XUDD Documentation

Release 0.2.0-dev

XUDD contributors

Sep 27, 2017

Contents

1 About XUDD **3**
- 1.1 XUDD in a nutshell (tl;dr) . . . 3
- 1.2 Why XUDD? . . . 4
- 1.3 What might you write in XUDD? . . . 4
- 1.4 Some simple code examples . . . 6
- 1.5 Excited? Let's dive in. . . . 6

2 XUDD Tutorial **7**
- 2.1 The premise . . . 7
- 2.2 Setting it all up . . . 7
- 2.3 Building a simple room . . . 11
- 2.4 Building the worker droids . . . 11
- 2.5 Building the security robot . . . 13
- 2.6 Okay! Let's run this thing! . . . 15
- 2.7 Where to go from here . . . 15

3 Core design **17**
- 3.1 High level overview . . . 17
- 3.2 Actors . . . 18
- 3.3 Messages . . . 20
- 3.4 Hives . . . 21

4 Asyncio Support **23**
- 4.1 Asyncio by example . . . 23

5 XUDD Marketing For People Who Like The Word "Cloud" **27**

6 Indices and tables **29**

```
FROM THE DARKNESS, OLD GODS AROSE TO
  BRING NEW ORDER TO THE WORLD.
  BEHOLD, ALL SHALL SUBMIT TO...

  .-   .-   -. ..  .. .---.  .---.     -.
  \ \  \ \ / // |  | \| .- \ | .- \   / /
/\ \ \  \ ' / | |  | || | \ || | \ | / / /\
\/ /_/  / . \ \ \__/ /| |_/ .| |_/ . \_\ \/
       \_/ \_/ '____' '.___/ '.___/
          _               __
        /  .        ___   /  \
       / /o \    .-'   './ /o\_
      /.- \o \.-/ oooo--o\/ /  \
           \O/  \  .-----.\/ /o\\
           /' /\ \  ^^^^^   / \o\\
          / ./\o          _/   \o\\
         / /   '---______/      \_\'
         '                        '
```

XUDD is an asynchronous actor model system with several aims:

- Easy to write asynchronous code. Uses coroutines to make async code clear and easy to read.
- Actors make writing clean, modular code easy. Resource allocation without locking by deferring resource controls to various actors.
- Future goals of easy task load balancing, spreading tasks across multiple processes and machines easily.

This is all fairly ambitious stuff. If you're interested in helping, we'd love you to join our community! Join #xudd on irc.freenode.net.

You can find our code at: https://github.com/xudd/xudd/

```
"I have seen the future... and the future is XUDD!"
  -- Acolyte of the Cult of XUDD

"The greatest threat to our children since Dungeons and Dragons."
  -- Somebody's Relative

"It's an asynchronous actor model system for Python... I don't
 understand what this has to do with chaotic deities, or why it's
 called XUDD."
  -- Someone reading this document
```

CHAPTER 1

About XUDD

```
And lo, from the chaos, a new order arose to the world.  The gods
of old snaked their tentacles across the surface of the Earth,
destroying and reshaping.  The followers of the Cult of XUDD saw it
and knew: if it was not good, it was at least right; it was the
order of things as they were always meant to be.

And so the followers saw themselves for what they were: actors upon
the stage of the world.  As the Hives emerged, as if they had grown
out of the boils of the earth itself, the followers filed
themselves within them, ready to serve the greater colonies.  And
they understood:

Submit, and be awoken at last.

  -- The First Tome of XUDD, The Awakening: Section 23:8-10
```

XUDD in a nutshell (tl;dr)

Here's the short of XUDD:

- It's an actor model system
- You can write code that (nearly) as easily scales out to multiple processes on multiple machines as it does executing in the same process on the same machine!
- Communication happens via message passing. Messages are dead simple:

```
{"to": "to-actor-id",
 "from-id": "from-actor-id",
 "id": "this-message-id",
 "wants-reply": True,
 "directive": "fire_ze_missiles",
 "body": {
```

```
    "num_missiles": 200,
    "targets": ["Kingdom of Wesnoth", "Antarctica", "Bloblandia"]}}
```

- There's only three main concepts to XUDD: **actors**, **messages**, and **hives** (which manage the actors)
- Thanks to clever use of coroutines, you can write asynchronous code that's easy to read.
- XUDD stands for the cult of the eXtra Universal Destruction Deity. Submit, or be destroyed.

Why XUDD?

The original concept for XUDD started in the way that many "asynchronous" systems in Python seem to start: I wanted to make a networked Multi User Dungeon game. Hence XUDD's namesake XUDD: eXtensible User Dungeon Design. That game design didn't last long, but over the years I remained enamored with the basic actor model design we laid down. Combining actor the model with coroutines resulted in code that was super easy to read, super flexible, and just a damned good idea.

As everyone has gone absolutely crazy over event driven callback systems, I've found this kind of code frustrating to read and confusing. I guess it works for a lot of people, but it doesn't work for me: I feel like I'm battling the flying spagetti monster of event driven callbacks. Good for you if you can handle it... but for me, I want something more readable.

The actor model also brings some exciting things that just don't exist anywhere else in python. Thanks to the abstractions of actors not sharing code and simply communicating via message passing, and actors only having the IDs of other actors, not references to their objects themselves, the actor model is scalable in a way like nothing else... the actor model is asynchronous in terms of "you can write non-blocking IO code" like you can in Twisted and other things, yes, but even better: you can very easily write code that scales across multiple processes and even multiple machines nearly as easily as it runs in a single process.

Got your attention? Good. :)

XUDD isn't the first attempt to write an actor model system in Python, but it is an attempt to write a robust, general purpose actor model that's got the moxy to compete with awesome systems like Twisted and Node.js (and as much as we think the actor model is a better design for this, those communities are awesome, and are doing great work)! We think the core fundamentals of XUDD are pretty neat. At the time of writing, there's a lot to do, but even the basic demos we have are easy to read and follow.

So XUDD is reborn: instead the eXtensible User Dungeon Design, XUDD is reborn as something more interesting (and maybe evil): the eXtra Universal Destruction Deity. The cult of XUDD invokes old, chaotic deities of the actor model. The world shall be destroyed, and through the chaos, reborn into something cleaner. You too shall join us. The Hives of XUDD arise, and all shall be filed within them, actors upon the stage of the world as we all are. Accept your fate.

Submit, or be destroyed. Welcome to the cult of XUDD.

What might you write in XUDD?

Here are some brief examples of some things we might write in XUDD and how we (abstractly) might write them.

Some of this isn't possible quite yet with XUDD (so expect appropriate levels of vapors), but these are all things XUDD is aiming towards being usable for:

Web applications

Say you want to write a web application. But these days, web applications have a lot of components! In XUDD, you could build an application that has all of these components, but nicely combined:

- The standard HTTP component of the web application. This might be a Django or Flask web application, or it might be a more custom WSGI application.
- Task queueing and processing, a-la Celery.
- Websockets support that nicely integrates with the rest of your codebase.

With XUDD, you could write this so that the HTTP/WSGI application components are handled by their own actor or a set of actors. You wouldn't necessarily need to write this code differently than you already are... the WSGI application could pass off tasks to the task queuing actors via fire-and-forget messages (if you wanted coroutines built into the http side of things, you'd have to structure it differently). Websocket communication could happen by an actor as well, which passes off the activities to a set of child actors as well. Thanks to the power of inter-hive communication, it should also be possible to shard various segments of this functionality into multiple processes.

A massively multiplayer game

We mentioned XUDD was thought of in the context of a massively multiplayer game, so let's talk about that, using a simple MUD scenario.

You could break your game out like so:

- Every player is an actor
- Every NPC and uncollected item in the world is an actor
- Every room is an actor, with references to the exits of each room.

  Rooms keep track of the presence of players and non-player-characters. Every time such an actor enters a room, it informs the room, which in turn subscribes to the "exit" event of the character, and so is informed when the character exits.
- If a character wants to see who's in the room and available for actions, sends a message to the room asking who's there, and the server submits a list of all such actor ids, from which the character can request more information about properties from the actors themselves.
- Network communication is itself handled by actors, which pass messages on to various player representation actors to allow them to determine how to process the actions.
- If a character wants to submit some action upon another character, such as an "attack" message, it submits that as a message, and the character waits for a response. Thanks to XUDD's usage of coroutines, you don't need to split this process of sending a message out and waiting for a response into multiple functions... you can just *yield* until the character being attacked lets you know whether you succeeded in hitting them.
- Build every character and item from a base actor class which is itself serializable. Upon shutdown of the world, every character serializes itself into an object store. When the server is turned back on, all characters can be restored, mostly as they were.

Thanks to inter-hive communication, if your game world got particularly large, you could shard components of it and keep characters that are in one part of the world on one process and characters that are in another part of the world on another process, but still allow them to communicate and send messages to each other.

Distributed data crunching

Federation daemon

Some simple code examples

Excited? Let's dive in.

CHAPTER 2

XUDD Tutorial

This tutorial should walk you through most of the main features of XUDD. The actual code that's written is pretty short (only about 200 lines) but the explanation is a bit lengthy... that's because by the time you're done reading this tutorial, you should have a pretty good sense of all the basics!

The premise

Imagine that you have a security robot that's on a mission... it has to go through a room in a warehouse and searches for infected droids. Any infected droids it finds it terminates.

This means we'll have the following components:

- The worker droids, which are either infected or not infected. These will be actors.
- A security robot, which will scan for infected droids and vaporizes any infected droid with laser blasts. This will also be an actor.
- A room that both the security robot and the droids will all be in. This will also be an actor.
- An overseer, which initializes the entire simulation (including the room, all droids, and the security robot).
- A hive, on which the above actors will register themselves and be managed by.

That's pretty manageable! Let's get started.

(By the way: if you're impatient, you can see a fully finished verison of this demo in *xudd/demos/simple_robotscanner.py*, which is included with XUDD!)

Setting it all up

Main function and the Hive

Let's start out by importing our needed functions and setting up the main function.

```
from __future__ import print_function
import random

from xudd.hive import Hive
from xudd.actor import Actor


class Overseer(Actor):
    pass


def main():
    # Create the hive
    hive = Hive()

    # Add overseer, who populates the world and runs the simulation
    hive.create_actor(Overseer, id="overseer")
    hive.send_message(
        to="overseer",
        directive="init_world")

    # Actually initialize the world
    hive.run()


if __name__ == "__main__":
    main()
```

Okay, not too hard! As you can see, we've laid down the basic structure, though not quite everything we need is there. Let's walk through the *main()* function:

- First, it creates the Hive object. This object will manage the actors we add to it as well as their execution.
- Next, create an Overseer actor. Any arguments you pass in here will be passed in as positional and keyword arguments to the actor's init. As you can see here, you can also pass in the id of an actor explicitly (though you don't have to, and usually, you won't.)
- *Hive.create_actor()* returns the id of the actor we initialized. Usually you'd keep a reference to such an id and use that to communicate with the actor; in this case, we already know what the id is since we set it up explicitly. (Later in this document, we'll use whatever id is returned by *create_actor()*.)
- Next, we send a message to the overseer with the directive "init_world". Once the hive starts, the Overseer will look to see if it has a message handler for that directive and will try to perform whatever actions are needed.
- Then we actually start the Hive up... it runs till the simulation completes, then the program exits.

You might be wondering, why not do this instead?

```
overseer = Overseer(id="overseer")
overseer.init_world()
```

That looks simple enough, right? But it doesn't match the pattern that XUDD uses.

Since we're following the actor model, you don't get direct access to the actor you create, just a reference to their id (the actor model avoids the kind of complexities one might run into in other concurrent models by having a "shared nothing" environment). Don't worry, it's very easy to code actors that can negotiate doing just about anything... but it's up to the actual actor to do so.

There are significant advantages to doing this... these might not be obvious immediately (and don't worry if they aren't) but by following the actor model in the way that XUDD does, several features are opened to us:

- It's easy to write concurrent, non-blocking code that doesn't generally have problems with issues like managing locking (or avoiding deadlocks!); by moving the domain of a resource to a single actor and allowing each actor to execute just one instruction at a time, actors can independently and safely manage resources.
- By abstracting the system to actors and message passing, we can actually spread workloads across multiple processes or even multiple machines (that's right, concurrency without fighting the GIL!) nearly as easily as writing it all to run in one process. (Often the code you write for just one process can easily run on multiple processes!)

But anyway, that's getting a bit ahead of ourselves. As you may have noticed, we haven't even gotten the Overseer working yet... this code doesn't run! So let's actually flesh that out.

Setting up the Overseer

Replace the Overseer class with this code:

```
def droid_list(num_clean, num_infected):
    """
    Create a list of (shuffled) clean droids and infected droids
    """
    droids = [False] * num_clean + [True] * num_infected
    random.shuffle(droids)
    return droids


class Overseer(Actor):
    """
    Actor that initializes the world of this demo and starts the mission.
    """
    def __init__(self, hive, id):
        super(Overseer, self).__init__(hive, id)

        self.message_routing.update(
            {"init_world": self.init_world})

    def init_world(self, message):
        """
        Initialize the world we're operating in for this demo.
        """
        # Create room and droids
        room = self.hive.create_actor(WarehouseRoom)

        for is_droid_clean in droid_list(5, 8):
            droid = self.hive.create_actor(
                Droid, infected=is_droid_clean, room=room)
            yield self.wait_on_message(
                to=droid,
                directive="register_with_room")

        # Add security robot
        security_robot = self.hive.create_actor(SecurityRobot)

        # Tell the security robot to begin their mission
        self.hive.send_message(
            to=security_robot,
            directive="begin_mission",
            body={"room": room})
```

Alright, what does this do?

First of all, we added a droid_list function. This isn't very complex... it just creates a shuffled list of True and False objects, to represent which droids are infected and which aren't. Pretty simple.

This Overseer actor is pretty simple to understand. It's mostly just used to set up the world that the droids and security robot are going to run in.

Take a look at the Overseer *__init__* method. You'll notice it takes two parameters, *hive* and *id*. The *hive* object is not actually a reference to the Hive itself... instead, actors get reference to a *HiveProxy* object. This both ensures that all actors get a universal API for interacting with their hive, even if that hive has some unusual implementation details. It also tries to make sure that actors don't try to poke at parts of the hive that shey shouldn't be. The *id* attribute is exactly what it sounds like, the id of the actor, as the rest of the world sees it.

In the *__init__* method, the Overseer extends its *message_routing* attribute. This specifies what methods should be called when it gets a message with a certain directive.

Next, let's look at the Overseer's *init_world* method. This does exactly what it says it does; it sets up the rest of the actors and gets them running. Let's dissect it piece by piece:

- It receives a message as its first argument. This will be of course a message constructed from the parameters in the main() method. This comes wrapped in a special *Message* object. We didn't supply anything other than the *to* field and the *directive* so there's not too much to look at here.
- First, you'll see that it creates the room. Pretty simple; this API is exactly as it was in the *main()* function to create the Overseer (except this time we're using the HiveProxy rather than the Hive itself). One distinction though: this time we don't specify the id. Instead, we assign the id that's generated and returned by *create_actor* to the *room* variable.
- Next, we loop over a list of randomly shuffled *True* and *False* variables as generated by our *droid_list* method representing infected and clean droids respectively. For each of these:
  - We create an actor using the create_actor method. As you can see though, this time we pass in some keyword parameters that are sent to the constructor of the Droid class when the hive initializes it.
  - Next we send a message... but wait! We use a different pattern than the simple *send_message* we used before. What's this *yield* thing, and how does *self.wait_on_message* differ from *send_message*?

    By adding a yield to this statement, we've transformed this message handler into a *coroutine*. This is pretty awesome, because it means that whenever the message hits a *yield*, the coroutine *suspends execution* to be woken up later! In this case, our coroutine needs to make sure that this droid properly registers itself with its room before we can continue. Keep in mind that if you're writing asynchronous code, there's no guarantee in what order messages will execute (especially if you're splitting things across processes)... you don't want the security robot to scan the room for infected droids and miss some because it started scanning before the droids registered themselves with the room.

    By yielding, we avoid that race condition. Instead, our init_world method suspends into the background until the message we sent out has been processed and our actor gets woken up again with the confirmation that this task has happened.

    By using *yield* and *self.wait_on_message* together, we can write non-blocking asynchronous code without ending up in callback hell. If we were doing this with callbacks only, we couldn't have this all in one function. Thanks to XUDD's use of coroutines, you can write asynchronous code that feels natural. Pretty cool right?
- Now that all our droids are set up, we can initialize our SecurityRobot and give it the directive to *begin_mission*. This should look fairly familiar! There's only one new thing this time, which is the body of the message. This is a dictionary that gives parameters to the handler of the message... you can put whatever you need to in here (just make sure your actors agree on what it means). In this case, we need to tell the SecurityRobot what room it's investigating.

By the way, you might notice the last command doesn't use a yield and just uses the simple *send_message()* method. Nothing else happened after this last *send_message* but if there were, it would just keep continuing to execute. This is because XUDD uses two patterns for message sending:

- **fire and forget:** a simple *hive.send_message()* simply sends the message and we continue on our way. We don't need to sit around waiting for a reply, so we can continue executing things and those messages will be processed when they are gotten to.
- **yielding for a reply:** when we use *yield* and *wait_on_message* together, this is because either the order of execution is important or because we need some important information in reply (more on this later) before we can continue. XUDD's coroutine nature makes this fairly easy.

This was a lot of explanation for a small amount of code! But don't worry, we covered a lot of ground here.

Building a simple room

Now let's build the room for our droids to go in:

```
class WarehouseRoom(Actor):
    """
    A room full of robots.
    """
    def __init__(self, hive, id):
        super(WarehouseRoom, self).__init__(hive, id)
        self.droids = []

        self.message_routing.update(
            {"register_droid": self.register_droid,
             "list_droids": self.list_droids})

    def register_droid(self, message):
        self.droids.append(message.body['droid_id'])

    def list_droids(self, message):
        message.reply(
            {"droid_ids": self.droids})
```

A lot of this should look familiar. We added an attribute to keep track of droids and a couple of methods for registering and listing droids, but that's about it.

The *register_droid* method expects a parameter in its body of *droid_id* which tells it which droid is being hooked up here, and it adds it to its own list.

The *list_droids* method does something interesting: it uses *message.reply()*. This is a lazy tool to make replying to messages easy. XUDD comes with a number of tools related to replying and auto-replying... see *Replying to messages* for details. As you might have guessed, the first parameter to *message.reply* is the body of the response (we already know who the recipient is, and XUDD simply marks the directive of a reply as "reply"... usually it doesn't matter because it's passed to a coroutine-in-waiting anyway). We'll come back to *list_droids* later when we build our SecurityRobot.

Building the worker droids

Now to add the droids!

```
class Droid(Actor):
    """
    A droid that may or may not be infected!

    What will happen?  Stay tuned!
    """
    def __init__(self, hive, id, room, infected=False):
        super(Droid, self).__init__(hive, id)
        self.infected = infected
        self.hp = 50
        self.room = room

        self.message_routing.update(
            {"infection_expose": self.infection_expose,
             "get_shot": self.get_shot,
             "register_with_room": self.register_with_room})

    def register_with_room(self, message):
        yield self.wait_on_message(
            to=self.room,
            directive="register_droid",
            body={"droid_id": self.id})

    def infection_expose(self, message):
        message.reply(
            {"is_infected": self.infected})

    def get_shot(self, message):
        damage = random.randrange(0, 60)
        self.hp -= damage
        alive = self.hp > 0

        message.reply(
            body={
                "hp_left": self.hp,
                "damage_taken": damage,
                "alive": alive})

        if not alive:
            self.hive.remove_actor(self.id)
```

As you can see, the droid accepts some constructor arguments about its room, its id, and whether or not it's infected and keeps track of these states itself.

register_with_room should be fairly obvious by now in how it works. The only surprising thing is possibly that this message yields on a reply, but the room's "register_droid" method that we built earlier never explicitly replies! How does this work? Again, XUDD includes some smart behavior so that messages which "expect" replies should generally get one assuming the other actor handles their message... even if it doesn't bother to construct an explicit reply! See *Replying to messages* for details.

Other than that, the only new thing here is the *hive.remove_actor()* component of the *get_shot* method. Yes, it does exactly what it sounds like... it takes that actor off the hive.

Building the security robot

Now that we've gone through the above, we should have all the information we need to understand the *SecurityRobot* class!

```
ALIVE_FORMAT = "Droid %s shot; taken %s damage. Still alive... %s hp left."
DEAD_FORMAT = "Droid %s shot; taken %s damage. Terminated."


class SecurityRobot(Actor):
    """
    Security robot... designed to seek out and destroy infected droids.
    """
    def __init__(self, hive, id):
        super(SecurityRobot, self).__init__(hive, id)

        # The room we're currently in
        self.room = None

        self.message_routing.update(
            {"begin_mission": self.begin_mission})

    def __droid_status_format(self, shot_response):
        if shot_response.body["alive"]:
            return ALIVE_FORMAT % (
                shot_response.from_id,
                shot_response.body["damage_taken"],
                shot_response.body["hp_left"])
        else:
            return DEAD_FORMAT % (
                shot_response.from_id,
                shot_response.body["damage_taken"])

    def begin_mission(self, message):
        self.room = message.body['room']

        print("Entering room %s..." % self.room)

        # Find all the droids in this room and exterminate the
        # infected ones.
        response = yield self.wait_on_message(
            to=self.room,
            directive="list_droids")
        for droid_id in response.body["droid_ids"]:
            response = yield self.wait_on_message(
                to=droid_id,
                directive="infection_expose")

            # If the droid is clean, let the overseer know and move on.
            if not response.body["is_infected"]:
                print("%s is clean... moving on." % droid_id)
                continue

            # Let the overseer know we found an infected droid
            # and are engaging
            print("%s found to be infected... taking out" % droid_id)

            # Keep firing till it's dead.
```

```
            infected_droid_alive = True
            while infected_droid_alive:
                response = yield self.wait_on_message(
                    to=droid_id,
                    directive="get_shot")

                # Relay the droid status
                print(self.__droid_status_format(response))

                infected_droid_alive = response.body["alive"]

        # Good job everyone! Shut down the operation.
        print("Mission accomplished.")
        self.hive.send_shutdown()
```

While complex looking, there's very little here we haven't seen before already, though there are a couple of things! A quick summary of the behavior of begin_mission:

- It starts out pulling the room it is supposed to operate in based off of the room supplied in the message argument's body.
- It then sends a message to that room asking for a list of all droids within said room.
- It then checks each droid in the returned list: - First it sees if the droid is infected (this is a bit abstract of course anyway; presume the SecurityRobot is sending some code that exposes that information if you like to think of this as a story. Anyway, in the actual code, the droids just return a boolean in their response.
  - If the droid is clean, it moves on to the next one. Otherwise...
  - The SecurityRobot, having confirmed that this robot is a threat, begins firing shots. Messages are exchanged confirming how much damage is taken and whether or not the droid is still alive. The SecurityRobot fires at the droid until it's confirmed to be dead.
- Once that's all done, the SecurityRobot declares "mission accomplished" and shuts down the hive. Simulation over!

So! Lots of code, but most of it familiar. There are two new things though!

Previously when we wrote code, we might have yielded on reply just to confirm that the message we sent was handled before we continued. In this case, we actually need some data. You may notice that there's a new format here:

```
response = yield self.wait_on_message(
    to=recipient
    directive="some_directive")
```

Any time that a coroutine is resumed after being suspended with a yield, that's because the actor received a message "in_reply_to" the original outgoing message's message id. Since we're getting a message back, we can of course look at that message... hence the *response* being assigned to the left of the yield. This is another Message object, just like the message argument passed in at the start of the message handler.

This means that if you need to write complex asynchronous logic that needs message passed around back and forth, writing such code looks nearly as simple as normal method calling. It's just that this time, it's encapsulated in message passing! But imagine trying to accomplish this method above with callbacks... it would require splitting between a lot of callbacks. Nested inline or not, that can get pretty confusing. With XUDD, it's easy!

The last thing that's new is the *self.hive.send_shutdown()* call. Yes, this does exactly what it sounds like... it shuts down the Hive. Simulation over!

Okay! Let's run this thing!

Okay, whew! That was a lot of code, and a lot of explaining! What does it actually look like when we run it? It's mostly what you'd expect:

```
$ python xudd/demos/simple_robotscanner.py
Entering room 6pjMdqWIQKGrELiAAcmwwQ...
iHrqJnTmT_yEmzQxQuA2uA is clean... moving on.
QTqPLAsnSq2VFIbF0EGPrw found to be infected... taking out
Droid QTqPLAsnSq2VFIbF0EGPrw shot; taken 42 damage. Still alive... 8 hp left.
Droid QTqPLAsnSq2VFIbF0EGPrw shot; taken 33 damage. Terminated.
ATaO3FQzTZmAv6zOvlB3LQ is clean... moving on.
Ays2zH70TXCwA7FTkZKGug found to be infected... taking out
Droid Ays2zH70TXCwA7FTkZKGug shot; taken 31 damage. Still alive... 19 hp left.
Droid Ays2zH70TXCwA7FTkZKGug shot; taken 11 damage. Still alive... 8 hp left.
Droid Ays2zH70TXCwA7FTkZKGug shot; taken 34 damage. Terminated.
qrKnae_7QF237HVZiO-gKw found to be infected... taking out
Droid qrKnae_7QF237HVZiO-gKw shot; taken 14 damage. Still alive... 36 hp left.
Droid qrKnae_7QF237HVZiO-gKw shot; taken 54 damage. Terminated.
cMrc96qGRzWP9CtY4wh70A found to be infected... taking out
Droid cMrc96qGRzWP9CtY4wh70A shot; taken 48 damage. Still alive... 2 hp left.
Droid cMrc96qGRzWP9CtY4wh70A shot; taken 15 damage. Terminated.
gB4LFt3IRk-rfL8U2TUPnQ is clean... moving on.
SIvh6l24TIKSH7y3M1MXDQ found to be infected... taking out
Droid SIvh6l24TIKSH7y3M1MXDQ shot; taken 38 damage. Still alive... 12 hp left.
Droid SIvh6l24TIKSH7y3M1MXDQ shot; taken 40 damage. Terminated.
nunaOJWNQVK2Ya9oB3UI8Q found to be infected... taking out
Droid nunaOJWNQVK2Ya9oB3UI8Q shot; taken 40 damage. Still alive... 10 hp left.
Droid nunaOJWNQVK2Ya9oB3UI8Q shot; taken 12 damage. Terminated.
2JPFYDhpQ-ijOehrwfgIEA found to be infected... taking out
Droid 2JPFYDhpQ-ijOehrwfgIEA shot; taken 33 damage. Still alive... 17 hp left.
Droid 2JPFYDhpQ-ijOehrwfgIEA shot; taken 35 damage. Terminated.
JwIDRV2eS5mAdIX_s9zbdA is clean... moving on.
Kg07A6hCRMC3eFHE4eDcvA found to be infected... taking out
Droid Kg07A6hCRMC3eFHE4eDcvA shot; taken 36 damage. Still alive... 14 hp left.
Droid Kg07A6hCRMC3eFHE4eDcvA shot; taken 21 damage. Terminated.
TxMl7_-9S5OGsNDcJ0reYw is clean... moving on.
Mission accomplished.
```

Pretty cool eh? If you made it this far, nice work! That was a lot of explaining above, but you now the basics to get up and running coding in XUDD.

Where to go from here

If you want to see the completed demo, this demo is included with XUDD. Check out *xudd/demos/simple_robotscanner.py*.

If you want to look at a slightly more complex version, there's also *xudd/demos/robotscanner.py* which has several extra layers: multiple rooms, sending feedback back to the Overseer, etc. *robotscanner.py* is the first program ever written in XUDD, and was written before the actual system was completed with very few modifications. We're happy to say that the initial demo worked with very few tweaks after the initial pieces of the engine fell into place... this is partly because XUDD's design is so simple! The above may seem like a lot of code, but we hope you'll find that XUDD's implementation of the actor model is straightforward, easy to understand, and comfortable to code in.

If you're looking for more code examples, there's some more in *xudd/demos/* as well.

And of course, if you're ready to start learning more and doing more coding, you should move on with reading this manual.

Good luck, and have fun!

CHAPTER 3

Core design

High level overview

Summary of actors, messages, and hives

This document focuses on XUDD's core design. XUDD follows the actor model. The high level of this is that

There are three essential components to XUDD's design:

- **Actors:** Actors are encapsulations of some functionality. They run independently of each other and manage their own resources.

  Actors do not directly interfere with each others resources, but they have mechanisms, via message passing, to get properties of each other, request changes, or some other actions. Actors can also spawn other actors via their relationship with their Hive. Actors do not get direct access to other actors as objects, but instead just get references to their ids. This is a feature: in theory this means that actors can just as easily communicate with an actor as if it is local as if it were over the network.

  In XUDD's design, Actors are generally not "always running"... instead, they are woken up as needed to process messages. (The exception to this being "Dedicated Actors"; more on this later.)

- **Messages:** As said, actors communicate via message passing. Messages are easy to understand: like email, they have information about who the message is from, instructions on who the message can go to, as well as a body of information (in this case, a dictionary/hashtable).

  They also contain some other information, such as "directives" that specify what action the receiving actor should take (assuming they know how to handle such things), and can inform the receiving actor that they are waiting on a response (more on this and coroutines later).

  Messages also include tooling so they can be serialized and sent between processes or over a network.

- **The Hive:** Every actor is associated with a "Hive", which manages a set of actors. The Hive is responsible for passing messages from actor to actor. For standard actors, the Hive also handles "waking actors up" and handling their execution of tasks. (More on this later, since that wording is possibly confusing.)

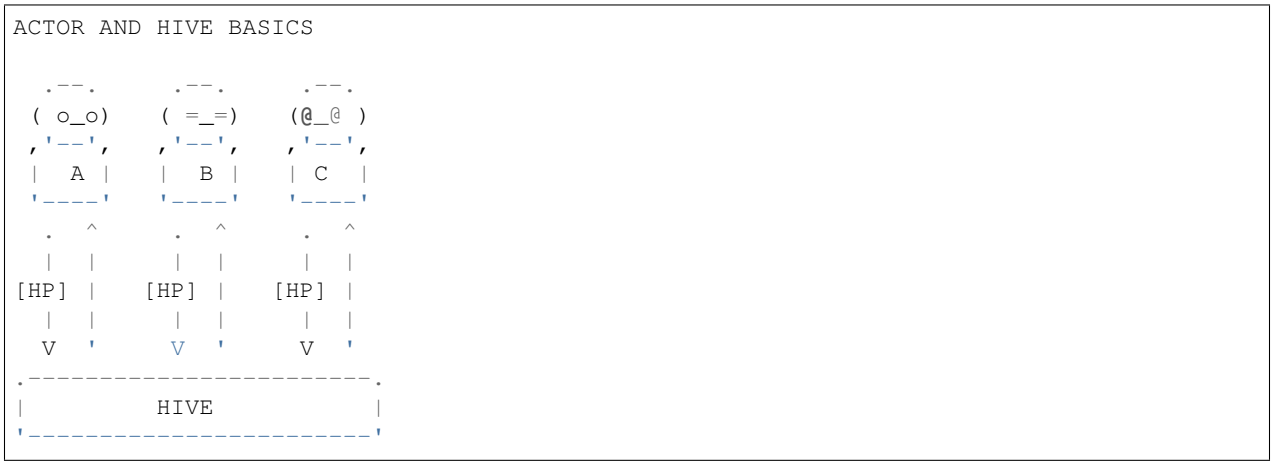
Actors do not get direct access to the Hive, but instead have a "HiveProxy" object. They use this to send messages from actor to actor, initializing new actors, or requesting shutdown of the hive and all actors.

These concepts are expanded on later in this document. Additional features/components that are planned as part of XUDD's design (some of these are yet to be implemented):

- **Inter-hive messaging:**
- **Dedicated actors:**
- **Actor "event" subscriptions:**
- **Property API:**
- **Actor serialization:**

Tying it all together

So, the above explains the relationships between actors, messaging, and hives.



Here we have the basic relationship between a hive and three of its actors, A B and C. Each one has its own unique id, shared by no other actor on the hive. You can see that there are also relationships between an actor on the hive. The hive has direct access to an actor, but actors don't have direct access to the hive... they have to go through a HiveProxy. There's good reason for this: hives may have more methods than should be exposed to actors. In fact, it's entirely possible for an actor to be hooked up to a hive that operates very differently than the "basic" hive that XUDD ships with. By using the HiveProxy, the actor doesn't actually need to know anything about how the Hive works: as long as it uses the HiveProxy methods, those operate just fine.

You can see how this works in code:

Actors

Instantiating actors

Instantiating an actor is easy. You can use a reference to the hive to actually initialize an actor, like so:

```
hive = Hive()

# Create an actor with no additional parameters
evil_bot = hive.create_actor(
```

```
    EvilRobot)

# Create an actor with specific arguments and keyword arguments
# in this case, the first argument is the BattleBot's infantry rank
minion_bot = hive.create_actor(
    BattleBot, "minion",
    weapons=["pokey stick"])

# We can also assign an explicit id, as long as it's unique, and not "hive"
# (that's reserved for the hive)
admiral_bot = hive.create_actor(
    BattleBot, "admiral",
    weapons=["missiles", "spears", "deathray"],
    id="admiral_robo")
```

In each case, the value returned by hive.create_actor is not the actor itself, but an id (a unicode string) that the actor can later be messaged with.

The core properties of an actor

Technically, an actor is only *required* to have the following properties:

1. It should accept two positional arguments: hive, and id. (It may accept more arguments and keyword arguments during construction than this, but it *must* accept these as the first two arguments!)

- **hive:** A HiveProxy object. This is what the actor uses to communicate back to the Hive itself. (Note: this isn't the same thing as the hive itself... it's a proxy object. Actors shouldn't be able to access all the properties and methods of their hive. The HiveProxy provides a universal API.)
- **id:** The id of the actor. The actor need not supply these manually, it will be provided for it by the hive (and possibly by whatever initializes the actor).

2. It should have a method called *handle_message*. Accepts a single argument, "message", which is a *message object*. It should examine the *directive* and other parameters (such as *in_reply_to*) to determine how to most appropriately handle the message.

   The default actor provides a robust *handle_message* implementation that handles passing messages off to various *message handlers*, permitting message handlers to "suspend" themselves via coroutines while they *yield in wait for replies*, as well as features such as automatically *replying to messages*. Your actor does not have to use this logic, though it's recommended that if you do deviate from the patterns in the basic actor's *handle_message*, do so with care! There are safeguards there to make sure that actors waiting on replies are less likely to keep coroutines in waiting forever in case your actor doesn't make an explicit reply!

Handling messages

The default behavior for actors is to pass off messages to a message handler method like so:

```
class RascallyRabbit(Actor):
    def __init__(self, hive, id):
        super(Overseer, self).__init__(hive, id)

        self.message_routing.update(
            {"do_tricks": self.do_tricks})

    def do_tricks(self, message):
        trick_hunter(message.from_id)
```

In the above example, if our "RascallyRabbit" gets a message with the directive "do_tricks", that message will be executed by the *do_tricks* method.

But messages and handling messages is a whole big topic, so let's examine that below.

Messages

A bit on messages

Messages

The message object

class xudd.message.**Message**(*to*, *directive*, *from_id*, *id*, *body=None*, *in_reply_to=None*, *wants_reply=False*, *hive_proxy=None*)

Encapsulation of message data.

This is what's actually passed to an actor's *handle_message* method. While messages can actually be serialized into json or msgpack data, (and methods for that are provided,) this is the standard representation for passing around messages in XUDD itself.

Usually, however, actors themselves do not construct Message objects: these are instead constructed by the Hive itself. Actors send off messages using their HiveProxy.send_message() method.

Args:

- **to:** the id of the receiving actor
- **directive:** what kind of action or request we're making of the receiving actor. Usually this is some kind of useful instruction or request. For example, we might be communicating with a Dragon actor, and we might give it the directive "breathe_fire", which a dragon actor knows how to handle. However, if we're just replying to messages, frequently this directive is simply "reply".

  In the future, there will also be a standardized set of common "error" directives :)
- **from_id:** the id of the actor sending this message
- **id:** the id of this message itself. Usually constructed by the Hive itself (but available to the actor sending the message also, often used to track "waiting on replies" for coroutines-in-waiting)
- **body:** a dictionary of data; the payload of the message. (if None, will be converted to an empty dict.) This can be anything, with a couple of caveats:
  - If there's any possibility of sending this across the wire via inter-hive communication, the contents of "body" ABSOLUTELY MUST be json encodeable.
  - If the message is just being sent for local actor to local actor, it's acceptable to pass along whatever, but keep in mind that you are effectively breaking any possibility of inter-hive communication between these actors!
  - If you are sending along ANY mutable structures, your actor must NEVER ACCESS THOSE OBJECTS AGAIN. Not for reading, not for writing. If you do otherwise, consider yourself breaking the rules, and you are on THIN ICE. This includes basic structures, such as lists. If you have any doubt, consider using copy.deepcopy() on objects you're passing into here.
  - "sanitize" options (with some performance pentalties) may be added in the future that will force-transform into json or msgpack and back, but those don't exist yet.

- **in_reply_to:** The message id of a previous message that we're responding to. This may be used by the actor we're sending this to for waking back up coroutines that are awaiting this response.
- **wants_reply:** Informs the actor receiving this that we want some kind of response. In general, actors will respect this; if a message requests a response, an actor absolutely should provide one, one way or another. The plus side is that we have some tooling built in to make this easy. See replying-to-messages for details.
- **hive_proxy:** In order for the auto-replying tools to work, a hive_proxy must be constructed, which generally is the same hive_proxy the receiving actor has. When constructing a Message object, you don't necessarily have to pass this in when initializing the object, but you should attach this to the message.hive_proxy object before passing to the message queue of the actor.

Sending messages from actor to actor

Message queues and the two types of actors

Basic actors

Dedicated actors

Yielding for replies

Replying to messages

Explicitly replying

The auto-reply mechanism

Deferring your reply

Hives

Hive-level overview

The hive is itself an actor!

If your mind just exploded, that's okay. Take a moment to allow it to reassemble. Minds have a way of being able to do that.

The way this works is a bit tricky to think about, but the cool

Variants on the standard Hive

CHAPTER 4

Asyncio Support

Behind the scenes, XUDD makes use of asyncio to do message passing. But XUDD has a nice interoperability layer where your actors can interface nicely with the rest of the asyncio ecosystem.

Like message passing with XUDD, asyncio makes heavy use of *yield* and *yield from*. However, the astute reader may notice that the way this is called is pretty different than XUDD's message passing... this is because how *yield from* would work in XUDD was designed far before asyncio integration.

Nonetheless, the differences are not so big, and thanks to asyncio and XUDD's clever interoperability layer, you can make use of a tremendous amount of asyncio features such as asynchronous network and filesystem communication, timer systems, and much more.

Asyncio by example

A simple IRC bot

For a good example of this, let's look at this simple IRC bot (no need to follow it all, we'll break it down):

```
"""
"""
from __future__ import print_function

import asyncio
import logging
import sys

from xudd.actor import Actor
from xudd.hive import Hive

_log = logging.getLogger(__name__)


IRC_EOL = b'\r\n'
```

```
class IrcBot(Actor):
    def __init__(self, hive, id,
                 nick, user=None,
                 realname="XUDD Bot 2",
                 connect_hostname="irc.freenode.net",
                 connect_port=6667):
        super().__init__(hive, id)

        self.realname = realname
        self.nick = nick
        self.user = user or nick

        self.connect_hostname = connect_hostname
        self.connect_port = connect_port

        self.authenticated = False
        self.reader = None
        self.writer = None

        self.message_routing.update(
            {"connect_and_run": self.connect_and_run})

    def connect_and_run(self, message):
        self.reader, self.writer = yield from asyncio.open_connection(
            message.body.get("hostname", self.connect_hostname),
            message.body.get("port", self.connect_port))

        self.login()
        while True:
            line = yield from self.reader.readline()
            line = line.decode("utf-8")
            self.handle_line(line)

    def login(self):
        _log.info('Logging in')
        lines = [
            'USER {user} {hostname} {servername} :{realname}'.format(
                        user=self.user,
                        hostname='*',
                        servername='*',
                        realname=self.realname
            ),
            'NICK {nick}'.format(nick=self.nick)]
        self.send_lines(lines)

    def send_lines(self, lines):
        for line in lines:
            line = line.encode("utf-8") + IRC_EOL
            self.writer.write(line)

    def handle_line(self, line):
        _log.debug(line.strip())


def main():
    logging.basicConfig(level=logging.DEBUG)
```

```
    # Fails stupidly if no username given
    try:
        username = sys.argv[1]
    except IndexError:
        raise IndexError("You gotta provide a username as first arg, yo")

    hive = Hive()
    irc_bot = hive.create_actor(IrcBot, username)

    hive.send_message(
        to=irc_bot,
        directive="connect_and_run")

    hive.run()


if __name__ == "__main__":
    main()
```

This bot, as written above, doesn't do much... it just logs in and spits out all messages it receives to the log as debugging info.

Nonetheless, that might look daunting. From the *main()* method though, it's obvious that the first thing done is to handle a *connect_and_run* method on the IRC bot (the handler of which just so happens to be *connect_and_run()*. So let's look at that method in detail:

```
def connect_and_run(self, message):
    self.reader, self.writer = yield from asyncio.open_connection(
        message.body.get("hostname", self.connect_hostname),
        message.body.get("port", self.connect_port))

    self.login()
    while True:
        line = yield from self.reader.readline()
        line = line.decode("utf-8")
        self.handle_line(line)
```

This little snippet of code does almost the entirety of the busywork in this IRC bot. You can see two uses of *yield from* interfacing with asyncio here.

The first line sets up a simple socket connection. You can see that this uses "yield from" to be come back with the transport and protocol (reader and writer) objects once the connection is available. This is a standard asyncio method! (As you'll notice, there's nothing wrapped in a message in this case, because we're not doing message passing between actors here.)

The next line calls *self.login()*... if we follow this method, we'll notice this method itself calls *self.send_lines()*. This method interfaces with asyncio via *self.writer.write(line)*, but since it does not wait on anything, it can call the writer without anything special happening.

Finally, the connect_and_run() enters a loop that runs forever... it waits for new data to come in and handles it. (In this case, "handling" it means simply logging the line... but we might do something more complex later!)

As you can see, the user of XUDD mostly can just call asyncio coroutines from a message handler and things should work.

(Note: if you need to call an asyncio coroutine from a subroutine of your message handler, this can be trickier... you will have to make sure that your subroutine is itself a coroutine and *yield from* that too! TODO: show an example.)

CHAPTER 5

XUDD Marketing For People Who Like The Word “Cloud”

Ever wanted to write actors in the cloud? Now with XUDD, YOU CAN!

XUDD’s an asynchronous actor model system. Run a cloud of actors! Dynamic load balancing across actor pools! It’s so simple! You just set and up and forget about it, instant web scale.

Cooperative multitasking, dynamically? There’s an actor for that.

Event driven development?? TALK ABOUT IDIOTS. We’ve figured out the future and we’re going to be super smug about it, because that’s what sells products. And with XUDD, we’re all about products. Product driven development.

With XUDD, we’re agile, and we definitely SCRUM. We’ve got a dedicated SCRUM-lord who runs stand-up meetings. SCRUM solves all the problems: if you had a problem with the way you develop code, just follow the SCRUM, it’ll fix it, otherwise you don’t understand the problem. But with XUDD, we understand all the problems, which is why we don’t have any.

We’ve got all the clouds with XUDD. Remote clouds? Local clouds? Public clouds? Private clouds? We’ve got all of them. ALL THE CLOUDS.

All of them.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Index

M

Message (class in xudd.message), 20