XUDD Documentation

Release 0.1.0

XUDD contributors

Sep 27, 2017

Contents

```
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/cwebber/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 asyncronous 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

Introduction

```
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
```

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 succeded 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 mesages to each other.

Distributed data crunching

Federation daemon

Some simple code examples

Excited? Let's dive in.

CHAPTER 2

Core design

High level overview

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, intializing new actors, or requesting shutdown of the hive and all actors.

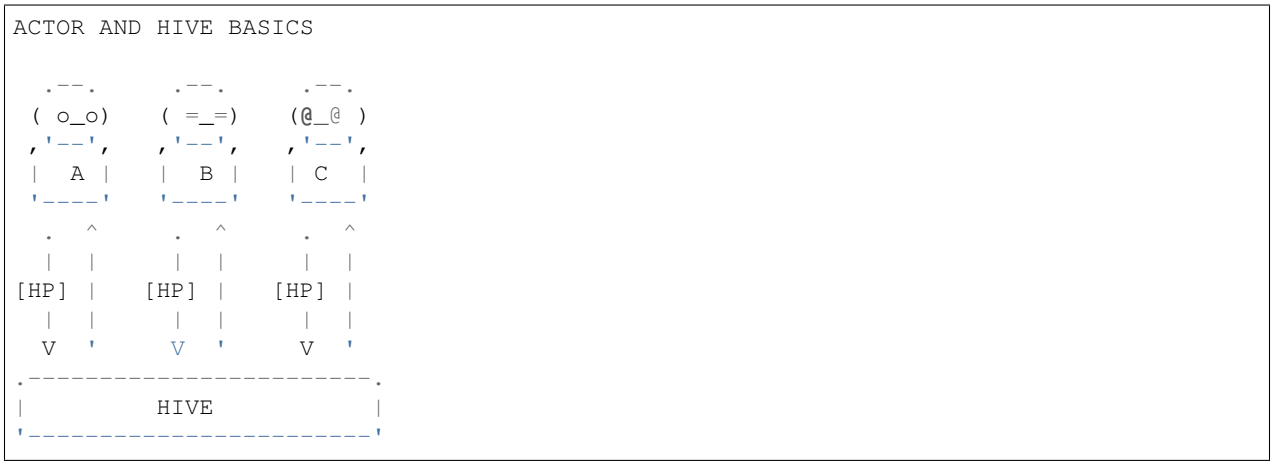
These concepts are expanded on later in this document. Additional features/components that are planned as part of XUDD's design (some of these are yet to be implemented):

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

Detailed design

Actors, hives, and other actors

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:

Sending messages from actor to actor

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 put in an actor's message queue. 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.

Message queues and the two types of actors

Basic actors

Dedicated actors

Yielding for replies

Replying to messages

Hives

Variants and meta-discussion

Hives

The standard hive

Variants on the standard Hive

CHAPTER 3

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 asychronous 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 4

Indices and tables

- genindex
- modindex
- search

Index

M