School of Computation, Information and Technology - Informatics
Technical University of Munich

Bachelor's Thesis in Informatics: Games Engineering

Ubi-Interact Python Node - Implementation and Testing

Anian Kalb

School of Computation, Information and Technology - Informatics
Technical University of Munich

Bachelor’s Thesis in Informatics: Games Engineering

Ubi-Interact Python Node - Implementation and Testing

Ubi-Interact Python Node - Implementation und Testen

Author:	Anian Kalb
Supervisor:	Prof. Gudrun Klinker, Ph.D.
Advisors:	Sandro Weber, Ph.D.
Submission Date:	March 15, 2024

Eidesstattliche Erklärung

Ich versichere hiermit, dass ich diese Bachelor's Thesis selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, March 15, 2024

Anian Kalb

Abstract

This thesis will explain the implementation of a client node in python for the UBI-Interact framework. The role of the UBII framework is to handle communication in a distributed system. To understand the concrete implementation, the functionality of the framework will first be elaborated. This includes the role of the implemented client node as well as the underlying connection mechanisms, communication protocols, and message formats inside the framework. Through this process, the relevance of concurrency in the context of network communication will analysed. Then, two concrete python implementations for concurrency will be introduced, and the decision behind the choice for one of the concepts will be discussed. Following this, the software structure of this project will be explained, and the process of how the different parts of the python client node implement the required functionality will be elaborated on. The goal of the thesis is to give the reader an understanding of the UBII framework in general as well as language-specific solutions for its specific problems.

Contents

1 Introduction and Motivation

IT systems play a significant role in nearly every aspect of modern life, and with those systems expanding, there are more and more applications for distributed systems. One essential aspect of distributed systems is the communication between the different components of the system. Therefore, a fitting implementation that handles communication inside the system is necessary to share data efficiently. This communication consists of many parts that need to be considered. On the lower level, the internet connection between the components is established. The correct type of connection has to be chosen, with well known methods being sockets or a REST server. On the higher level, in the system design, a protocol needs to define how the components send and receive messages. Typical approaches are a server-client structure, where clients send requests to the server and the server answers, or a peer-to-peer network, where every host can directly communicate. To then work with the data, it has to be encoded into a form that can be transmitted over the network and decoded at the receiver to work with it.

The UBI-Interact framework is one approach to handle communication. To do so, it defines a master node and client nodes. These nodes build a network with the master node as a central unit that directs communication between the client nodes by offering services that the clients can request and sending data from one client to another. To work with UBII, the developer has to use a client node to send and receive data. The easiest way for this to work is if the client node can be directly integrated into the system. However, this requires a client node specific to the system's environment. Therefore, to make Ubi-Interact available for a broader range of systems, client nodes must be developed for different languages.

This thesis will be about the implementation of one specific client node, a client node for python, the ubii-python-node v2 (https://github.com/SandroWeber/ubii-node-python-v2/tree/anian-kalb). Throughout the thesis, an overview of the UBII framework will be given, and the implementation of the python node v2 will be explained and tested. To answer why there is a need for a reimplementation of the existing python node, it is important to understand the context of student projects. Student projects work in a specific time frame and with a particular goal, and once the project is finished, there is a lack of maintenance. For software projects to be long-lived, it is relevant that they are continuously maintained to function with future versions surrounding environments and to get regular updates.

2 UBI-Interact

To understand the choices behind specific implementation details and system design of the python-node-v2, it is necessary to know how the UBI-Interact framework works. Therefore, the following part of the thesis will give a detailed overview of the UBI-Interact framework.

2.1 Network Structure

In the introduction, the general structure of the network of master and client nodes, where the master node directs communication between the client nodes, was already briefly mentioned. This design shows many parallels to the server-client design, but the master node does not only answer requests but can also directly send data to clients. The client nodes can generate data and send this to the master node, which is then transmitted to other clients that request this data. They can also request the master node to forward particular data sent from other client nodes.

2.2 Communication Protocol

There are two types of messages that are sent between the nodes. On the one hand, there is the data that contains the payload, which is produced or consumed by the client nodes. On the other hand, there are messages to request services at the master node, which are used to control the communication flow and the state of the network.

For the messages containing the payload, the communication is handled with topics and a publish-subscribe mechanism. A topic is an identifier that is used to mark data's affiliation to a specific group of data messages. For a client node to receive data, it needs to be subscribed to a topic, and whenever a client node sends data to the master node, the master node will send this data to all nodes that are subscribed to the topic that the data belongs to. Accordingly, publishing data means generating data for a certain topic and sending it to the master node, which will then send it to the subscribed clients. This data always has the form of a TopicData Object, which, if it does not contain an error message, contains either a TopicDataRecord or a TopicDataRecordList, which are multiple TopicDataRecords. A TopicDataRecord can contain the topic it belongs to, a timestamp, the id of the client that sent it, and the payload, which is exactly one piece of information that can be of different types, for example, a bool value, a string, an integer or many other options.[1]

The other type of messages follow the classic server-client approach, where the client node sends a request to the master node, and the master node answers. The request is of the type ServiceRequest, which contains a string that defines the service that is requested and one datatype if required by the specific service. The answer from the master node comes in the form of a ServiceReply, which contains exactly one object of a specific type defined by the request. There are a number of services the UBII framework offers, one example that was already mentioned is the topicSubscription, which makes the master node forward incoming TopicData for a certain topic to the client node that made the service request.

2.3 Message types

The last chapter explained the two types of messages in the framework and the way they are transmitted. However, a ServiceRequest or a TopicData object cannot be sent directly over the network but has to be serialized. This means encoding it into a format that can be sent over the network and that can be decoded by the receiver back to the original object to then work with the data. For serialization and deserialization, UBII uses Protocol Buffers. Protocol Buffers allow a developer to define data structures in the protocol buffer language, which in UBII is proto3. To work with these definitions, the proto compiler can be used to compile the proto files into language-specific files that allow for instantiation, serialization and deserialization of objects of the defined types in certain programming languages. Using protocol buffers therefore requires a proto message definition for all data types sent over the network. The framework defines several proto message definitions that are required to be used for the network to function in the intended way. Some of these data structures were already mentioned as TopicData, TopicDataRecord,TopicDataRecordList, ServiceRequest and ServiceReply, but for the framework to function properly, there is a lot more information that needs to be transmitted. These informations often contain the state of parts of the system or configuration details and can be sent between the nodes as part of the already mentioned types. [2][3]

2.4 Master Node

The past chapters sufficiently explained the role of the master node as a central component that directs communication between the client nodes. To do so, the master node offers 5 endpoints for the clients to connect to.

For service requests, the master node offers one REST server for messages sent in the JSON format and one for messages in a binary format encoded via protocol buffers. Depending on the configuration that the user can change, those REST servers can be accessed with HTTP or HTTPS requests. The client nodes can also use ZeroMQ sockets with the request-reply pattern to perform service requests.

To send and receive topic data, the master node offers two endpoints, a websocket, and a ZeroMQ socket. Both require the topic data to be serialized into a binary format with the proto files.[1]

2.5 Client Node

To use all the functionality the UBII framework offers, the client node has to connect with the master node for both the service requests and the topic data. However, which of the options provided by the master node is chosen is up to the developer. To initialize a client node in the UBII network, it has to send a registration service request to the master node, which after successful registration, will answer with configuration informations about the client node. For a client node to keep an “active” state, a ping-pong protocol has to be followed where the master node regularly sends a “PING” message with the topic data endpoint and the client has to answer with a “PONG” message to not be marked as inactive.

3 Concurrency

In the chapter about UBI-Interact, it was established that the UBII framework works as a network. One typical problem that always comes with networking in software projects is the time a message needs to be sent over a network connection. This problem is especially important for UBII since it is a framework made for communication and sends a lot of messages. If all method calls inside the python-node-v2 were synchronous, this would lead to a bad performance of the node. The reason for this is that every time a message is sent to the master node, the execution of the program is delayed by this instruction until it is done transmitting. This would also significantly impact the performance of the program that uses the python-node-v2 since all the code around the framework would be delayed whenever a service request is made or a topic data sent.

The general idea for dealing with this problem is to use the time during this delay to execute different instructions instead of actively waiting until the sending process is finished. This type of programming is called asynchronous or concurrent. To write concurrent code, different languages offer different options. In the following, two popular approaches that I decided to use for the project will be explained and discussed what is best suited.

3.1 Asyncio

Asyncio is a python library that allows the user to write asynchronous code as a solution to the problem of time-intensive IO operations. To be asynchronous, the library uses an approach where the time that a method waits for an operation, that does not need any CPU resources, to finish is used to execute different tasks. In the case of the python-node-v2 this waiting time is the delay when sending messages over the network. To implement this behavior asyncio mainly uses the following three concepts:

- Tasks
- Coroutines
- Event loops

3.1.1 Coroutine

A coroutine can be thought of as a method that can be paused during its execution and be continued at a later point in time. This functionality is used to pause the execution during the already mentioned time intensive waiting time that comes from the network communication. Once this operation is done, the method can then be continued with the remaining nonblocking code. During this pause, other instructions can be executed, this can either be nonblocking synchronous code or different coroutines that then again can be paused.

To work with coroutines, asyncio offers two crucial keywords, “async” and “await”. The async keyword is used in the definition of a function to define it as coroutine. The await keyword is used to pause the coroutine. Normally, the await keyword is followed by a different coroutine. When the await keyword is used, the current coroutine will be paused until the awaited code is finished.[4]
Source Code 3.1 is an example of a coroutine.

Listing 3.1: Coroutine

```python
async def waitForDelay(delay):
  await asyncio.sleep(delay)
  print("Hello World")
```

In this code example, the async keyword is used to define the coroutine waitForDelay. This coroutine uses the await keyword to wait for delay seconds before printing “Hello World”. If waitForDelay is called with the await keyword, its execution pauses at the sleep instruction to allow other code to be run, and once the delay is over it returns and prints.

3.1.2 Task

Tasks are objects that wrap coroutines. When tasks are executed, the coroutine they wrap is run concurrently. The difference in functionality between tasks and coroutines is that coroutines are called with the await keyword, which pauses the current coroutine until the awaited one is finished. Tasks run the wrapped coroutine concurrently, however they do not pause the execution of the environment they are called in.[4]
Source Code 3.2 is an example with a task.

Listing 3.2: Task

```python
def printInstantly(delay):
  asyncio.create_task(asyncio.sleep(delay))
  print("Hello World")
```

This code fragment looks similar to the one above, with the difference that it does not define a coroutine but a method and instead of awaiting the sleep coroutine we wrap it in a task and run it. If printInstantly is called, it starts the task with the coroutine, which is run concurrently, and will instantly, without pausing the execution of the method, print “Hello World”. After the printInstantly method is finished, the sleep coroutine is still executed.

3.1.3 Event Loop

With coroutines and tasks explained, there is the question of how this form of concurrency is scheduled, how it is decided when a function gets paused and what other code runs in the waiting time? This work is performed by the event loop. The event loop is a loop that iterates over a queue of tasks. From this queue, the event loop runs one task

at a time until the coroutine wrapped in this task is at the point where it is waiting. This point is signaled by the await keyword. Once this point is reached, the coroutine gives the control back to the event loop, which will start a different task, one that is not waiting. Whenever the event loop chooses a task to run, it checks if a task that was waiting is ready to continue and can allow this task to continue running with the nonblocking synchronous code. [4]
Source Code 3.3 is an example for an event loop:

Listing 3.3: Event Loop

```
async def printAfterDelay(message, delay):
  await asyncio.sleep(delay)
  print(message)

async def main():
  await asyncio.gather(printAfterDelay(1,"Hello World 1"),
  printAfterDelay(2,"Hello World 2"),
  printAfterDelay(3,"Hello World 3"))

loop = asyncio.new_event_loop()
asyncio.set_event_loop(loop)
res = loop.run_until_complete(main())
loop.close()
```

This code is an example of running multiple coroutines concurrently. The coroutine printAfterDelay prints a given string after a given delay is awaited. In the main coroutine, the gather method is used to schedule the three coroutines to be run concurrently as tasks by the event loop. To execute the main coroutine, a new event loop is created and set as the current loop for this thread, and the run_until_complete method is used to schedule the main coroutine as a task.

One important detail to mention is that while the main coroutine and also the three printAfterDelay coroutines run concurrently the event loop does not. This means that using an event loop is blocking, and all instructions that follow this code snippet would wait until all tasks in the event loop are finished.

3.2 Threading

It was shown that asyncio is one option to write concurrent code. In this chapter, the threading module will be introduced as a second way to do so. To understand how threading works, it is important to know the following concepts and how they work:

- Processes
- Threads
- Threading
- Global Interpreter Lock

3.2.1 Process

A process is a program that is executed by the operating system on the CPU. It is not directly associated with a programming language or a specific python implementation but is typically linked to the operating system. A process is also assigned its own hardware resources like memory or CPU computing time. An example of a process would be the execution of a python file.[4]

3.2.2 Thread

Similar to a process, a thread is a number of instructions run on the CPU. Threads are created by processes and share resources like memory with them. A process can also create multiple threads that run concurrently.

Most people in the field of computer science have worked with threads or at least heard of them because many programming languages offer multithreading as a form of concurrency. Threads are often linked to a special form of concurrency called parallelism. Concurrency was described as tasks running simultaneously, meaning that at one point in time, multiple tasks are in a state where they have been started and are not finished. Parallelism, on the other hand, describes multiple tasks being executed at the same moment. This is achieved through the use of multiple CPU cores where different tasks run on different cores at the same time. [4]

3.2.3 Threading

Python also offers a number of libraries that allow for multithreading, one being the threading module from the standard library. This module can be used to create threads and run multiple threads concurrently. The simplest form to use threading is to create a thread with a function that should be executed by it and start the thread.
Source Code 3.4 is an example for a thread:

Listing 3.4: Threading

```python
def helloWorld():
  print('Hello World')

thread = Thread(target=helloWorld, args=[])
```

```
thread.start()
```

This code shows how the threading module could be used to create a thread that runs source code, in this case the helloWorld method.

Different from the way many programming languages offer multithreading for parallelism, the threading module does not. When threading is used to create and run multiple threads those are executed concurrently but not parallel. This is due to the Global Interpreter Lock.

3.2.4 Global Interpreter Lock

As mentioned, the Global Interpreter Lock or GIL prevents parallelism when multithreading is used. This is because, more generally said, the GIL prevents a process from running more than one python bytecode instruction at once. Since threads are associated with a process, they are affected by this behavior. The reason the GIL exists is because of the way python manages and cleans up memory.

With this explanation of the GIL, there is the question of the use cases for threading if it cannot be used to run threads in parallel. One general use case of multithreading that the threading module cannot be efficiently used for is the utilization of multiple CPU cores to optimize calculations that need lots of computation power. For this kind of problem, a multiprocessing approach would be necessary in python. However, threading can be used for the same use cases in which the asyncio module is used, when there is code that comes with a lot of waiting time for IO operations. In this case, the scheduling of the threads can be utilized to give computation time to the threads that are not waiting. This works because IO operations like sending a message over a network connection release the GIL since these are low-level operating system calls that do not run python bytecode.[4]

3.3 Concurrency in the Python-Node-v2

At the beginning of this chapter, it was explained that a form of concurrency is essential for the python-node-v2 because of performance reasons. In the past chapters, the functionality of the asyncio and the threading module were explained. Both modules offer concurrency in a similar way where the waiting time that comes from the network communication could be used to execute different sourcecode. With asyncio, this would happen by the use of an event loop and the blocking functionality would be inside coroutines, that would be scheduled by the event loop. With threading, the blocking operations would run in their own thread, which the operation system would schedule.

At the start of this project, I used the asyncio approach to develop the node in an asynchronous way. The way asyncio handled concurrency solves the exact problems the system faces. Asyncio also seemed more intuitive than threading without the need to create threads and with the async and await keywords already known from different programming languages. However, there are some points that could be problematic when using asyncio.

In general, asyncio depends on code that encapsulates blocking code in coroutines by making use of the await and async keywords. This requirement could lead to problems when using third party libraries for blocking operations. To use those libraries with asyncio they would have to support asyncio and already put those operations inside coroutines that can be awaited. However, this was not a problem for this project since all the functionality needed was networking related, and with websockets and http(s) requests, the aiohttp library offers all necessary connection methods to connect with the master node with support for asyncio.

A different problem that could come with asyncio is that the coroutines must be run as tasks with an event loop. This means that the entry point of a program, which normally is the main function, would need to create an event loop, with the actual code as a coroutine that would be passed to the loop. This would negatively impact the usability of the python node v2, since the user would need to know how to use asyncio and create this event loop. This would also block the main function because, as mentioned in chapter 3.1.3, running the event loop is blocking. So this approach would force the user to integrate asyncio into his project because coroutines can only be called with an event loop or with await, which makes the calling function a coroutine as well.

Because usability is a high priority for frameworks, this made me look into other options like threading. Threading solves all the usability issues since there would not be any coroutines but only normal methods that could be called. This reason made me go with the threading approach in this project. However, there are also some problems with threading that are worth mentioning.

If the user has to wait for a blocking operation to finish because it is relevant for his use case, this functionality was given with asyncio since the called coroutine could just be awaited. With threading, this functionality is more complicated to implement and therefore is only offered for a few operations. Ways to implement this could be with a return value of the blocking function, for example, by returning the executing thread so it can be joined.
Another problem is the sharing of resources between threads, which could create potential race conditions that require extra synchronization.

One topic that has yet to be explained is return values. So far, concurrency has been reviewed in the context of sending messages over the network. However, when making service requests, a service reply is answered by the master node. With the offered method to make service requests, it is expected that this method will return the service reply. With asyncio, this would be possible by awaiting the coroutine and then returning the reply. To get the same functionality with threading a concept called futures is used. A future object is an object that holds a value that is expected to be set in the future. This object can be in different states, such as running, done, or exception. When making a service request, a future object is returned, and the user has the option to wait for the future to finish and then work with its value, which is the service reply.

4 UBII-Python-Node-v2

In the following part of the thesis, the implementation of the python-node-v2 will be analysed. For this, an overview of the general structure of the code will be given and each module will be explained in detail with its function inside the client node and with relevant implementation details.

4.1 Structure of the Python-Node-v2

Besides the functionality's implementation, the system design is an essential aspect of every software project. A good system design is important for several aspects, like keeping the code maintainable if changes are necessary or new functionality is implemented.

The system design of the ubii-python-node-v2 is adopted from the ubii-node-unity3D[5]. This is because I used the unity node as an example of a client node to understand how UBII works and what functionality a client node has to provide. The node follows an object-oriented design where the functionality is implemented inside classes. To use this functionality, objects of certain classes need to be instantiated and then used. The functionality of the node can be grouped into different scopes. This was used to implement these scopes in their own class. This led to the following six classes:

- UbiiClientNode
- TopicDataProxy
- UbiiNetworkClient
- TopicDataBuffer
- UbiiTopicDataClient
- UbiiServiceClient

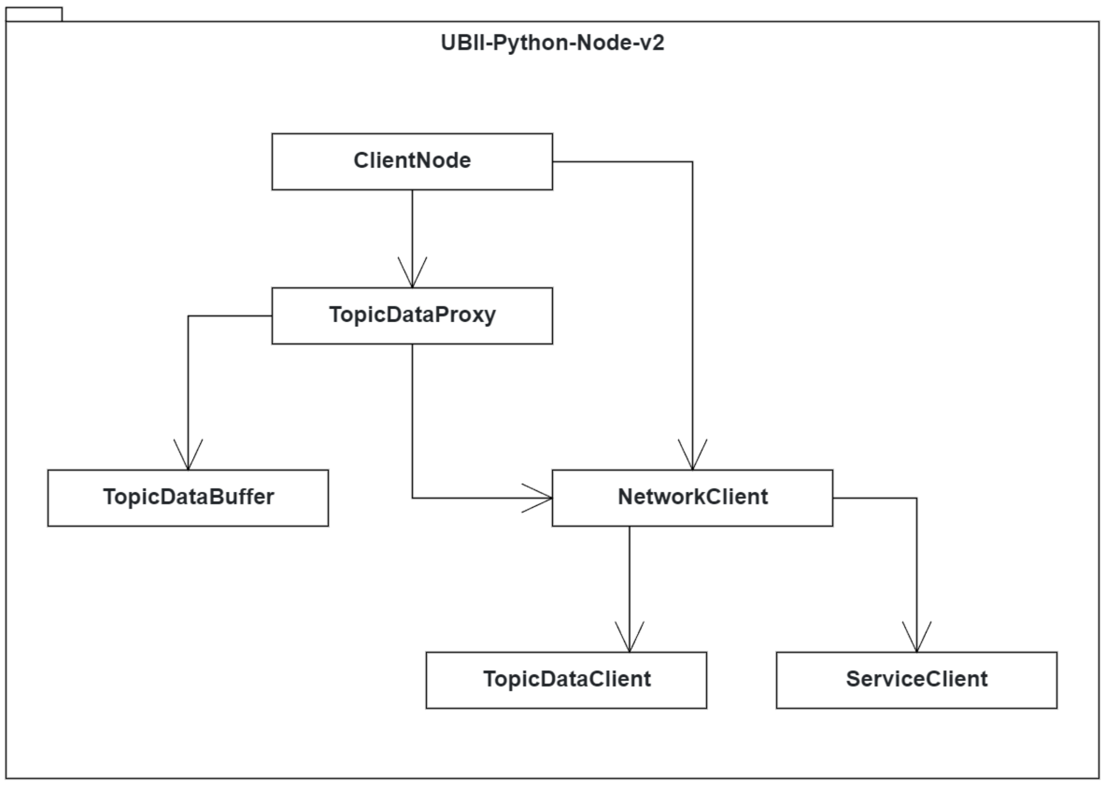

Figure 4.1: UML class diagram with associations beween the classes

This UML diagram shows the associations between the different classes. This is not completely accurate since, in the actual code, some of the associations go in both directions, but it shows the logical delegation hierarchy. This delegation hierarchy works in a way where the client node is the entry point to the system. When it is used to perform some of the functionality the framework offers, it delegates it to the other classes, which can continue delegating it until the responsible class performs the action. In the following chapters, each class or module containing the class will be looked at, and their different functions will be explained.

4.2 UbiiClientNode

The UbiiClientNode class does not implement any functionality but provides the user with an interface that offers the UBII functionality through different methods. Those methods call either the TopicDataProxy or the NetworkClient and delegate the work to those classes. To use those methods, the user can instantiate a ClientNode object with a name and endpoints for topic data and service requests. The following methods are provided to use the UBII functionality:

4.2.1 Interface

Method: call_service

Description: This method is used to make service requests. It returns a future that contains the service reply.

Method: publish

Description: The publish method is used to send TopicData to the master node. The way publishing works is that every TopicDataRecord that is published with this method is stored. The content of this storage is then sent to the master node at a fixed interval. When multiple TopicDataRecords for the same topic are published in the same interval, the one that exists in the storage is overwritten, and only the newest one is sent.

Method: publishList

Description: Takes a TopicDataRecordList and publishes every element of it.

Method: publishImmediately

Description: This method can be used to send a TopicDataRecord directly to the master node without waiting for the next publish interval.

Method: subscribeTopic

Description: This method is used to subscribe to a topic. Besides the topic that is subscribed to, the method takes a callback as a parameter. This callback takes one TopicDataRecord and returns nothing. Whenever the master node sends a TopicData that belongs to this topic to this client node, the callback is invoked with the sent TopicDataRecords. This callback is how the user can work with incoming data published by different clients.
This method returns a SubscriptionToken, which is an object that can be used to unsubscribe a topic.

Method: subscribeRegex

Description: This method works similarly to subscribeTopic but subscribes to a regex. Whenever a TopicData is published to the master node where the topic matches this regex, it will be sent to this node.

Method: unsubscribe

Description: This method unsubscribes a callback for a topic or a regex, which callback is defined by the SubscriptionToken that was returned when it was subscribed. This does not necessarily mean that the node no longer receives TopicData for the topic or regex from the master node, this only happens when there are no more callbacks subscribed to a topic or regex.

4.2.2 Initialization

Other than offering the described methods, the ClientNode class also starts the node's initialization. This includes making a service request with the master node to register this node and get relevant client information. Since this has to happen concurrently, it could lead to methods of the node being called before it has initialized. If this happens, the thread where the method call is executed will wait until the initialization is finished. To do this, the client node uses an event that can be used to wait in the current thread and that wakes every waiting thread up when it is set.

4.2.3 Error handling

When programming, it is important to deal with potential errors and exceptions. One kind of exception that could occur in the UBII framework is connection errors when the network connection between the master and client node gets interrupted or cannot be established. If those exceptions were just thrown, the program where the node is used would stop when such an exception occurs. This would force the user of the python-node-v2 to handle exceptions. For usability of the system this would not be ideal since the user would need to read all the networking code to see what libraries are used and what exceptions can be thrown.

To deal with those exceptions, the ClientNode class offers an Events object that contains four events to which the user can subscribe. Events can be subscribed to with methods, and whenever an event is called, all methods that are subscribed to this event will be executed. Those events are defined for the interaction methods with the master node and general connection errors. Whenever an error occurs during the interaction, the fitting event is called. This allows the user to decide how an error is handled without knowing the details of each exception.

4.3 TopicDataBuffer

The TopicDataBuffer class manages the local topic and regex subscriptions and the corresponding callbacks. It is also responsible for generating a SubscriptionToken whenever the user uses one of the subscription methods. A SubscriptionToken contains all relevant information for one subscription: the unique id of the token, the topic or regex that is subscribed to, the callback that is invoked whenever an according TopicData is received and if it was a topic or regex subscription.

The most basic function of the TopicDataBuffer is to execute all necessary callbacks, passed when subscribing to a topic, whenever the node receives TopicData. To do this, the TopicDataBuffer uses four dictionaries to store the required data. A dictionary is a container that stores key-value pairs. These dictionaries are one with all the topics and the most recently received TopicDataRecord, one with the topics and lists with callbacks that belong to a certain topic, one with the regexes and lists with according callbacks, and one with topics and lists of regexes that the topic matches. To function correctly, the buffer has to keep those dictionaries updated whenever a topic or regex is subscribed or unsubscribed.

Those actions on the dictionaries greatly influence the node's performance. Because they are CPU-bound, they do not have any waiting time that could profit from multithreading and are, therefore, executed sequentially. Because of this, the choice of collection to store the relevant information should be made with a focus on performance. In the regular use of the node, the most performed action is the lookup of whether the collection contains an element and the access to this element. While those operations are performed whenever a TopicData is received and when a topic or regex is subscribed or unsubscribed, adding or removing an element is less performed since it is mostly done when subscribing or unsubscribing. For all of those operations, dictionaries have an average time complexity of O(1) and an amortized worst-case time complexity of O(n). Alternative data structures would be sets or lists. Sets, however, have very similar time complexities to dictionaries. Since the average time complexity of the lookup, if a list contains a certain element is O(n) and therefore is a lot worse than in the dictionary, the dictionary is the preferable data structure from the point of performance.[6]

4.4 UbiiNetworkClient

The UbiiNetworkClient class manages every functionality that works over the network. This means every communication with the master node. It does, however, not connect to the master node directly and send messages, but it delegates this work to either the UbiiTopicDataClient or the UbiiServiceClient. Besides the delegation process, the UbiiNetworkClient performs two types of workloads.

Since the network communication performed by the UbiiTopicDataClient and UbiiServiceClient is blocking, the UbiiNetworkClient is responsible for creating the threads that execute these methods concurrently. This is done mainly by creating a thread with the requested method and starting it. This means that for every message that is sent, a thread is created that exists for the duration of this operation. For TopicData this is done like in the example 3.4 and for service requests a ThreadPoolExecutor is used. A ThreadPoolExecutor can be used for threadpools, however I decided to use it because it offers a simple way to get a future object where a different thread provides the value. This future object containing the ServiceReply is also the return value of the service request.

The second type of work is the instantiation of some necessary proto message objects, like certain ServiceRequest or TopicData objects, that can then be passed to the methods that send the messages over the network.

4.5 UbiiTopicDataClient

The UbiiTopicDataClient class is responsible for the topic data communication with the master node. As described in chapter 2.4 the master node offers two endpoints to communicate topic data. The python-node-v2 only supports connection with websockets, this connection is established in the UbiiTopicDataClient with the websockets library. This library offers basic websocket functionality like sending or receiving messages with and without support for asyncio, which is not needed in this project because multithreading is used.

The topic communication consists of two parts, publishing and subscribing. The publishing of topic data happens in a fixed interval, and it is undefined when the next TopicData is sent by the master node. Therefore, an approach where a new thread is created every time a message should be sent or received is not working efficiently. Because of this, the UbiiTopicDataClient creates two threads that run for as long as the node is used. One of those threads is responsible for publishing the stored TopicDataRecords in the fixed interval. The other one is responsible for receiving TopicData from the master node whenever necessary as well as performing the "PING"-"PONG" protocol. This thread calls the TopicDataBuffer whenever a TopicData is received to invoke the correct callbacks.

To make sure those threads stop properly when the node is no longer needed, they are stopped when the user calls the stopNode method, which handles the shutdown of the node. They are additionally set as daemon threads, which means they will automatically stop when only daemon threads are running.

4.6 UbiiServiceClient

The UbiiServiceClient class handles service communication with the master node. It supports HTTP post requests, where the transmitted ServiceRequest is encoded in the binary format. These requests are performed with the aiohttp library. Since this library supports asyncio, the calls are coroutines, and the UbiiNetworkClient, therefore, has to create an event loop inside the thread that performs the service request.

4.7 TopicDataProxy

The TopicDataProxy acts as an intermediary object between the UbiiClientNode and the UbiiNetworkClient. There are certain methods of the UbiiClientNode that sometimes require communication with the master node. Those method calls are delegated to the TopicDataProxy. It then determines if communication with the master node is necessary and delegates the work to the UbiiNetworkClient or if the work can be performed locally and then delegates it to the TopicDataBuffer. Sometimes both options are required.

Those methods are the subscribe and unsubscribe methods. When a topic or regex is subscribed, the passed callback should be invoked whenever an according TopicData is received from the master node. However, not every time the subscribeTopic or subscribeRegex method is called, a subscription service request has to be made. When there are already SubscriptionTokens registered for this topic or regex, the node should already be subscribed at the master node and receive the TopicData. When this is the case, the callback only has to be added to a certain topic or regex in the TopicDataBuffer. Only when there is no subscription for a topic or regex the service request has to be made. The other way around, not every call of the unsubscribe method should end in a service call to unsubscribe a topic from the master node. Because there could still be different subscription tokens for the topic or regex that require the TopicData, only when there are no more SubscriptionTokens the service request to unsubscribe from a topic or regex needs to be performed.

5 Distribution

The reference to the python-node-v2 is the github repository, which includes all relevant files as well as instructions on how to use the node. However, using github as the main form of distribution of the system would lead to a lot of difficulties. The project has some dependencies on third party libraries like aiohttp or websockets that would need to be installed by the user. It would also complicate the placement of the project folder inside the user's system, especially in the context of relative imports where the import path depends on the placement and position of the node and the file containing the main function.

To solve these problems, the python-node-v2 was uploaded to PyPi as a package that could be easily installed using pip as package manager. This allows all relevant dependencies to be automatically installed together with the node. Besides third party libraries, the node also depends on the proto message definitions. Those are not included in the package of the node but are uploaded to PyPi as a separate package as well. This allows changing or adding new message definitions without uploading a new version of the node, which is independent of those and does not necessarily need to be changed. Only the compiled files generated for Python were uploaded since those are required to work with the data structures. It also makes it easy for the user to update the message definitions using pip.

6 Testing

Before releasing a software project, testing for correctness is essential. To test the python-node-v2 the unittest module was used. This module offers a number of assertion methods that can be used to compare the actually produced data with the artificially generated expected data. It also provides the functionality to execute multiple test methods sequentially.

With this package, a test case was created for every method that offers the UBII functionality in the UbiiClientNode class. This package is named unittest, and it can be used for unit tests. However, the test cases for this project were integration tests because it is not possible to test one of those methods without using other functionality from the UbiiClientNode and the different modules. If the test for a method passes, it is likely that the functionality in all the used modules down the delegation hierarchy is working correctly as well. The test cases all had a similar design. Multiple nodes were used, one of which performed the tested method and recorded the result to be tested for correctness later. The other nodes performed the actions that were necessary to test a specific behavior. For example, when testing publishing, nodes needed to receive the data to test if everything was published the expected way. It was not only tested if the correct behavior occurred but also the absence of false behavior.

The same tests were conducted for the python-node-v2 modules in the same project folder as the test module and as a single test module using the installed pip package. The tests were carried out on three different Python versions: 3.7, 3.10, and 3.12, both on Windows 10 and Linux (Ubuntu 22.04.3).

7 Future Work

7.1 Features

The python-node-v2 implements the core functionality of the UBII framework. Additional to those features as described in chapter 2 UBII provides further functionality.

Devices and Components

Two data structures that are defined in the proto message format that were not mentioned so far are Device and Component. Besides config information components contain information about the type of topic data that the node interacts with. This includes the format and the publish-subscribe behavior. Devices contain these components. While the communication with the master node is handled by the client node, the client node can register these devices at the masternode through provided services.[1][7]

Processing Modules and Sessions

Processing modules can be used to transform data outside of the client node. The transformations they offer can differ in functionality, and input and output formats for the data can be defined by each processing module accordingly. Processing modules are executed inside sessions that offer a particular execution environment.[1][7]

Connection Endpoints

To make the node more versatile, an implementation for the different endpoints that are not supported yet could be provided. This would include ZeroMQ connection as well as ServiceRequest formats encoded in the JSON format. That would allow the user to choose the optimal option for each usage of the node.

7.2 Maintenance

Compatibility

The python-node-v2 was tested for python 3.7, 3.10 and 3.12 on windows and linux. This testing can be extended to different runtime environments. This would guarantee compatibility for different versions of these environments and would make the node available for a broader range of projects. This could include different versions of the node with adaptations inside the node to fit special python versions.

Optimization

The performance aspect played a very important role in this thesis, especially in terms of concurrency. To solve this, many threads are created throughout the usage of the node. This leads to overhead in the form of the creation process of each thread. This could be optimized in a way where one long running thread is used for multiple service requests instead of creating a thread for each request.

8 Conclusion

UBII is a framework that takes care of communication in distributed systems. This is done with a network of client nodes that produce and consume data and a master node that distributes this data. Multiple ways exist to establish a network connection between the nodes, such as HTTP requests to a REST server and websockets. Once the connection is established, the nodes can communicate data with a publish-subscribe pattern. The choice of communicated data is made via requests between the nodes in a request-response protocol. To share the same data between the different nodes, the framework offers several data structures that define the format of every message. These data structures are defined with protocol buffers, which include their own way of serialization and deserialization.

Network communication like this always poses the problem of time intensive IO operations when sending messages over a network connection. This duration cannot be optimized by the program but has to be waited for. To work around this delay, python offers multiple concurrency models that allow the execution of different code while the message is being transmitted. With asyncio we have an approach where a concept called event loop schedules different tasks so the delay can be used to run other code in this time. The threading module offers an approach where those IO operations are performed in their own threads, and the operating system schedules the execution of those threads. While the asyncio library offers good functionality with coroutines that can be awaited to time different functions correctly, the threading module offers better usability for the end users of this framework since they do not have to deal with a third-party library.

To implement all the referred functionality for the python-node-v2 a delegation model is used. The ClientNode class provides the user with the functionality of the node through a number of methods. When such a method is called, the work is delegated to a proxy that determines if the work can be performed locally or if communication with the master node is required. The TopicDataBuffer then performs the local work, and networking related requests are delegated to the NetworkClient. The NetworkClient then creates the necessary objects for the communication and delegates the sending process to the TopicDataClient or ServiceClient in its own thread.

To use the offered functionality in a project the python-node-v2 can be installed with pip and can then be imported and used. This pip package is tested for three python versions on windows and linux.

Listings

Bibliography

[1] Sandro Weber, Marian Ludwig, and Gudrun Klinker. Ubi-interact: A modular approach to connecting systems. *EAI Endorsed Transactions on Mobile Communications and Applications*, 6(19), 7 2021.

[2] Protocol Buffers Documentation. Protocol buffers. `https://protobuf.dev/`, 2024. [Online, accessed 27-February-2024].

[3] Sandro Weber. ubii-msg-formats. `https://github.com/SandroWeber/ubii-msg-formats`, 2024. [Online, accessed 27-February-2024].

[4] Matthew Fowler. *Python Concurrency with asyncio*. Manning Publications, 2022.

[5] Sandro Weber. ubii-node-unity3d. `https://github.com/SandroWeber/ubii-node-unity3D`.

[6] wiki.python. Timecomplexity. `https://wiki.python.org/moin/TimeComplexity`, 2023. [Online, accessed 02-March-2024].

[7] Sandro Weber. ubi-interact overview. `https://github.com/SandroWeber/ubi-interact/wiki/Overview`, 2022. [Online, accessed 21-February-2024].