1 & RWShared sds2
externalProcess pollInterval executablePath parameters mbStartDir exitcode
                mbPtyOptions stdinsds stdouterrsds = ...
```

The process task involves executing the C program included above, providing two numbers as input. An enter keypress is encoded as a newline character. In this example program, pipes are used as a means of communication with the external process. This is a result of the pseudoterminal options argument being `?None`. The input numbers are provided to the stdin of the external process. The external process adds the numbers and prints the sum of the two numbers to stdout. This output is read from the pipe and written to the stdout/stderr SDS. When the program terminates, the output that is stored in the stdout/stderr SDS is retrieved and shown to the user.

8.1 Internal implementation

This section describes the internal implementation of the `externalProcess` task. This description is based on the example program included above.

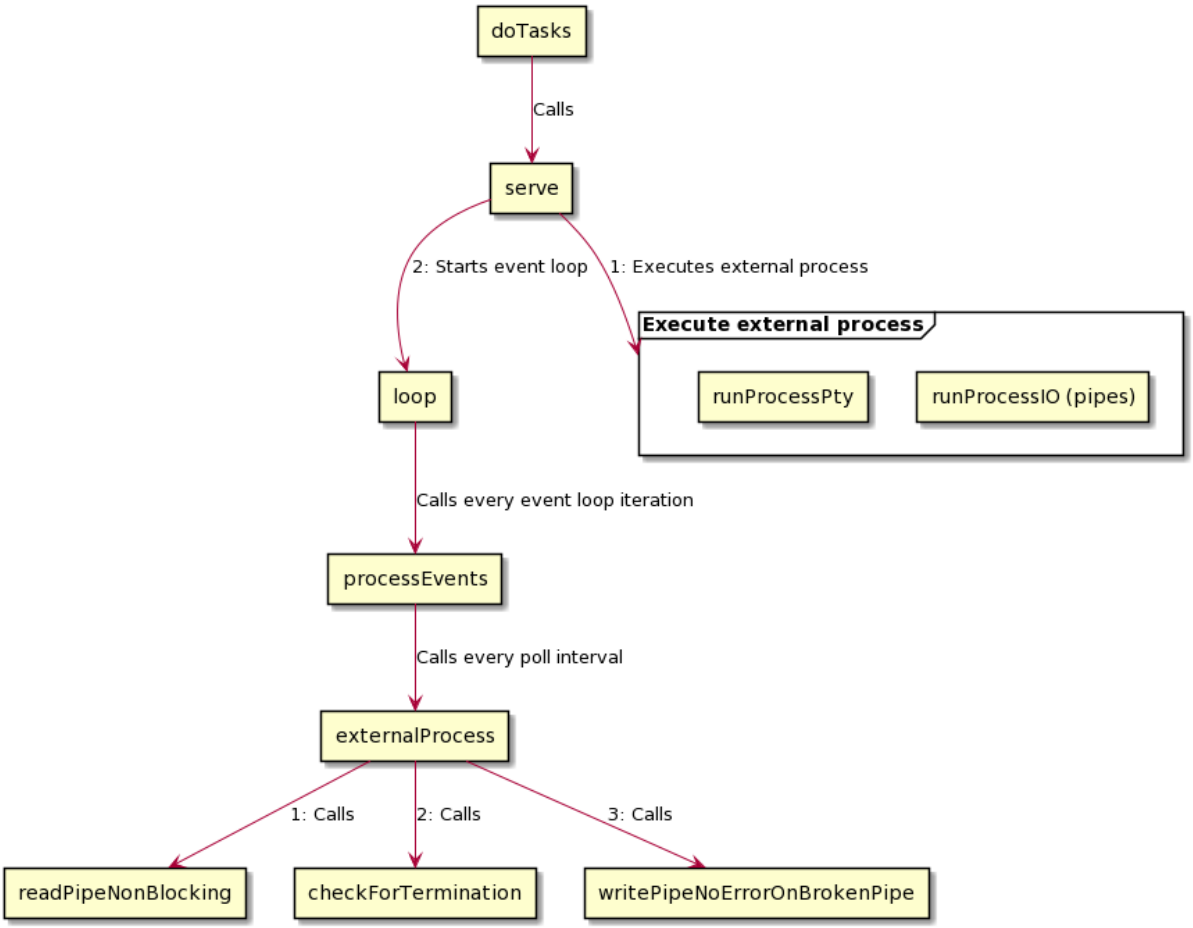


Figure 8.1: Overview of the internal implementation of `externalProcess`.

The internal implementation of the example program included above can be summarized as follows:

1. The `doTasks` function specified in `Start` ends up evaluating the `serve` function. The `Start` function is the entry point of any iTasks/Clean program.
2. The serve function executes the external process, this process is different depending on whether pipes or a pty is used as a means of communication. The `runProcessPty` function executes the external process if the end-user specified to use a pty. The `runProcessIO` is used if the end-user specified to use pipes. The input and output channels of the external process are redirected to pipes/a pty. The `externalProcess` task has access to the pipes/pty.
3. After executing the external process, the serve functions starts the event loop.
4. The loop function evaluates the `processEvents` function every event loop iteration.
5. The `processEvents` function evaluates the `externalProcess` task every poll interval.
6. The `externalProcess` task reads the output printed by the external process through `readPipeNonBlocking`. If there was any output, it is written to the SDS which is used to the store the output. When using a pty, the data written to stdout and stderr is not properly separated. This is considered to be a bug. The data may end up in stdout/stderr based on timing. The internal implementation

performs two non-blocking reads in succession on the same file descriptor when using a pty. The data from the first non blocking read is written to the stdout part of the SDS. The data from the second non blocking read is written to the stderr part of the SDS. In case of using pipes, stdout and stderr are redirected to two different file descriptors so the problem does not occur in this case.

7. Afterwards, it is checked whether the external process terminated. If the external process terminated, the task returns the exitcode of the external process. In this case, the `externalProcess` task is finished and the `externalProcess` task is no longer periodically evaluated.
8. The `externalProcess` task proceeds by reading the input which is stored in the stdin SDS. The contents of the SDS is written to the stdin of the external process using `writePipeNoErrorOnBrokenPipe`. Afterwards, the input stored in the SDS is cleared as it has been sent. The data is written in a blocking manner. A goal for the replacement implementation is to implement all I/O operations in a non-blocking manner.

8.2 Summary

To summarize, iTasks allows to execute an external process. An external process can be seen as any executable computer program. iTasks allows to provide input to the external process and process the output that is produced by the external process. Providing input to the external process and reading the output of the external process happens through IPC (inter-process communication). The existing IPC implementation relies on a time-based polling mechanism. The output of the external process is periodically read. Likewise, input to be provided to the external process is periodically sent. This is a different approach than the I/O multiplexing approach taken by the network I/O implementation. As a result of taking this approach, the program attempts to read and send data even though it may not be possible to read or write data.

As a result of examining the existing implementation, the following points of improvement were identified:

- IPC should be provided through a concept that is based on I/O multiplexing as well. This results in providing IPC and network I/O through the same concept. This increases the consistency and maintainability of the implementation and avoids attempting to read or send data when it is not possible to do so.
- Pseudoterminals do not allow to separate stderr and stdout output. The output SDS of the `externalProcess` task has a separate storage for stdout and stderr data. Currently, the output of the external process is written to the stdout and stderr storage based on timing when using a pseudoterminal. It was decided that it would make more sense to store all of the output data in the stdout storage when using a pseudoterminal. In the existing implementation, the output will most likely all end up in the stdout storage but this is not guaranteed. This is considered to be a problem.
- Writing the input to the external process is done in a blocking manner. A general goal of the thesis is to implement all I/O operations in a non-blocking manner.
- The network I/O tasks make use of a callback record for sending input and receiving output. The `externalProcess` task makes use of SDSs. It was decided to add an IPC task that makes use of a callback record as well so that all the I/O tasks make use of a consistent approach to sending and receiving data. The current `externalProcess` task should be re-implemented as well for backwards compatibility.

Chapter 9

Replacement Implementation - IPC

iTasks provides a means to execute an external process. An external process can be seen as any kind of executable computer program. External processes have I/O channels (stdin, stdout and stderr). iTasks allows to interact with these channels through IPC (inter-process communication). This chapter discusses how IPC is provided in the replacement implementation. The preceding chapter describes how IPC is provided in the existing implementation. A point of improvement of the existing implementation is that IPC and network I/O handle interaction through a different approach. The network I/O functionality makes use of callback records while the IPC functionality makes use of SDSs (Shared Data Sources). The replacement implementation introduces a new task called `externalProcessHandlers`. This task provides IPC through a callback record as well. This chapter first introduces how IPC is provided to the end-user in the replacement implementation. This includes an introduction of the `externalProcessHandlers` task. This is followed by an explanation of the internal implementation of IPC in the replacement implementation.

9.1 Providing IPC to the end-user

In the replacement implementation, IPC is provided to the end-users through the `externalProcess` and `externalProcessHandlers` tasks. These tasks allow executing and communicating with an external process through an iTasks program. The end-user may provide data to the stdin of the external process through a pipe/pty. The stdout and stderr output of the external process is redirected to pipes or a pseudoterminal. As a consequence, whenever an external process writes to stdout, this output is redirected to the pipe/pseudoterminal. The iTasks program may then read the pipes or pseudoterminal to read the stdout and stderr output of the external process. This allows the iTasks program to provide the output of the external process to the end-user. To summarize the above, it is possible for an iTasks program to communicate with an external process. The figure below provides an overview of this communication. The parent process is the iTasks program which is being executed. The child process is the external process. The external process (child process) is created by the iTasks program (parent process).

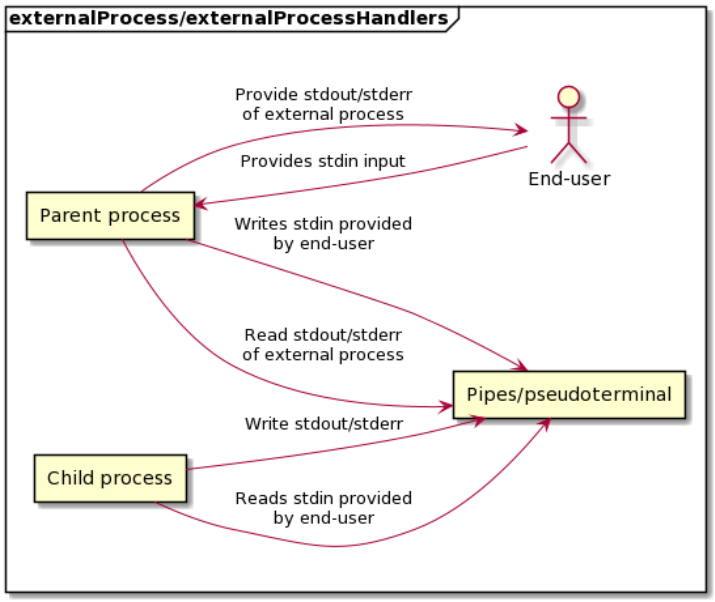


Figure 9.1: Communication between an external process and the iTasks program.

When the `externalProcessHandlers` or `externalProcess` task is evaluated, the currently running Clean process is duplicated using `fork`. The `fork` function duplicates the currently running Clean process to create a child process. This is necessary to obtain a process on which the external process may be executed. The Clean process which is used to create the child process is called the parent process. The parent process continues execution of the Clean/iTasks program. The child process is used to execute the external process.

This means that the executable which the end-user wishes to execute (for example, `/bin/ls`) is executed on the child process. The child process is replaced for this external process using the `exec` function. `exec` leads to the duplicate Clean process being replaced for the external process itself. The end-user provides may provide input to the external process through the parent process. Likewise, the end-user may process output from the external process through the parent process. The parent process and the child process (the external process) communicate through pipes/a pseudoterminal.

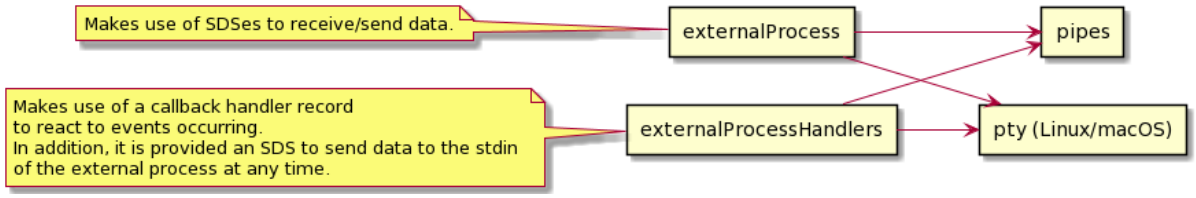
Compared to the existing implementation, the replacement implementation may provide IPC to the end-user in a different manner. `externalProcess` is the task that is used to provide IPC in the existing implementation. Within the replacement implementation, this task has been split into two separate tasks. Namely, `externalProcess` and `externalProcessHandlers`.

The `externalProcess` task provides IPC to the end-users in the same way as the existing implementation. Chapter **8** describes how this task provides IPC to the end-user. Nonetheless, the internal implementation of the `externalProcess` task has been re-implemented such that it makes use of the I/O multiplexing concept instead of the time-based solution that is used by the existing implementation. Furthermore, the I/O operations that are performed by `externalProcess` are implemented in a non-blocking manner.

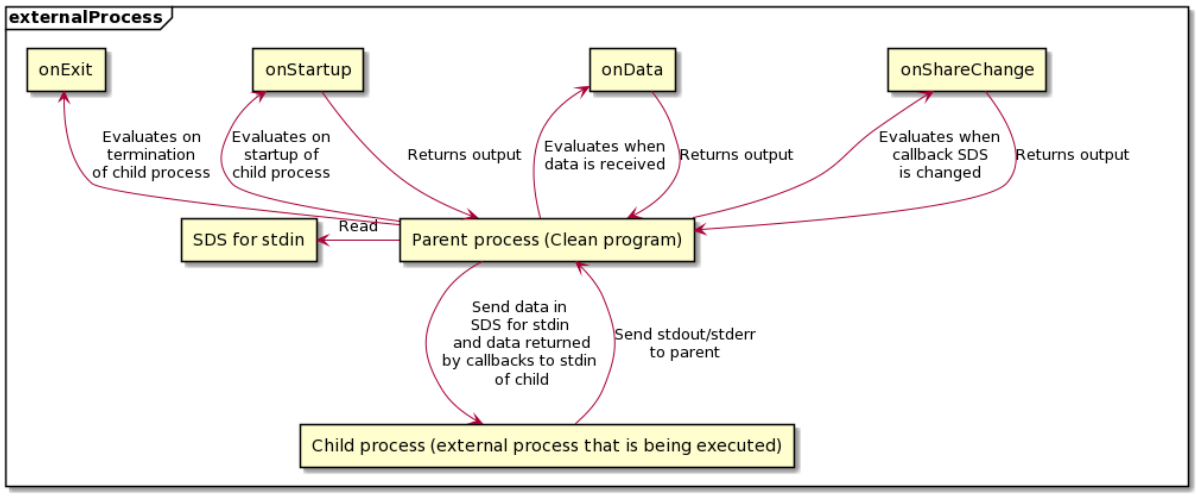
`externalProcessHandlers` has a different type and provides IPC to the end-user in a different way than `externalProcess`. It involves a callback record which is similar to the callback record provided to `tcpconnect` and `tcplisten`. These are the tasks which provide network I/O to the end-users.

The reasoning for adding the `externalProcessHandlers` task is to make the I/O Tasks (`tcplisten`, `tcpconnect` and `externalProcessHandlers`) more uniform. The `externalProcess` task is provided for backwards compatibility.

The overview included below illustrates the options the end-user has when using `externalProcess` and `externalProcessHandlers`.



The `externalProcessHandlers` task provides a different way of interacting with an external process. The overview included below illustrates how `externalProcessHandlers` provides this interaction.



`externalProcessHandlers` is provided a SDS for sending data to the stdin of the external process at any point in time. This SDS is read by the parent process and the data contained within the SDS is sent to the stdin of the child process. The data written to stdout/stderr by the external process is sent over to the parent process. This is done by using either pipes or a pseudoterminal as a means of communication. `externalProcessHandlers` is provided a callback record which is used to react to events occurring when interacting with the external process. With the exception of the `onExit` callback, the callbacks may produce output which is then sent to the stdin of the child process. This allows a Clean program to communicate with an external process.

The `externalProcessHandlers` task has the following type:

```
externalProcessHandlers :: FilePath [String]
                           (? FilePath) Int (? ProcessPtyOptions)
                           (Shared sdsin [String]) (sdshandlers () r w)
                           (ExternalProcesshandlers l r w) -> Task Int
                           | RWShared sdsin & RWShared sdshandlers
                           | iTask l & iTask r & iTask w

externalProcessHandlers executablePath parameters mbStartDir exitcode
                        mbPtyOptions stdinsds sdshandlers handlers = ...
```

`externalProcessHandlers` takes the following arguments:

- The path of the executable of the process to be executed.
- The command line arguments (parameters) that should be provided to the executable.
- The starting directory (useful if providing a relative path for `executablePath`)
- The exitcode to send when the task is destroyed.

- Possibly pseudoterminal options. If this is `?None`, pipes are used to provide the inter-process communication. If this is `?Just` then a pseudoterminal will be used to provide inter-process communication. Pseudoterminals are not supported on Windows. The choice of using a pseudoterminal or pipes needs to be accounted for within the internal implementation. It also has impact on the end-user. For example, it is not possible to execute certain processes externally using pipes. For example, the `su` command may only be executed through a (pseudo)terminal. Pipes allow to separate the stdout and stderr output of the external process while the pseudoterminal does not.
- A shared data source (SDS) in which the input to be provided to the external process is stored.
- A SDS which is used to provide custom arguments to the callback functions.
- A callback record that allows the user to define what should happen which certain events occur (e.g: data is received from the external process, the external process exists, ...). The callback record has the following type:

```
:: ExternalProcessHandlers l r w =
{ onStartup      :: (           r -> (?w, [Output], Close))
, onOutData      :: (Data       r -> (?w, [Output], Close))
, onErrData      :: (Data       r -> (?w, [Output], Close))
, onShareChange  :: (           r -> (?w, [Output], Close))
, onExit         :: (ExitCode   r -> (?w                  ))
}
:: ExitCode = ExitCode Int
:: Output :== String
:: Data :== String
:: Close :== Bool
```

The `r` and `w` type variables correspond to the read and write types of the callback SDS that is provided to the
`externalProcessHandlers` task. All callbacks are thus provided the read value of the SDS to access its contents and optionally return a write value to write back to the SDS.

An example of the use of the `externalProcessHandlers` task is given below. A simple C program is executed to serve as an external process. The C program reads two integers from stdin and prints the sum to stdout. The C program is included below.

```
// addtwonumbers.c: Reads two numbers from stdin and prints the sum to
    stdout.
int main() {
    int a, b;
    scanf("%d", &a);
    scanf("%d", &b);
    printf("%d\n", a + b);
    return 0;
}
```

An iTasks program which executes the C program above is included below. It provides two numbers as stdin to the external process. The external process then writes the sum of the numbers to stdout. The stdout of the external process is redirected to a pipe. The iTasks program reads from this pipe to obtain the sum and prints it to the console. This is done by using the `trace_n` function in the provided `onOutData` callback.

```
module process

import iTasks
import StdDebug
import System.Process
```

```
Start w = doTasks (onStartup process) w

handlersOutStartup :: ExternalProcessHandlers () () ()
handlersOutStartup =
        {ExternalProcessHandlers|
          onStartup     = \r          -> (?None, ["2\n3\n"], False)
        , onOutData     = \sum r      -> trace_n sum (?None, [], False)
        , onErrData     = \data r     -> (?None, [], False)
        , onShareChange = \r          -> (?None, [], False)
        , onExit        = \exitcode r -> ?None
        }

sdshandlers :: SDSSource () () ()
sdshandlers = nullShare

process :: Task Int
process = withShared [] \sdsin ->
          externalProcessHandlers "/path/to/addtwonumbers" [] ?None 0 ?None
                                  sdsin sdshandlers handlersOutStartup

externalProcessHandlers :: FilePath [String]
                           (? FilePath) Int (? ProcessPtyOptions)
                           (Shared sdsin [String]) (sdshandlers () r w)
                           (ExternalProcesshandlers l r w) -> Task Int
                           | RWShared sdsin & RWShared sdshandlers
                           | iTask l & iTask r & iTask w
externalProcessHandlers executablePath parameters mbStartDir exitcode
                        mbPtyOptions stdinsds sdshandlers handlers = ...
```

The program included above makes use of the `externalProcessHandlers` task to execute the addtwonumbers program. The `Start` function makes sure the `process` task is executed on startup. The `process` tasks creates a SDS in which input for the stdin of the externalProcess may be stored and evaluates `externalProcessHandlers`.

The numbers that the external process tries to read from stdin using `scanf` are sent once the external process has been started, this is defined through the `handlersOutStartup` callback record. This program makes use of pipes instead of the pseudoterminal to communicate with the external process. This is the case because `?None` is provided as the argument for `?ProcessPtyOptions`. Note that the end-user may have also provided the numbers to the share for the stdin like so:

```
process :: Task Int
process = withShared ["2\n3\n"] \sdsin ->
          externalProcessHandlers "/path/to/addtwonumbers" [] ?None 0 ?None
                                  sdsin sdshandlers handlersNoOutput
```

In this case, the onStartup callback that is defined by the end-user would not have to return output.

9.2 Internal implementation - Overview

The internal implementations of the `externalProcess` and `externalProcessHandlers` tasks are quite similar. Therefore, it suffices to use a single overview. The differences between `externalProcess` and `externalProcessHandlers` are made explicit as the overview is explained. An overview of the internal implementations of the `externalProcess` and `externalProcessHandlers` tasks is included below.

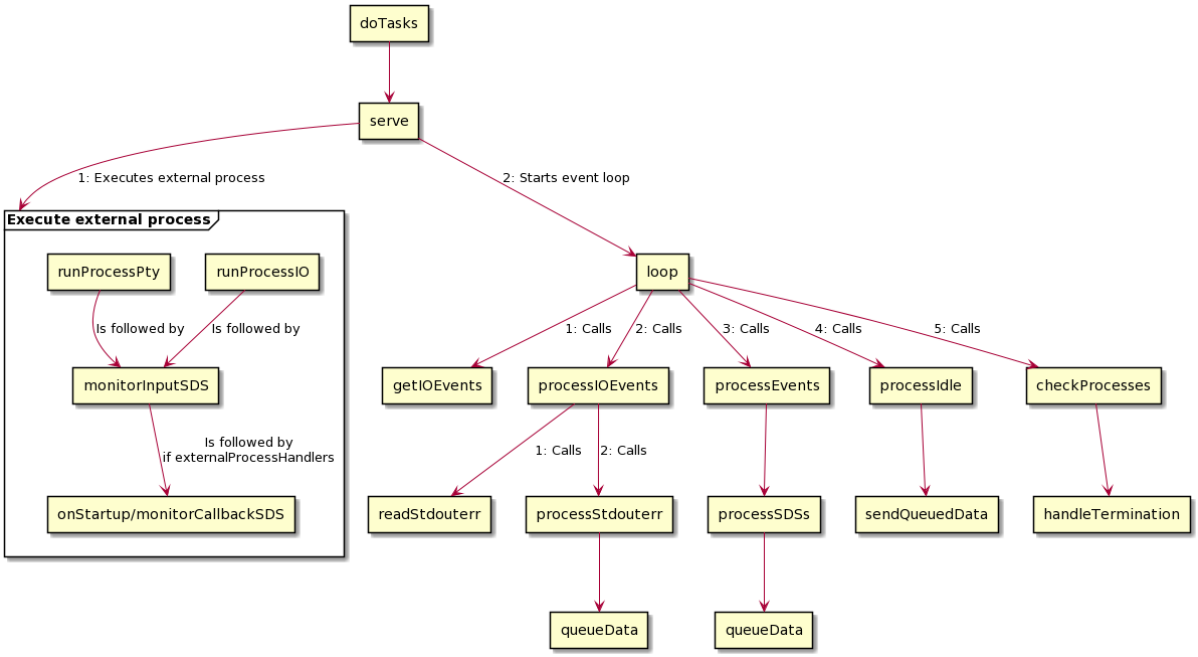


Figure 9.2: Complete overview of the internal implementation of the `externalProcess` and `externalProcessHandlers` tasks.

The callgraph above provides a somewhat daunting overview of the internal implementation of IPC. However, the callgraph can be divided into two phases which both perform various tasks. The phases are named the startup phase and the monitor phase. Individually, the tasks that are performed by both phases are relatively small and understandable. Combined together, they perform the relatively complicated task of providing IPC to the end-user. The phases are described separately for clarity. During the startup phase, the external process is started and the file descriptors that are used to communicate with the external process are monitored. Furthermore, the SDSs involved with the task are monitored. During the monitor phase, the external process is monitored for events and the events that occur for the external process are processed.

9.3 Internal implementation - Startup phase

The first phase is named the startup phase. The startup phase aims at performing several tasks sequentially:

1. Creating pipes/a pty such that the stdin/stdout and stderr of the external process may be redirected to the created pipes/pty.

2. Monitoring the created pipes/pty using the I/O multiplexing mechanism.
3. Redirecting the stdout/stderr/stdin of the external process to the monitored pipes/pty.
4. Executing the external process.
5. Registering interest in changes to the SDS that is used to store data that is to be sent to stdin.
6. When using the `externalProcessHandlers` task, registering interest in changes to the callback SDS. The `externalProcessHandlers` task suffers from the same problem as the network I/O tasks in regards to monitoring this SDS. This means that the `onShareChange` callback may be evaluated more often than is necessary (see chapter **12**).

The callgraph for the startup phase is included below, note that this call graph is a section of the complete overview included above.

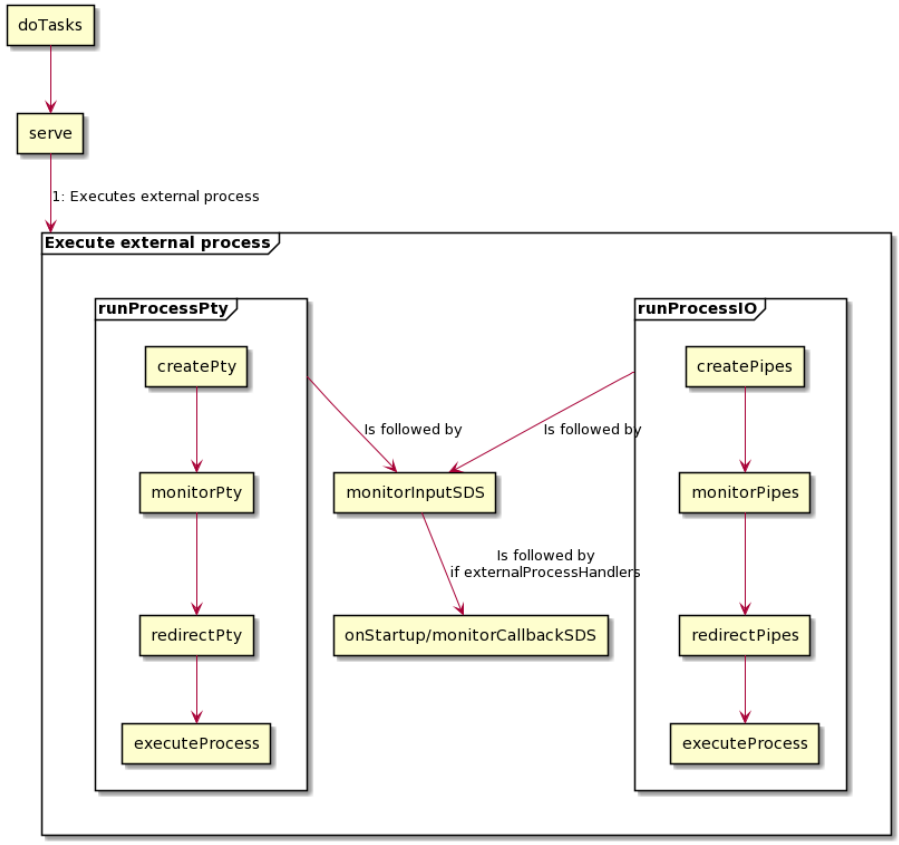


Figure 9.3: Overview of the startup phase of the internal implementation

The following steps are performed during the startup phase:

1. The `doTasks` function calls the `serve` function. The `serve` function leads to starting the external process. This process has been simplified in the above overview. `runProcessIO` is used to start and monitor the external process when pipes are used as a communication channel. `runProcessPty` is used to start and monitor the external process when a pty (pseudoterminal) is used as a communication channel.
2. `runProcessIO` first creates the pipes that will be used to communicate with the external process. Likewise, `runProcessPty` first creates a pty to communicate with the external process. The pipes/pty are then monitored by the I/O multiplexing mechanism. Afterwards, `runProcessIO` and `runProcessPty` duplicate the currently running Clean process (the parent process) using `fork` to create a new process. The newly created process, on which the external process will be executed,

is called the child process. After this, the stdout/stderr/stdin channels of the child process are redirected to the created pipes/pty. This is followed by the duplicate child process being replaced for the external process using `exec`. As a result, the stdout/stderr/stdin of the external process are redirected to pipes or a pty. The parent process has access to the created pipes/pty. The pipes/pty may then be used to communicate with the external process. Furthermore, the pipes/pty used to communicate with the external process are monitored by the I/O multiplexing mechanism.

Note that using a pseudoterminal is not supported on Windows. When pipes are used; stdin, stdout and stderr are redirected to separate file descriptors (pipes). When a pty is used; stdin, stdout and stderr are redirected to a single file descriptor (pty). This is also the case in the existing implementation.

When using a pseudoterminal, data written to both stdout and stderr is treated as if it were all stdout. This is a result of a single file descriptor being used. It is not possible to separate stdout and stderr data when using a pseudoterminal. This is inherent to using pseudoterminals as a means of communication.

This characteristic of pseudoterminals is dealt with differently in the existing implementation. In this case output may be written to the stdout SDS or stderr SDS based on timing. The existing implementation performs two non-blocking reads on the file descriptor in succession. The data from the first read is written to the stdout SDS. The data from the second read is written to the stderr SDS. This means that usually all data will end up in the stdout SDS. However, it is possible that some might end up in the stderr SDS. This can happen regardless of whether it is stdout or stderr data. This is a result of stdout/stderr not being distinguishable when using a pseudoterminal. The approach taken by the replacement implementation is preferable as it is always clear where the data ends up. When using pipes, the existing implementation does separate stderr/stdout data.

3. The SDS that is used for providing stdin to the external process is monitored for changes.
4. The `onStartup` callback is evaluated in case the `externalProcessHandlers` task is used.
5. The callback SDS is monitored for changes in case of the `externalProcessHandlers` task is used.

Upon completion of the startup phase, the monitor phase commences. This is done by the `loop` function being evaluated, starting the event loop.

9.4 Internal Implementation - Monitor phase

The second phase is named the monitor phase. During the monitor phase, the following tasks are performed:

- The stdout and stderr of the external process are monitored for output. This is a result of monitoring the pty/pipes that are used to communicate with the external process using the I/O multiplexing mechanism.

  When using `externalProcess`, the stderr and stdout output of the external process is written to a SDS. This SDS is provided to the `externalProcess` task by the end-user. When using a pseudoterminal, all output is written to the stdout storage of the SDS.

  If `externalProcessHandlers` is used, the stderr and stdout output is provided to the `onErrData` and `onOutData` callbacks. These callbacks are evaluated when output is received. When using pseudoterminals, all output is processed through the `onOutData` callback. This is a consequence of not being able to process stderr and stdout separately if a pseudoterminal is used.

- The SDS for stdin provided to the `externalProcess` task is monitored for changes. If the SDS changes, the data it contains is retrieved. Afterwards, the data it contains is written to the stdin of the external process using a pipe/pty. The data stored within the SDS is then cleared.
- When using the `externalProcessHandlers` task, the callback SDS is monitored for changes. If the callback SDS changes, the `onShareChange` callback is evaluated and the return values are processed.

- The external process is monitored for termination. If the external process terminated, the task returns a stable value.

The monitor phase ends if the external process terminates or a callback returns that the external process should be terminated (externalProcessHandlers).

An overview of the monitor phase is included below:

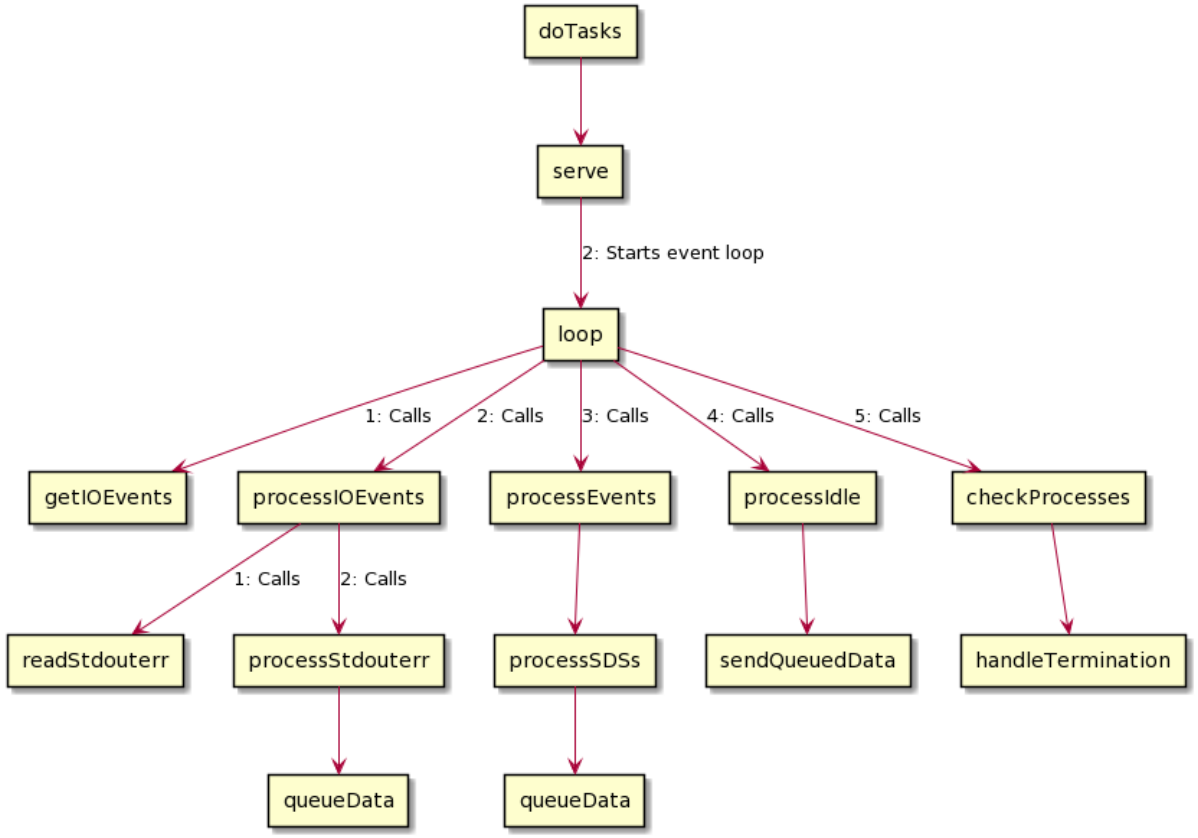


Figure 9.4: Overview of the monitor phase of the internal implementation of the `externalProcess` task
.

Notice how the implementation of IPC is very similar to the implementation of the HTTP server and network I/O in the replacement implementation. This is a result of IPC now being provided to the I/O multiplexing concept as well. On a lower level than is shown by the overview, this allows to reuse a significant amount of code between the network I/O and IPC implementation. For example, Linux and macOS provide a generic way to send and receive data to file descriptors (pipes and sockets). As a result, sending and receiving data has a single implementation on these platforms, regardless of whether a pipe or socket is used. Windows unfortunately requires using different functions, but the approach to reading/sending data to a pipe or socket is the same as well for Windows. In the existing implementation, this is not the case.

In the monitor phase, the startup phase has finished. As a result, the program is in a state where

- The event loop has been started.
- The external process has been started.
- The stdout/stderr/stdin channels of the external process have been redirected to pipes/a pty. The I/O multiplexing mechanism monitors the pipes/the pty.
- The SDS for stdin is monitored for changes.

- If the `externalProcessHandlers` task is used, the callback SDS that is provided to `externalProcessHandlers` is monitored for changes.

As stated, several tasks are performed during the monitor phase:

- The stdout and stderr of the external process are monitored for output. This is done using the I/O multiplexing mechanism. The output is then processed depending on whether the `externalProcessHandlers` or `externalProcess` task is used. In either case, the end-user can retrieve the data sent to stdout/stderr by the external process.
- The SDS for stdin provided to the `externalProcess` or `externalProcessHandlers` task is monitored for changes. Whenever the SDS changes, the data content is read and sent to the stdin of the external process. The SDS for stdin is then emptied.
- if the `externalProcessHandlers` task is used, the callback SDS that is provided to the `externalProcessHandlers` task is monitored for changes. If the SDS changes, the `onShareChange` callback is evaluated and processed.
- The external process is monitored for termination. If the external process terminated, the task returns a stable value. This can be useful when the end-user wants to use a sequential iTasks combinator to react to the external process terminating. For example, the end-user could specify a program such as:

```
process = withShared [] \stdin ->
          withShared [] \stdouterr ->
          externalProcess ... stdin stdouterr >>- \exitcode ->
          get stdouterr >>~ \(stdout, stderr) -> ...
```

  In this case the >>- combinator is used to react to the `externalProcess` returning a stable value. This only happens when the externalProcess terminates. In this example, the stdout and stderr of the external process is read after the external process terminates.

The internal implementation of each of these tasks is discussed separately for clarity.

9.4.1 Monitoring the SDSs

When `externalProcess` is used, the SDS for storing stdin that is to be sent to the external process is monitored through the event loop.

When `externalProcessHandlers` is used, in addition to monitoring the SDS for stdin, the callback SDS is monitored for changes as well.

The overview below shows the relevant part of the call graph for monitoring the SDSs that are monitored by the `externalProcess` and `externalProcessHandlers` tasks:

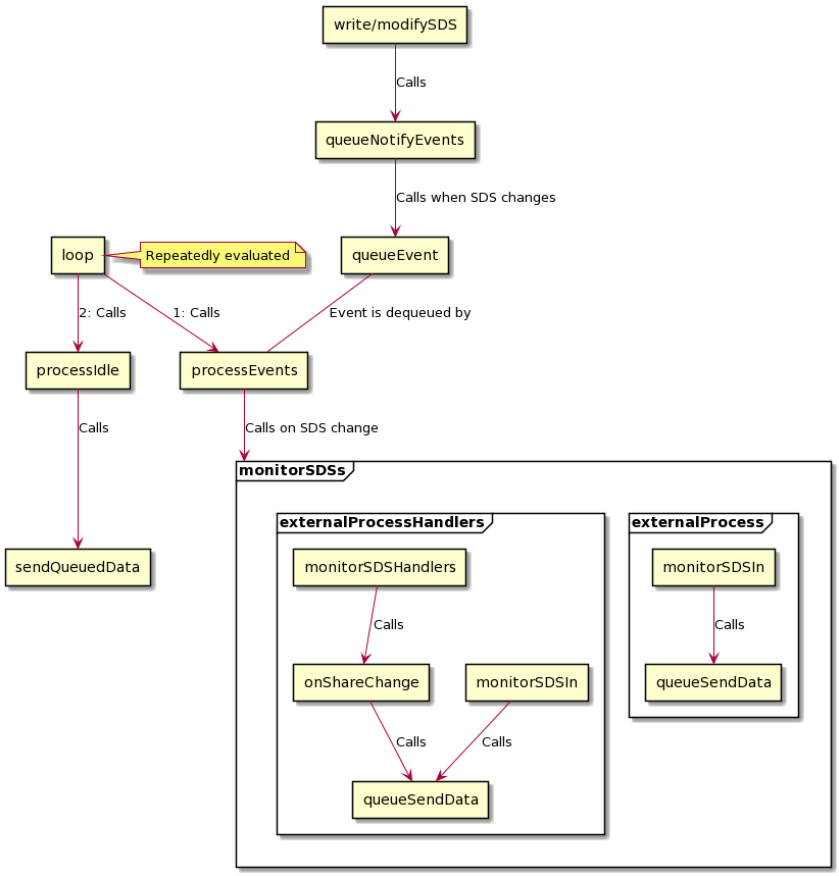


Figure 9.5: Monitoring the SDS for stdin.

As the figure illustrates, if the content of the SDS for stdin is changed, the content of the SDS is queued for sending. The SDS is emptied afterwards. Similarly, when the value of the callback SDS (`externalProcessHandlers`) is modified, the `onShareChange` callback is evaluated. The `onShareChange` callback may produce output which is queued for sending. The approach to sending data in the replacement implementation is explained in more detail in section **3.3**. The data that is queued is sent to the stdin of the external process, provided that the external process has not terminated.

9.4.2 Monitoring the output of the external process

Following the startup phase, the file descriptors involved in communication with the I/O multiplexing mechanism are monitored for readability. The aim is to provide the output of the external process to the end-user. The output is provided to the end-user through user-defined callbacks or a SDS. The `externalProcess` task makes use of a SDS to process the output of the external process. The `externalProcessHandlers` task makes use of callbacks to process the output of the external process.

The figure included below illustrates the relevant part of the call graph.

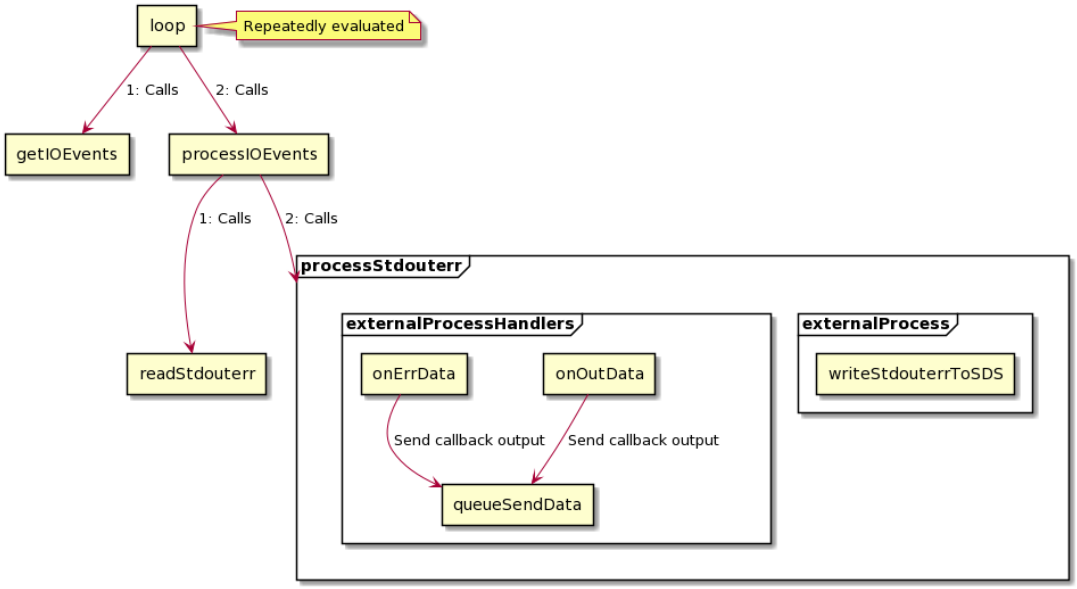


Figure 9.6: Monitoring the external process for output.

`getIOEvents` returning a readability event indicates that data can be read (Linux, macOS) or has been read (Windows). In either case, the data ends up being processed by `processIOEvents`. `processIOEvents` reads the output of the external process. Afterwards, `processIOEvents` proceeds by processing the output of the external process. This is done differently depending on whether `externalProcess` or `externalProcessHandlers` is used.

`externalProcess` stores the output data in a SDS. As stated, if using a pseudoterminal, the stdout and stderr output will both end up being stored as stdout in the SDS. This is a result of it not being possible to separate stdout and stderr when using a pseudoterminal. When using pipes, stderr and stdout is correctly separated within the SDS.

`externalProcessHandlers` evaluates a callback in response to output being received. When using a pseudoterminal, the `onOutData` callback is evaluated, the received data is provided to the callback. When using pipes, the `onOutData` or `onErrData` callback is evaluated, depending on whether the output was written to stderr or stdout by the external process. If a pseudoterminal is used, output being received will always lead to `onOutData` being evaluated. The `onOutData` and `onErrData` callbacks may produce output, which is queued to be sent. The output is sent to the stdin of the external process at a later point in time. The process of sending data in the replacement implementation is explained in section **3.3**.

9.4.3 Monitoring for termination

The following illustration shows how processes are monitored for termination in the replacement implementation:

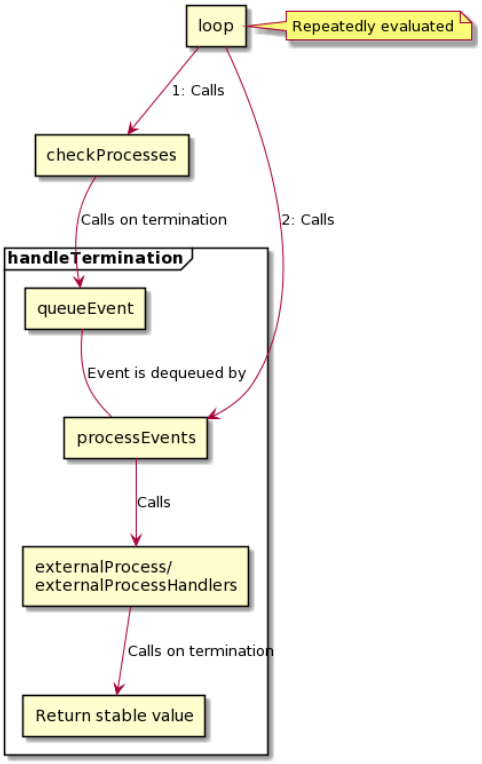


Figure 9.7: Monitoring external processes for termination.

Every iteration of the event loop, the external processes that are being executed are monitored for termination. This is done through the `checkProcesses` function. If a process terminated, an event is queued for the `externalProcess`/`externalProcessHandlers`. This event is processed by returning a stable value.

9.5 Summary

To summarize, the replacement implementation provides two tasks for executing an external process and communicating with it through IPC. The tasks allow to provide input to the external process that was executed and processing the output of the external process. The `externalProcess` task allows to communicate with an external process in the same manner as the `externalProcess` task that is provided in the existing implementation. The `externalProcessHandlers` is a new task. `externalProcessHandlers` allows to communicate with the external process using a callback record. This allows to communicate with the external process in a style that is consistent with how the network I/O functionality provides communication to the end-user.

The replacement implementation improves on the points of improvement that were identified for the existing IPC implementation. Concretely, this means that:

- IPC is now provided through the I/O multiplexing concept. This is an improvement on the time-based polling approach that is taken by the existing implementation. As a result, the monitor phase of the HTTP server, the network I/O tasks and IPC are now very similar. This increases the consistency and maintainability of the iTasks I/O functionality. Furthermore, it allows for code reuse.
- When using a pseudoterminal, all output of the external process is processed as stdout output. This is a consequence of not being able to separate stdout and stderr output when using a pseudoterminal. The existing implementation has a bug which would make it possible for data to end up being processed as stderr or stdout based on timing. As a result the behavior of the `externalProcess`

task is the existing implementation is not deterministic. The behavior of the `externalProcess` task is deterministic in the replacement implementation.

- Writing the output to the external process is now done using a non-blocking approach. Performing I/O operations in a non-blocking manner is a general goal of the thesis. This is a necessary step for being able to horizontally scale iTasks applications. Furthermore, this makes sure the iTasks program will not block while sending data. If the iTasks program blocks it may become (temporarily) unresponsive.
- The replacement implementation introduces the `externalProcessHandlers` task, which allows to communicate with an external process using a callback record. As a result, there is an option for using IPC in a similar style as the network I/O tasks (`tcplisten` and `tcpconnect`).

Chapter 10

Benchmarking the iTasks HTTP Server

Most iTasks applications rely on the iTasks HTTP server as iTasks is a framework for building web applications. As a consequence, increasing the performance and scalability of the HTTP server is beneficial for most iTasks applications. The HTTP server is affected by replacing the I/O multiplexing mechanism and other changes to the internal implementation. As a result, it is interesting to investigate how the replacement implementation scales and performs compared to the existing implementation.

In the problem description it is stated that replacing `select` for `epoll`, `kqueue` and `IOCP` is not expected to result in significant performance benefits. The reasoning behind this statement is that the performance benefits only become significant as thousands of file descriptors are being monitored [9]. However, it was not merely the I/O multiplexing mechanism that was changed. For example, the SDSs of connected clients are read significantly less often thanks to the new approach to the `onShareChange` and `onTick` callbacks. These changes and other changes may also have resulted in performance benefits or drawbacks. As a result, benchmarking the HTTP server is considered to be worthwhile.

The iTasks HTTP server was benchmarked using two methods for testing the scalability of the server. The first method makes use of a load testing tool that is called `hey` [16]. `hey` can be used to send a given number of HTTP requests to a server and measures the time it takes for a response to arrive. It provides various statistics on the time it takes to send requests and receive responses.

To explicitly test the latency of task events, which are sent using the WebSocket protocol, the HTTP server was benchmarked using javascript. This was done because iTasks task events are sent and handled by the browser through javascript. Task events are events that result from user input of clients that interact with iTasks web applications. Task events are described in more detail in chapter **4**.

This chapter first discusses the measurements that were obtained by performing the benchmark and the hardware that was used to obtain the measurements. The remainder of the chapter contains an explanation of how the benchmark results may be reproduced.

10.1 Hardware specification

This section lists the hardware that was used to host the HTTP server when performing the benchmark for each operating system that was used. The used hardware is specified to increase the reproducibility of the actual results. Nonetheless, the results of the benchmark should show a similar trend regardless of the hardware that is used.

Linux

Component	**Specification**
CPU	Intel i7-5600U (4) @ 3,2GHz
RAM	12GB DDR3L-SDRAM
Operating system	Linux, kernel version: 5.12.11

MacOS

Component	Specification
CPU	Intel i5-5287U (4) @ 2,9GHz
RAM	8GB
Operating system	macOS Big Sur 11.4

Windows

Component	Specification
CPU	Intel i5-5300U (4) @ 2,3GHz
RAM	8GB DDR3L-SDRAM
Operating system	Windows 10 Pro version: 20H2

10.2 Benchmarking the HTTP server using hey

This section describes the process of benchmarking the HTTP server using hey and discusses the benchmark results.

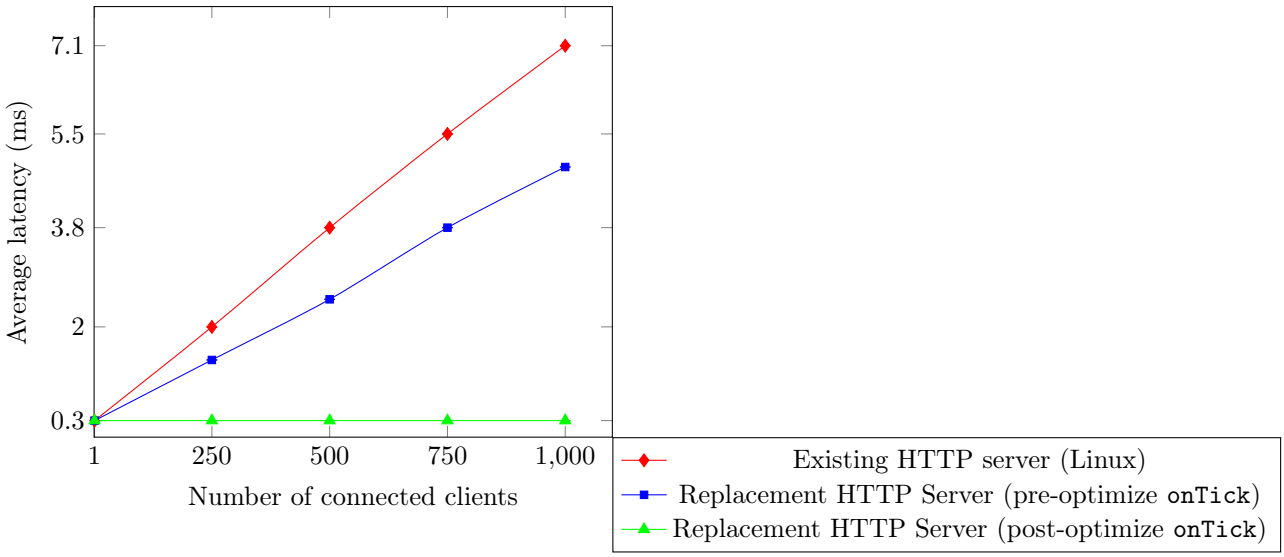
Using a program to setup idle connections, a given number of idle connections to the HTTP server are created. Afterwards, using the `hey` tool, 1000 measurements are performed to benchmark the HTTP server. A measurement involves sending a request to the HTTP server and waiting for the response. Using these measurements, `hey` returns various statistics that indicate the performance of the HTTP server:

- Latency denotes the time it takes for a client to send a request and retrieve a response.
- Resp. wait denotes the time hey spent waiting for the iTasks HTTP server to return a response after performing the request.
- Average denotes the average latency for the given number of connected clients.
- Slowest denotes the latency of the measurement which had the highest latency.
- Fastest denotes the latency of the measurement which had the lowest latency.

These statistics for given amounts of idle connections form the results of the benchmark.

On Linux, measurements were performed on a version that preceded the optimization of the `onTick` callback. The optimization of the `onTick` callback is described in more detail in chapter **5**. These results were also included to indicate the effect that the `onTick` optimization had on the scalability of the HTTP server (see figure **10.2.1**). Accordingly, the measurements before optimizing the `onTick` callback are labelled "pre-optimize". The measurements after optimizing the `onTick` callback are labelled "post-optimize".

10.2.1 Benchmark results for the Linux operating system

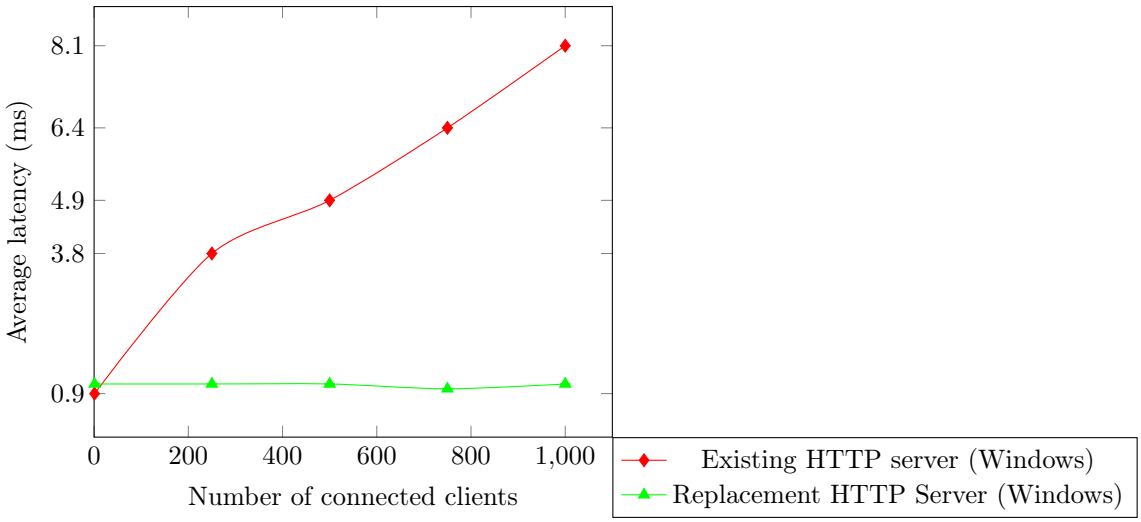


Existing HTTP server				
N clients	Average (ms)	resp. wait (avg, fast, slow) (ms)	slowest (ms)	fastest (ms)
1	0.3	0.2, 0.1, 2.4	2.4	0.2
250	2	1.9, 1.4, 9.1	9.9	1.5
500	3.8	3.6, 2.8, 7.6	8.3	2.9
750	5.5	5.4, 4.2, 14.1	14.9	4.3
1000	7.1	7.0, 5.4, 14.8	15.6	5.6

Replacement HTTP server pre-optimize				
N clients	Average (ms)	resp. wait (avg, fast, slow) (ms)	slowest (ms)	fastest (ms)
1	0.3	0.3, 0.2, 1.9	2.0	0.2
250	1.4	1.3, 0.9, 3.7	3.8	1.0
500	2.5	2.4, 1.6, 4.3	4.4	2.0
750	3.8	3.7, 2.8, 6.8	7.0	2.9
1000	4.9	4.8, 3.7, 6.9	7.7	3.9

Replacement HTTP server post-optimize				
N clients	Average (ms)	resp. wait (avg, fast, slow) (ms)	slowest (ms)	fastest (ms)
1	0.3	0.2, 0.1, 2.4	2.4	0.2
250	0.3	0.3, 0.2, 1.2	3.6	0.2
500	0.3	0.3, 0.2, 1.3	1.5	0.2
750	0.3	0.3, 0.2, 1.3	1.3	0.2
1000	0.3	0.3, 0.2, 1.5	1.9	0.2

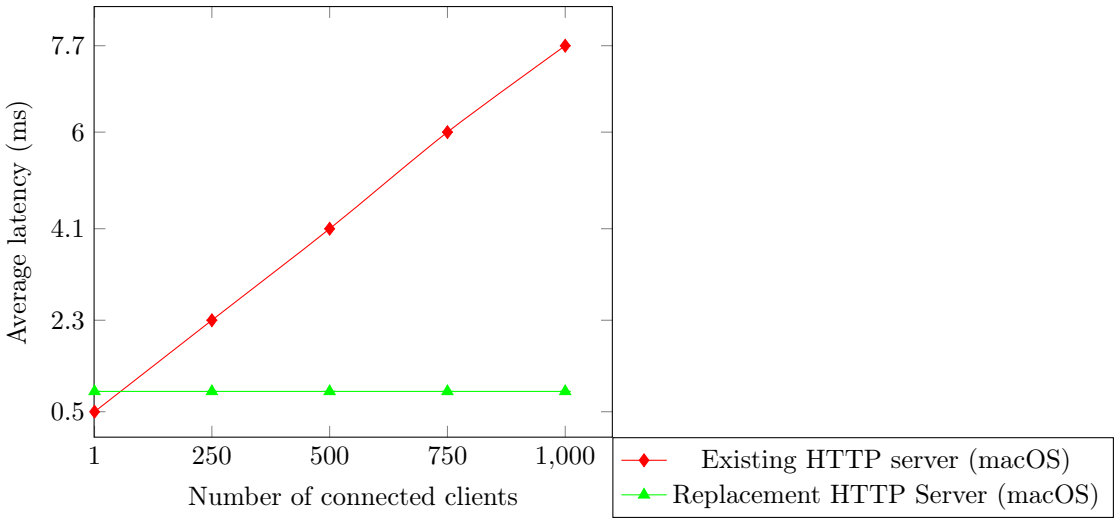
10.2.2 Benchmark results for the Windows operating system



Existing HTTP server				
N clients	Average (ms)	resp. wait (avg, fast, slow) (ms)	slowest (ms)	fastest (ms)
1	0.9	0.5, 0.3, 1.6	330.8	0.5
250	3.8	3.2, 1.8, 8.4	332.2	2.0
500	4.9	4.4, 3.1, 15.0	319.2	3.3
750	6.4	5.8, 4.5, 14.6	337.6	4.7
1000	8.1	7.6, 5.9, 24.2	329.1	6.1

Replacement HTTP server				
N clients	Average (ms)	resp. wait (avg, fast, slow) (ms)	slowest (ms)	fastest (ms)
1	1.1	0.6, 0.4, 2.8	321	0.5
50	1.1	0.6, 0.4, 4.5	322	0.5
250	1.1	0.6, 0.4, 2.3	321	0.5
500	1.1	0.6, 0.4, 2.5	322	0.5
750	1.0	0.6, 0.4, 3.5	324	0.5
1000	1.1	0.7, 0.4, 2.9	325	0.5

10.2.3 Benchmark results for the macOS operating system



Existing HTTP server				
N clients	Average (ms)	resp. wait (avg, fast, slow) (ms)	slowest (ms)	fastest (ms)
1	0.5	0.4, 0.3, 2.9	4.8	0.3
250	2.3	2.2, 1.7, 6.7	10.1	1.8
500	4.1	4.0, 3.0, 11.9	15.2	3.1
750	6.0	5.8, 4.5, 19.4	21.2	4.6
1000	7.7	7.6, 5.9, 24.2	28.1	5.9

Replacement HTTP server				
N clients	Average (ms)	resp. wait (avg, fast, slow) (ms)	slowest (ms)	fastest (ms)
1	0.9	0.7, 0.3, 6.0	6.4	0.4
250	0.9	0.7, 0.3, 6.6	6.7	0.4
500	0.9	0.7, 0.3, 6.8	6.9	0.4
750	0.9	0.7, 0.3, 6.8	7.0	0.4
1000	0.9	0.7, 0.4, 6.2	8.3	0.4

10.2.4 Conclusion

The `hey` benchmark results show a significant improvement in scalability on all platforms when using the replacement implementation. When a single file descriptor is monitored, the replacement implementation tends to perform slightly worse or equal to the existing implementation. This should not be considered a problem since the latency is low in this case regardless. As more clients are monitored, the latency of the existing implementation scales somewhat linearly. The latency of the replacement implementation remains constant as more clients are monitored on all platforms.

10.3 Benchmarking the WebSocket connection of the HTTP server

This section discusses how the WebSocket connection of the HTTP server was benchmarked. The WebSocket connection of the iTasks HTTP Server is used to communicate task events to the HTTP server. Javascript is used to be able to react to user input through sending task events to the HTTP server. Some examples of user input that lead to a task event are clicking a button or entering a value in an input field of a web application that is developed using iTasks. Javascript notifies the iTasks HTTP server of such user input through a WebSocket connection. This is just a regular socket connection but

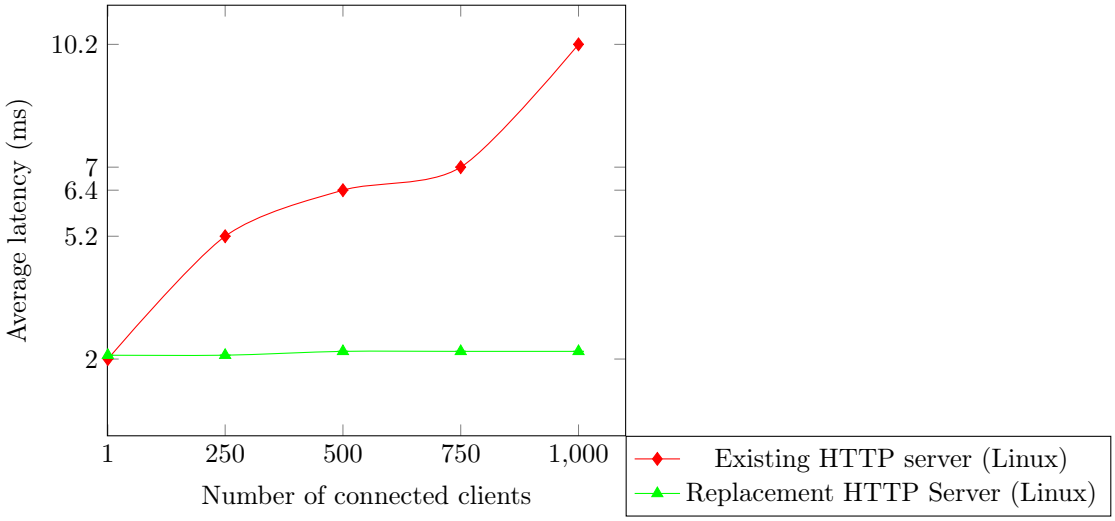
it uses the WebSocket protocol instead of the HTTP protocol. iTasks notifies the browser of the client of any changes that should be performed as a result of the user input. The required changes are processed by the browser using javascript.

This process is described in more detail in chapter **4**. Using javascript, 1000 task events were sent to the iTasks HTTP server and statistics were collected on the response time of the HTTP server. This measures how long it takes for javascript to be notified by the HTTP server after sending a task event. This measurements were performed while the HTTP server was monitoring a given number of idle connections. The idle connections are used to measure the scalability of the HTTP server.

The statistics that are measured for the WebSocket benchmark are:

- The average latency of the 1000 measurements that were performed.
- The latency of the measurement that had the highest latency.
- The latency of the measurement that had the lowest latency.

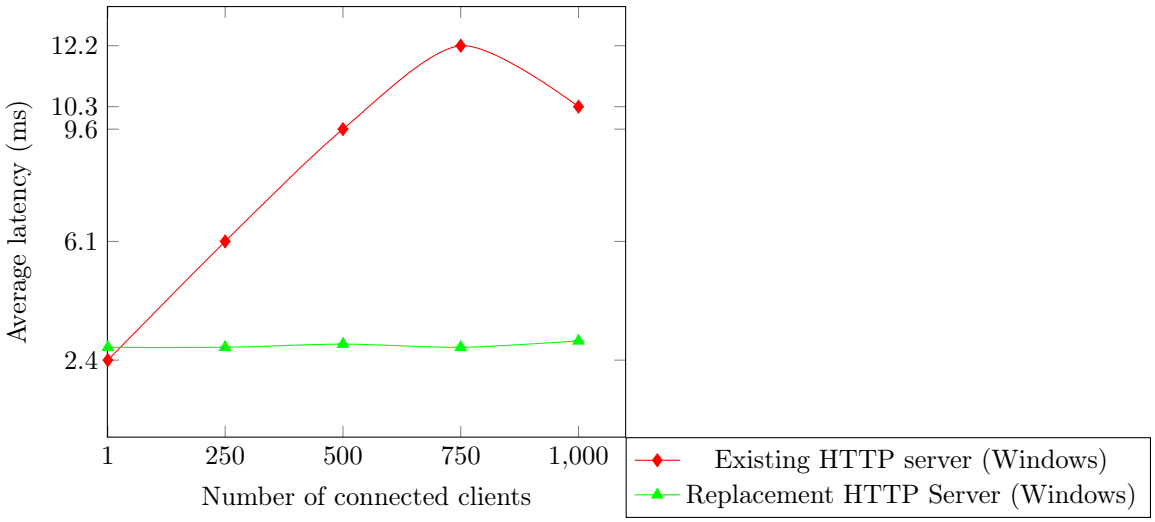
10.3.1 Benchmark results for the Linux operating system



Existing HTTP server			
N clients	Average (ms)	slowest (ms)	fastest (ms)
1	2.0	7.6	0.5
250	5.2	41.3	2.5
500	6.4	77.8	4.2
750	7.0	40.5	4.5
1000	10.2	56.6	7.4

Replacement HTTP server			
N clients	Average (ms)	slowest (ms)	fastest (ms)
1	2.1	4.3	1.7
250	2.1	10.2	1.6 (not a typo)
500	2.2	18.3	1.7
750	2.2	56.1	1.7
1000	2.2	9.0	1.7

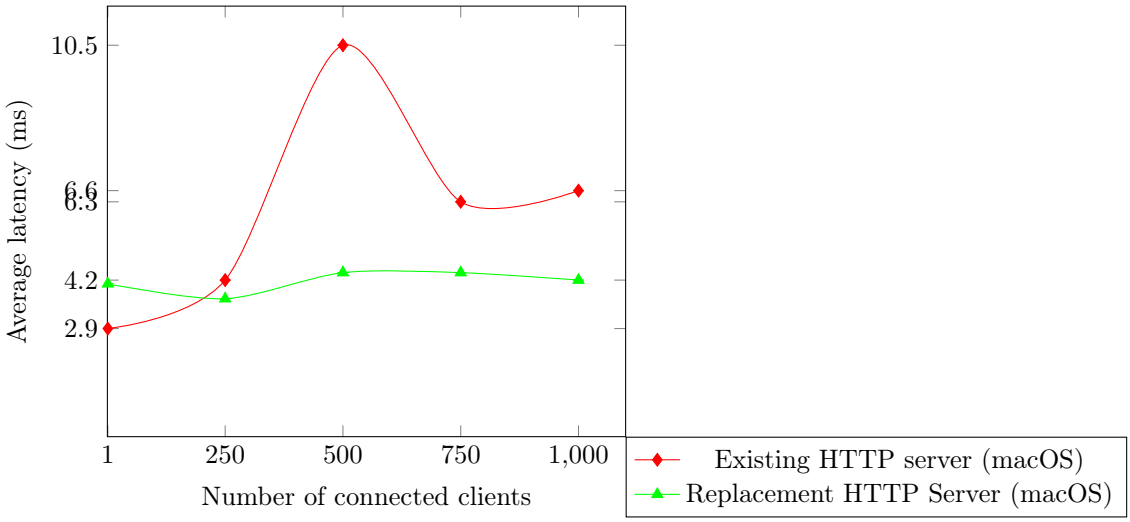
10.3.2 Benchmark results for the Windows operating system



Existing HTTP server			
N clients	Average (ms)	slowest (ms)	fastest (ms)
1	2.4	11.7	1.0
250	6.1	12.7	3.6
500	9.6	17.6	5.5
750	12.2	26.5	6.3
1000	10.3	29.2	7.6

Replacement HTTP server			
N clients	Average (ms)	slowest (ms)	fastest (ms)
1	2.8	6.2	2.2
250	2.8	6.3	2.2
500	2.9	21.1	2.2
750	2.8	19.7	2.2
1000	3.0	50.5	2.3

10.3.3 Benchmark results for the macOS operating system



Existing HTTP server			
N clients	Average (ms)	slowest (ms)	fastest (ms)
1	2.9	9.3	1.1
250	4.2	23.6	0.05
500	10.5	23.4	4.9
750	6.3	20.5	0.04
1000	6.6	27.4	0.05

Replacement HTTP server			
N clients	Average (ms)	slowest (ms)	fastest (ms)
1	4.1	21.0	0.2
250	3.7	11.4	0.05
500	4.4	13.4	0.1
750	4.4	37.1	2.6
1000	4.2	76.0	2.5

10.3.4 Conclusion

The WebSocket benchmark shows an improvement in scalability on all platforms. Compared to the `hey` benchmark, the growth in latency is less stable for the existing HTTP server. Like with the `hey` benchmark, the replacement implementation is slightly slower when a low number of file descriptors is monitored. This is not considered to be a problem as the latency is relatively low in this case either way. The instability of the existing HTTP server is certainly interesting to investigate. However, this was not deemed to be a worthwhile investment of time since the aim of the thesis is to replace the existing HTTP server.

10.4 Reproducing the results

10.4.1 Prerequisites for benchmarking the iTasks HTTP server

This section describes the prerequisites for benchmarking the iTasks HTTP server.

The iTasks web application used to perform the benchmarks is the BasicAPIExamples project included in any iTasks distribution.

The iTasks distributions that were used for performing the benchmark are accessible at `https://gitlab.com/GijsAlberts/itasks-httpserver-benchmark`. Furthermore, this repository includes the files that are necessary to perform the benchmark on the operating system of choice. To perform the benchmark, it is recommended to clone or download this repository.

It is assumed that the person performing the benchmark has access to the gcc compiler. Installation instructions for the Windows platform are listed below (see Windows specific prerequisites). On Linux/-macOS, it should be simple to install gcc through your package manager (brew on macOS).

Linux/macOS specific prerequisites

On macOS/Linux, there is a maximum number of file descriptors that may be opened by a process at once. This maximum number generally is 1024 due to the limitation of `select`. Monitoring over 1023 file descriptors using `select` results in undefined behavior. However, the limit may be an even lower number. This can result in problems when performing the benchmark. To retrieve the current maximum, the `ulimit -n` command may be performed.

If `ulimit -n` returns a number below 1000, it is required to increase it to 1024. To increase the number of file descriptors that may be monitored for the terminal session, the `ulimit -n 1024` command may be used to set the maximum to 1024. This command should be executed in the terminal that is used for creating the idle connections and the terminal that is used for executing the BasicAPIExamples executable.

If you wish, it may be possible to perform the benchmark on more recent iTasks distributions without extra effort on **Linux** and **macOS**.

Windows specific prerequisites

It is assumed that the person performing the benchmark downloaded MSYS2. MSYS2 can be downloaded from: `https://www.msys2.org/`. It provides a terminal called mintty which may be used to perform the commands on Windows.

The gcc compiler that is used to perform the benchmark can be installed through msys2 using the `pacman -Sy mingw-w64-x86_64-gcc` command. It should then be added to the system PATH. Assuming msys2 has been installed to `C:/msys64`, the command to locally add gcc to the path is:
`export PATH=$PATH:/c/users/msys64/mingw64/bin`.

When using **Windows**, it is highly recommended to make use of the iTasks distribution that is provided by the repository for benchmarking the existing implementation. This is the case because the existing implementation has a maximum limit of monitoring 64 file descriptors by default using `select`. The iTasks distribution of the existing implementation that is included in the repository comes with an increased limit.

10.4.2 Reproducing the results that were obtained using the hey tool

This section describes the process and results of benchmarking the iTasks HTTP server implementations using `hey`. The hey load testing tool may be downloaded from: `https://github.com/rakyll/hey`. It is assumed that the `hey` load testing tool has been installed. To benchmark the HTTP server, the following steps are performed sequentially:

1. The first step involves executing the BasicAPIExamples executable, which sets up the HTTP server. The BasicAPIExamples project can be built using the following command on **Linux**/**macOS**:

   ```
   cd iTasks-httpserver-benchmark/{macOS|linux-x86}/clean-bundle-
       complete-{select|kqueue|epoll}/bin
   ./cpm project ../examples/iTasks/BasicAPIExamples.prj build
   ```

   The following command is used on **Windows** (the Windows clean-bundle-complete has a different directory structure):

   ```
   cd itasks-httpserver-benchmark/windows/clean-bundle-complete-{
       iocp|select}/Examples/iTasks
   ../../cpm.exe project BasicAPIExamples.prj build
   ```

   This will create an executable called `BasicAPIExamples.exe`. `BasicAPIExamples.exe` is located at:

   ```
   itasks-httpserver-benchmark/{linux-x86|macOS|windows}/clean-
       bundle-complete-{select|iocp|epoll|kqueue}/Examples/iTasks
   ```

   This executable should be executed. Before doing so, make sure that `ulimit -n` is greater than or equal to 1024.

2. The second step involves executing a C program which is used to setup a given number of idle connections to the server. It is recommended to use a new terminal window.

   The program used for **Linux**/**macOS** is called `add-idle-connections-posix.c`, it is included in the benchmark repository. The C file can be compiled using the following command.

   ```
   gcc itasks-httpserver-benchmark/add-idle-connections-posix.c -
       o add-idle-connections
   ```

   The `add-idle-connections` executable may then be executed to setup the idle connections. The program takes the number of idle connections to setup as an argument. An example of using the executable to setup 250 idle connections is included below:

```
./add-idle-connections 250
```

Note that if the program is made to terminate, the idle connections are closed. Therefore, do not terminate the program before performing the measurements. Make sure that the `BasicAPIExamples.exe` that was built in step 1 is running.

The program used for **Windows** is called `add-idle-connections-windows.c`. This file is included in the benchmark repository. This program may be compiled using

```
gcc itasks-httpserver-benchmark/add-idle-connections-windows.c
    -o add-idle-connections.exe -lws2_32
```

The number of idle connections to setup is provided as an argument to the executable. To setup 250 idle connections the following command may be used:

```
./add-idle-connections.exe 250
```

Note that if the program is made to terminate, the idle connections are closed. Therefore, do not terminate the program before performing the measurements. Make sure that the `BasicAPIExamples.exe` that was built in step 1 is running.

3. The third step involves executing the hey program using the command included below. Make sure the program that is used to setup the idle connection has not been terminated.

```
hey -c 1 -n 1000 http://localhost:8080 > outputfile.txt
```

Note that port 80 is the default port on Windows instead of port 8080. The benchmark was executed on a localhost connection. This removes the risk of the network stability interfering with the results. By using this command, `hey` sends 1000 requests to localhost with a maximum of one connection being active at a time. `hey` establishes a connection, sends a request for the index of the web server 1000 times and then disconnects. The measurements are written to a file called `outputfile.txt` in this case. For obtaining the results of the hey benchmark, the command included above was used as well. This means that the results were obtained from performing 1000 measurements.

10.4.3 Reproducing the results that were obtained through the WebSocket benchmark

This section discusses the process and results of benchmarking the WebSocket connection that is used by the iTasks HTTP server implementations. The Websocket benchmark is performed using javascript.

Optional: disable reduction of timer precision

To get the most accurate results, it is required to (temporarily) disable the reduction of the timer precision in the browser settings. This can be considered **optional** as the results should show a similar trend either way. When obtaining the measurements that are included in this thesis, the reduction in timer precision was disabled to maximize the accuracy of the measurements.

Having a precise timer may harm your privacy and result in security issues [17] as long as the timer remains precise. It is highly recommended to re-enable the reduction in precision of the timer after performing the benchmark.

Disabling the reduction in accuracy of the timer can be done using the firefox browser. For performing the benchmark, the changes included below disable privacy features to increase the precision of the timer that is provided by the browser.

Using firefox, visit `about:config` using the URL bar. To maximalize the timer precision, make sure the following settings are set to the following values (consider using the search bar):

```
privacy.resistFingerprinting = false
privacy.resistFingerprinting.reduceTimerPrecision.jitter = false
privacy.resistFingerprinting.reduceTimerPrecision.microseconds = 0
```

```
privacy.reduceTimerPrecision = false
privacy.reduceTimerPrecision.unconditional = false
```

Again, it is highly recommended to (re-)enable the above privacy settings after performing the benchmark. The default value of `privacy.resistFingerprinting.reduceTimerPrecision.microseconds` is 1000.

Performing the WebSocket benchmark

To benchmark the WebSocket performance of the iTasks HTTP server, the following steps are performed sequentially:

1. The first step involves executing the BasicAPIExamples executable, which sets up the HTTP server. The BasicAPIExamples project can be built using the following command on **Linux/macOS**:

   ```
   cd iTasks-httpserver-benchmark/{linux-x86|macOS}/clean-bundle-complete-{select|kqueue|epoll}/bin
   ./cpm project ../examples/iTasks/BasicAPIExamples.prj build
   ```

   This will create an executable called `BasicAPIExamples.exe`. `BasicAPIExamples.exe` is located at:

   ```
   itasks-httpserver-benchmark/{linux-x86|macOS}/clean-bundle-complete-{select|epoll|kqueue}/Examples/iTasks
   ```

   Do not execute the executable yet.

   The following command is used on **Windows** (the Windows clean-bundle-complete has a different directory structure):

   ```
   cd itasks-httpserver-benchmark/windows/clean-bundle-complete-{iocp|select}/Examples/iTasks
   ../../cpm.exe project BasicAPIExamples.prj build
   ```

   This will create an executable called `BasicAPIExamples.exe`. `BasicAPIExamples.exe` is located at:

   ```
   itasks-httpserver-benchmark/windows/clean-bundle-complete-{select|iocp}/Examples/iTasks
   ```

   Do not execute the executable that was built yet.

2. The second step involves copying the `itasks-core.js` file that is included in the benchmark repository to the HTTP server js directory as follows:

   ```
   cp itasks-httpserver-benchmark/itasks-core.js itasks-httpserver-benchmark/{linux-x86|macOS|windows}/clean-bundle-complete-{select|kqueue|epoll|iocp}/Examples/iTasks/BasicAPIExamples-www/js
   ```

3. The third step involves executing the `BasicAPIExamples.exe` that resulted from building the project. `BasicAPIExamples.exe` is located at:

   ```
   itasks-httpserver-benchmark/{linux-x86|macOS|windows}/clean-bundle-complete-{select|iocp|kqueue|epoll}/Examples/iTasks
   ```

4. The fourth step involves executing a C program which is used to setup a given number of idle connections to the server. It is recommended to use a new terminal window.

   The program used for **Linux/macOS** is called `add-idle-connections-posix.c`, it is included in the benchmark repository. The C file can be compiled using the following command.

```
gcc itasks-httpserver-benchmark/add-idle-connections-posix.c -
    o add-idle-connections
```

The `add-idle-connections` executable may then be executed to setup the idle connections. The program takes the number of idle connections to setup as an argument. An example of using the executable to setup 250 idle connections is included below:

```
./add-idle-connections 250
```

Note that if the program is made to terminate, the idle connections are closed. Therefore, do not terminate the program before performing the measurements. Make sure that the `BasicAPIExamples.exe` that was built in step 1 is running.

The program used for **Windows** is called `add-idle-connections-windows.c`. This file is included in the benchmark repository. This program may be compiled using

```
gcc itasks-httpserver-benchmark/add-idle-connections-windows.c
    -o add-idle-connections.exe -lws2_32
```

The number of idle connections to setup is provided as an argument to the executable. To setup 250 idle connections the following command may be used:

```
./add-idle-connections.exe 250
```

Note that if the program is made to terminate, the idle connections are closed. Therefore, do not terminate the program before performing the measurements. Make sure that the `BasicAPIExamples.exe` that was built in step 1 is running.

5. The fifth step involves visiting `http://localhost:8080` or `http://localhost:80` (Windows) using the browser. By entering a value into the login field, the benchmark will start. The results of the benchmark are printed to the console of the browser. To access the firefox browser console, press `CTRL+SHIFT+K`. As the benchmark performs 1000 measurements, it may take 10 seconds before the output is printed when benchmarking the existing implementation.

Chapter 11

Conclusion

iTasks is a general-purpose framework for developing web applications. It is implemented in the pure functional programming language Clean. iTasks is a platform-independent framework, which means that iTasks applications may be executed on the Windows, macOS and Linux operating systems.

iTasks provides an HTTP server that is used to serve the web applications that are developed using iTasks. The iTasks HTTP server relies on network I/O to communicate with the clients that connect to the server. As a result, the applications that are developed using iTasks rely on network I/O. Furthermore, iTasks provides functionality that allows to perform network I/O as an end-user. Essentially, this allows the end-user to communicate over TCP using a custom communication protocol (or HTTP). To perform network I/O, sockets are used to abstract from network I/O communication channels. There are functions which allow the developer to:

- Create a socket.
- Establish a network connection to a peer using a socket. This happens either through accepting a connection request (server) or performing a connection request (client).
- Receive data from a socket, this leads to receiving data that was sent over the network connection that was established.
- Send data to a socket, this leads to sending data over the network connection that was established.

In addition, iTasks provides a means to execute an external process and communicate with it through IPC (inter-process communication). An external process can be seen as any kind of executable computer program. Computer programs have input and output channels. iTasks provides functionality for providing input to the external process that was executed. Likewise, iTasks provides functionality for reading the output of the external process that was executed. The process of providing input to an external process and reading the output of an external process involves IPC. The iTasks IPC implementation relies on I/O to communicate with the external process. To perform IPC, iTasks redirects the input and output channels of the external process that is executed to pipes. Pipes are used to abstract from IPC channels. As a result, data may be sent to the input channel of an external process by writing to a pipe. Similarly, the output of an external process may be received by reading from a pipe.

This thesis focuses on revisiting the existing network I/O and IPC implementation of iTasks. Revisiting the existing network I/O and IPC implementation is beneficial because the existing implementation has several drawbacks. Revisiting the existing implementation of network I/O and IPC lead to re-implementing the aforementioned functionality. The resulting implementation is named the replacement implementation. The replacement implementation provides solutions to the majority of the drawbacks of the existing implementation. The main drawbacks of the existing implementation are described in more detail below. The approach that is taken by the replacement implementation is inspired by the libuv library, which provides asynchronous I/O for node.js [10]. The research goal of this thesis and the preceding research internship was to figure out whether the approach that is taken by libuv could be applied to the existing architecture of iTasks. The conclusion is that that this is possible.

As stated, both network I/O and IPC rely on I/O. Performing I/O involves sending and receiving data. It may not always be possible to send or receive data over a network connection (network I/O). Similarly, it may not always be possible to receive output from an external process or send input to it (IPC). This poses the question of how programs that rely on I/O should handle the possibility of not immediately being able to receive or send data. There are various answers to the aforementioned question, which all have their advantages and drawbacks. The IPC and network I/O implementations each make use of a distinct solution for handling the possibility of not being able to receive or send data. As a consequence, network I/O and IPC are implemented through two separate concepts in the existing implementation.

In the case of network I/O, the iTasks program makes use of the `select` I/O multiplexing mechanism. The `select` I/O multiplexing mechanism enables a program to monitor a set of sockets. Using an operation, the subset of sockets that may receive or write data may be retrieved from the set of sockets that is monitored. The program may then react by reading data from sockets that have data available. Similarly, the program may react by sending data on the sockets for which data may be sent. In the context of the HTTP server, it is possible to retrieve the subset of connected clients for which data may be received, for instance.

In the case of IPC, iTasks makes use of a time-based solution to monitoring the pipes involved in communicating with external processes. This means that the program periodically attempts to read data from the monitored pipes to receive the output of external processes. Similarly, the program periodically attempts to send data to the monitored pipes to send input to external processes. This is done without having the knowledge of whether it is actually possible to read or send data.

As stated, iTasks is a platform-independent framework. The network I/O functionality of iTasks makes use of the `select` I/O multiplexing mechanism. The `select` I/O multiplexing mechanism is supported on the Linux, macOS and Windows operating systems. Generally, I/O multiplexing mechanisms allow to monitor both pipes and sockets. However, the Windows implementation of the `select` I/O multiplexing mechanism does not allow to monitor pipes. It is assumed that this lead to implementing IPC and network I/O according to two separate concepts in the existing implementation. However, `select` is not the only I/O multiplexing mechanism in existence. The `IOCP` I/O multiplexing mechanism does allow monitoring both pipes and sockets on the Windows platform. The replacement implementation makes use of `IOCP` to provide IPC and network I/O through a single concept on Windows. The emphasis lies on Windows since a drawback of the `IOCP` multiplexing mechanism is that it is not portable, unlike `select`. Linux and macOS provide the (non-portable) `epoll` and `kqueue` I/O multiplexing mechanisms, respectively. It was decided to replace `select` for `kqueue` on macOS and `epoll` on Linux in the replacement implementation as these mechanisms scale better. `select` can monitor at most 1023 file descriptors at once and retrieves events in $\mathcal{O}(n)$ time. The I/O multiplexing mechanisms that are used by the replacement implementation retrieve events in $\mathcal{O}(1)$ time and do not have a practical limit on the number of file descriptors that may be monitored. Like `IOCP`, `kqueue` and `epoll` allow to monitor both pipes and sockets as well. Thanks to replacing the `select` I/O multiplexing mechanism, IPC and network I/O are provided through a single concept in the replacement implementation. The drawback of the new approach is that the replacement implementation internally makes use of different I/O multiplexing mechanisms, depending on the target platform. `kqueue` and `epoll` are very similar and the `epoll` (Linux) implementation has essentially been translated to a `kqueue` (macOS) implementation. However, the `IOCP` mechanism takes a significantly different approach than the other mechanisms. This resulted in having to abstract away from the differences between `IOCP` and `kqueue`/`epoll`. This was necessary as iTasks is a platform-independent framework and the network I/O and IPC functionality should behave the same regardless of which platform is used.

The existing network I/O and IPC implementation make use of blocking I/O operations in several cases. To give an example, sending data happens in a blocking manner for both IPC and network I/O. At the time of writing, iTasks programs are single threaded. This means that iTasks programs may block while performing I/O operations. As a result of blocking while performing an I/O operation, the iTasks program is unable to perform other operations. This is undesirable as the iTasks program is unresponsive as long as the program blocks. In the replacement implementation, I/O operations are performed in a non-blocking manner. This complicates the implementation of these operations but avoids the drawback of leaving the iTasks program unresponsive while the operation blocks. Moreover, implementing I/O operations in a non-blocking manner is required for horizontally scaling iTasks web applications. This is

a future aim of the iTasks project (see chapter **12**).

The replacement implementation of network I/O alleviated performance bottlenecks that exist within the existing HTTP server implementation. The implementation of the HTTP server makes use of SDSs (Shared Data Sources) to read and write shared data. SDSs allow dealing with various data sources in a uniform manner, abstracting away from the actual source of the data. Some examples of data sources are shared memory, databases and files. One SDS that is used by the HTTP server makes use of shared memory as a means of storage. Another SDS makes use of a file as a means of storage. As a result, reading from and writing to these SDSs is relatively expensive. The performance bottlenecks of the existing implementation are caused by SDS writes and reads that occur for every client during every event loop iteration. The SDS reads and writes were a result of the implementation of the `onTick` and `onShareChange` callbacks. The implementation of these callbacks was optimized, saving a significant number of SDS reads and writes while providing the same functionality. As a consequence, it was considered to be worthwhile to measure the effects of the optimizations that were implemented. This lead to benchmarking the iTasks HTTP server. The benchmark involved a comparison of the performance of the iTasks HTTP server implementations as the number of clients that connected increased. The results of this benchmark show that the replacement implementation of the HTTP server scales significantly better than the existing one.

Chapter 12

Future work

This chapter provides suggestions for future work that is related to the subject of this thesis.

A future goal of the iTasks project that is related to this thesis is to be able to horizontally scale iTasks servers. Horizontally scaling iTasks servers involves being able to use all the threads of the CPU on which the iTasks server is hosted for serving the iTasks application. In this case, communication can occur through IPC (inter-process communication). Furthermore, it involves being able to have several iTasks servers form a distributed network to share the workload of an iTasks application. Communication between servers in a distributed network can occur through network I/O. A load balancer could be used to distribute the requests of the clients that interact with the iTasks applications over the servers involved in the distributed network. Being able to horizontally scale is important as this would result in iTasks applications being able to serve a larger number of simultaneous users. Not being able to do this is expected to become an issue in the near future for using iTasks in large-scale projects.

Arjan Oortgiese researched the topic of providing iTasks applications in a distributed manner [18]. The product of this research is interesting but a practical problem is that it is a synchronous (blocking) implementation. Having a distributed network of iTasks servers communicate in a blocking manner is unfeasible.

This thesis resulted in implementing all IPC and network I/O operations in a non-blocking manner. This is a necessary step for being able to horizontally scale iTasks applications. For example, this could be used to implement an asynchronous version of the distributed iTasks implementation that was developed by Arjan Oortgiese.

A further point of future work is that the current implementation of the `onShareChange` callback is fragile. The problem is that the implementation currently relies on refresh events. A refresh event for a task occurs when the SDS that is monitored by the task is modified, the implementation of the `onShareChange` callback relies on this. However, a refresh event may not just be caused by a SDS being modified. As a result, the `onShareChange` callback may be evaluated more often than is necessary. Essentially, this issue can be solved by doing "something" **only** whenever the SDS is modified and being able to react to this by processing the `onShareChange` callback. Introducing a new task event is not considered to be a good idea as this may break already existing tasks. In the existing implementation, the `onShareChange` is evaluated every event loop iteration instead. This is considered to be a bug as well. Therefore, the approach taken by the replacement implementation does not introduce a new problem. However, this problem remains to be properly solved.

Bibliography

[1] *sys_ epoll - making poll fast*
Retrieved from:
`https://lwn.net/Articles/14168/`
on 24 February 2021.

[2] *select(2) — Linux manual page*
Retrieved from:
`https://man7.org/linux/man-pages/man2/select.2.html`
on 24 February 2021.

[3] *select function (winsock2.h) - MSDN*
Retrieved from:
`https://docs.microsoft.com/en-us/windows/win32/api/winsock2/nf-winsock2-select`
on 28 May 2021.

[4] *recv(2) — Linux manual page*
Retrieved from:
`https://man7.org/linux/man-pages/man2/recv.2.html`
on 6 June 2021

[5] *Design overview*
Retrieved from:
`https://docs.libuv.org/en/v1.x/design.html`
on 24 February 2021.

[6] *Calling C functions from Clean*
Retrieved from:
`https://svn.cs.ru.nl/repos/clean-tools/tags/clean-2-1-0/htoclean/CallingCFromClean.html`
on 1 March 2021.

[7] R. Plasmeijer, M. van Eekelen
Let-Before Expression: Local Constants defined between Guards
Retrieved from:
`https://clean.cs.ru.nl/download/html_report/CleanRep.2.2_5.htm#_Toc311798006`
on 1 March 2021.

[8] R. Plasmeijer, M. van Eekelen
Basic Ideas behind Uniqueness Typing
Retrieved from:
`https://clean.cs.ru.nl/download/html_report/CleanRep.2.2_11.htm#_Toc311798093`
on 2 March 2021.

[9] *libevent – Benchmark*
Retrieved from:
`https://libevent.org/`
on 14 April 2021.

[10] *Libuv - Cross-platform asynchronous I/O*
Retrieved from:
https://github.com/libuv/libuv
on 2 May 2021.

[11] *connect — Linux manual page*
Retrieved from:
https://man7.org/linux/man-pages/man2/connect.2.html
on 3 May 2021.

[12] *Mac OS X Manual Page For connect(2) - Apple Developer*
Retrieved from:
https://developer.apple.com/library/archive/documentation/System/Conceptual/ManPages_iPhoneOS/man2/connect.2.html
on 3 May 2021.

[13] *WSARecv function*
Retrieved from:
https://docs.microsoft.com/en-us/windows/win32/api/winsock2/nf-winsock2-wsarecv
on 4 May 2021.

[14] *The Web Client Runtime*
Retrieved from:
https://gitlab.com/clean-and-itasks/itasks-sdk/-/blob/master/Documentation/Design/client-runtime.md#the-web-client-runtime
on 9 May 2021.

[15] I. Fette, A. Melnikov
The WebSocket Protocol
Retrieved from:
https://www.rfc-editor.org/rfc/pdfrfc/rfc6455.txt.pdf
on 10 May 2021.

[16] *hey - github*
Retrieved from:
https://github.com/rakyll/hey
on 11 May 2021.

[17] *Mitigations landing for new class of timing attack*
Retrieved from:
https://blog.mozilla.org/security/2018/01/03/mitigations-landing-new-class-timing-attack/
on 14 June 2021.

[18] A. Oortgiese
A Distributed Server Architecture for Task Oriented Programming
Retrieved from:
https://www.ru.nl/publish/pages/769526/arjan_oortgiese.pdf
on 17 June 2021.

[19] D. Stenberg
poll vs select vs event-based
Retrieved from:
https://daniel.haxx.se/docs/poll-vs-select.html
on 07 July 2021.

[20] *iTasks - The Task-Oriented Programming Framework*
Retrieved from:
http://www.itasks.org/
on 07 July 2021.

[21] E. Klitzke
Stdout Buffering
`https://eklitzke.org/stdout-buffering`
on 07 July 2021.