Pulp - Task #2951

Add the pulpcore.plugin.download.asyncio package with asyncio based downloaders

08/01/2017 11:45 PM - bmbouter

Status:	CLOSED - CURRENTRELEASE	**Start date:**	
Priority:	High	**Due date:**	
Assignee:	bmbouter	**% Done:**	100%
Category:		**Estimated time:**	0:00 hour
Sprint/Milestone:	3.0.0		
Platform Release:		**Tags:**	
Groomed:	No	**Sprint:**	Sprint 25
Sprint Candidate:	Yes	**Quarter:**	

Description

The way we offer concurrency in the plugin API is with the Batch() object which is built on code that we have to carry and maintain. Concurrent I/O (such as concurrent downloads) is a great fit for asyncio and we should:

- Create a ConcurrentHTTPDownloader which is designed to be managed by an asyncio event loop created by the plugin writer. This is described some on the wiki as the ConcurrentDownloader.
- Delete the Batch() object and it's supporting machinery. I think that is effectively everything in batch.py
- Add a ContentUnitDownloader to the download API, which is described here as the Content Unit Downloader

Associated revisions

Revision 1b36eeee - 09/18/2017 11:37 PM - bmbouter

Introducing asyncio downloaders

Adds a HttpDownloader

- handles both synchronous and asynchronous downloads
- designed for subclassing to make custom downloader
- size validation works
- digest validation works
- has a well defined return interface DownloadResult

Adds a FileDownloader

Adds a BaseDownloader

Reworks the Factory

- Updates the Factory to use HttpDownloader and FileDownloader
- Handles http, https, and file
- Fully configures HttpDownloader with all importer settings including: ssl client and server options, basic auth, proxy, proxy_auth
- Can be reused with custom downloaders

Adds a GroupDownloader

- Downloads files in parallel but only return back to the use when any group of files are fully downloaded.
- Drivable with or with generators
- Constrains in-memory objects through generator use

General Updates

- Updates the importer to use the Factory
- Updates the docs to use docs from the new code

https://pulp.plan.io/issues/2951 closes #2951

Revision 1b36eeee - 09/18/2017 11:37 PM - bmbouter

Introducing asyncio downloaders

Adds a HttpDownloader

- handles both synchronous and asynchronous downloads

- designed for subclassing to make custom downloader
- size validation works
- digest validation works
- has a well defined return interface DownloadResult

Adds a FileDownloader

Adds a BaseDownloader

Reworks the Factory

- Updates the Factory to use HttpDownloader and FileDownloader
- Handles http, https, and file
- Fully configures HttpDownloader with all importer settings including: ssl client and server options, basic auth, proxy, proxy_auth
- Can be reused with custom downloaders

Adds a GroupDownloader

- Downloads files in parallel but only return back to the use when any group of files are fully downloaded.
- Drivable with or with generators
- Constrains in-memory objects through generator use

General Updates

- Updates the importer to use the Factory
- Updates the docs to use docs from the new code

https://pulp.plan.io/issues/2951 closes #2951

Revision bca3b3ef - 09/19/2017 04:35 PM - bmbouter

Adds finalize() to BaseDownloader

- reworks the file-object creation to be simpler
- adds finalize()
- switches FileDownloader, HttpDownloader to use it
- adds docs

https://pulp.plan.io/issues/2951 re #2951

Revision bca3b3ef - 09/19/2017 04:35 PM - bmbouter

Adds finalize() to BaseDownloader

- reworks the file-object creation to be simpler
- adds finalize()
- switches FileDownloader, HttpDownloader to use it
- adds docs

https://pulp.plan.io/issues/2951 re #2951

Revision 66f9fd66 - 09/19/2017 09:05 PM - bmbouter

GroupDownloader now validates

The GroupDownloader was not correctly setting the expected_size and expected_digests even though it could. This fixes the GroupDownloader so that it sets those expected values causing validation to happen correctly.

https://pulp.plan.io/issues/2951 re #2951

Revision 66f9fd66 - 09/19/2017 09:05 PM - bmbouter

GroupDownloader now validates

The GroupDownloader was not correctly setting the expected_size and expected_digests even though it could. This fixes the GroupDownloader so that it sets those expected values causing validation to happen correctly.

https://pulp.plan.io/issues/2951 re #2951

History

#1 - 08/02/2017 12:06 AM - bmbouter

- Description updated

Added checklist items and a link to the Batch implementation.

#2 - 08/02/2017 12:21 AM - bmbouter

- Description updated

#3 - 08/02/2017 05:02 PM - mhrivnak

The Batch class has this nice example of its use:

```
        >>> from pulpcore.download import Batch, HttpDownload, DownloadError
        >>>
        >>> url = 'http://content.org/dog.rpm'
        >>> path = '/tmp/working/dog.rpm'
        >>> downloads = (HttpDownload(url, path) for _ in range(10))
        >>>
        >>> with Batch(downloads) as batch:
        >>>     for plan in batch():
        >>>         try:
        >>>             plan.result()
        >>>         except DownloadError:
        >>>             # An error occurred.
        >>>         else:
        >>>             # Use the downloaded file \o/
```

Could we see a similar example, even just a theoretical mock-up, of what this would look like using asyncio?

Switching download libraries introduces a substantial risk. There are many factors that have introduced pain in the past when switching libraries including proxy support, SSL client cert support, connection pooling with SSL, quirks of handling content-encoding headers, authentication to proxy vs. destination, etc. If we are to switch download libraries, I think we should assume a certain level of pain, some of which we may not discover for some time, and thus we would need a very compelling reason to switch. I like the idea of asyncio, but this aspect is scary.

One nice behavior we get with threading is that while receiving or sending data, a thread is not subject to the GIL, and so other threads are able to also engage in active IO or do other things (like write to the database, move a file into place, etc). When downloading in multiple coroutines, does asyncio allow true parallelism in the same way? I've been reading through docs and haven't found anything conclusive. Since coroutines all run in a single thread, I'm guessing that there is not an opportunity for parallelism, but maybe others have additional information?

Could we do database access from an asyncio coroutine? Is that safe? What about a message broker? Would kombu and django's ORM need to be asyncio-aware?

I don't mean to be overly skeptical, but I do see a lot of risk in moving to asyncio. Perhaps answers to the above questions will help. But the other side of the coin is: what exactly are we gaining? Is it just removing the Batch class? Or is there more?

#4 - 08/02/2017 05:13 PM - bmbouter

The main goal is to remove all of this code. That is a huge benefit considering that that is almost identical to what core Python provides itself.

As an FYI, threads are constrained by the GIL, and only in special situations are they not subject to it.
https://wiki.python.org/moin/GlobalInterpreterLock

We all want to see a prototype. That is the first step of this ticket. If that is what is needed to answer some of these questions then that can be done.

#5 - 08/03/2017 07:57 AM - mikea

I wouldn't mind taking a crack at this

#6 - 08/03/2017 05:35 PM - bmbouter

After some in person covo with mhrivnak, he wants to separate this ticket into two things:

1) The changes to the plugin API. This would be removing the Batch object and replacing it with ConcurrentHTTPDownload and ConcurrentFTPDownload

2) The rewrite of the machinery behind ^ objects to switch from our custom implementation to asyncio

I'm going to try to split these out today

#7 - 08/03/2017 11:51 PM - bmbouter

I rewrote the issue, but while doing so I realized that we can't separate these ideas. The Batch() object can't be removed without replacing it with some kind of scheduler like asyncio. So give that I think this has to be done as one piece of work. Given that, do others have suggestions about how this can be split up into smaller pieces of work?

#8 - 08/04/2017 04:48 AM - mikea

I was researching this a bit and while it's possible to use requests, it's definitely not clean or ideal.

I think the best option is to use aiohttp, but we could always look at something like

https://pypi.python.org/pypi/grequests - by the author of requests

https://github.com/ross/requests-futures - closer to what batch actually does.

#9 - 08/04/2017 05:46 PM - bmbouter

mikea, +1 to this prototype should use aiohttp since that is the what I'm seeing is the generally recommended downloading library that is for asyncio. Would you want to collaborate on a branch for a prototype? The first one to put up some code should post a link here.

#10 - 08/04/2017 06:53 PM - jortel@redhat.com

Just some initial thoughts after my first look at asyncio.

The batch.py that provides the Batch class contains 230 lines of actual python code (comments and white space removed). The Batch is intended to provide a simple synchronous interface for concurrent execution using concurrent.futures[1]. The actual concurrency is delegated to *concurrent.future* .

Here this the value proposition for Batch vs. using futures directly:

1. Constrain memory usage.

The ThreadPoolExecutor does not restrict the queue size which result in all items (downloads) to be executed all exist in memory. Assuming the downloader needs linkage back to the artifact and that artifacts have linkage to content, this means that all content units will would be in memory. This has caused OOM problems in the past. The thin executor subclass provided in batch.py and used by Batch limits the queue size. This is better for streams processing.

2. Support synchronous coding patterns.

The futures ThreadPoolExecutor.submit() returns a Future and expects the caller to either keep track of it and/or use callbacks (asynchronous). The batch returns an iterator of completed downloads (or Plan to be specific). This pattern eliminates the need for the caller to track each download while supporting a synchronous (like) flow in code. Except for building a generator of downloads and an additional *for* loop, the plugin writer can code multiple (concurrent) downloads as almost as simply as a single download. The code for downloading is 5-10 lines of code in 1 place. In this context, it seems that having the caller use callbacks would deviate from stream-processing. This pattern also supports the caller choosing in which thread post-download work is performed. A download worker thread, the main thread or other.

—

Looking at asyncio

After looking at asyncio documentation and tutorials, it seems like *asyncio* vs *concurrent.futures* is a more appropriate comparison than *Batch* vs *asyncio*. I could be missing something as I have not spent much time digging. This is where an example of how Batch would be replaced with asyncio would be helpful. Or, please point me to documentation that corrects my understanding.

In summary, asyncio seems like a really useful lib. I'm just trying to get my head around exactly what is being proposed here and how it will affect a plugin writer.

[1] https://docs.python.org/3/library/concurrent.futures.html

#11 - 08/04/2017 07:58 PM - bmbouter

@jortel, I agree with those goals of constrained memory usage and synchronous coding patterns.

I think we need a "quick and dirty" prototype that demos some basic downloading assuming the plugin writer's code instantiates the asyncio scheduler. Once we can actually see it we can make a better comparison. This prototype can even be outside of Pulp and just have the downloader code and a main program that pretends to be the plugin writer's code.

#12 - 08/11/2017 05:15 PM - bmbouter

OK the prototype is done, and it turned out to be pretty sweet (I think). This prototype downloads three files concurrently. asyncio provides several options regarding how exceptions are raised (or not) and also if you want to wait for all downloads to complete before continuing, or if you want to handle each one as it finishes. This prototype shows both ways using some commented code.

```
import aiohttp
import asyncio
import async_timeout
import os

async def ConcurrentHttpDownloader(url, timeout=10):
    async with aiohttp.ClientSession() as session:
        with async_timeout.timeout(timeout):
            async with session.get(url) as response:
```

```
                filename = os.path.basename(url)
                with open(filename, 'wb') as f_handle:
                    while True:
                        chunk = await response.content.read(1024)
                        if not chunk:
                            print('Finished downloading {filename}'.format(filename=filename))
                            break
                        f_handle.write(chunk)
                        # push each chunk through the digest and size validators
            await response.release()
            return filename  # TODO have this return the digest and size validators

def main():
    urls = ["https://repos.fedorapeople.org/pulp/pulp/fixtures/file/1.iso",
            "https://repos.fedorapeople.org/pulp/pulp/fixtures/file/2.iso",
            "https://repos.fedorapeople.org/pulp/pulp/fixtures/file/3.iso"
            ]
    not_done = [ConcurrentHttpDownloader(url) for url in urls]

    loop = asyncio.get_event_loop()

    # to wait for all to finish without exception raising support
    # done, not_done = loop.run_until_complete(asyncio.wait(unfinished, return_when=asyncio.ALL_COMPLETED))

    # to stream process each result
    # will not raise an exception in the foreground, but done results will have the 'exception'
    # attribute set and not the 'result' attribute in the case of an exception
    while not_done:
        done, not_done = loop.run_until_complete(asyncio.wait(not_done, return_when=asyncio.FIRST_COMPLETED))
        print('finished = %s' % done)
        print('unfinished = %s' % not_done)

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

#13 - 08/14/2017 04:53 PM - mhrivnak

Two questions on this implementation:

1. Is it possible to limit the number of items being worked on at a time? I assume it is possible, but I don't see it here. It's a required feature for Pulp.

2. @jortel described above "Support synchronous coding patterns", which is an important part of how the existing implementation makes it easy to do concurrency. The plugin writer hands in a generator and receives back a generator. Is that pattern possible here?

#14 - 08/14/2017 06:08 PM - mhrivnak

Regarding my questions in https://pulp.plan.io/issues/2951#note-3, I've made some progress toward answering them.

I can't find any evidence that using asyncio with a non-asyncio-aware library is unsafe, so that's good. I assume that the lack of safety when using non-coroutine concurrency side-by-side with previous event loops, like eventlets, does not apply here.

When we see people writing about asyncio, saying things along the lines of "When you use asyncio, you need your full stack to be asyncio-aware", I think that's not about safety, but simply about support for parallelism. My understanding is that code running in a coroutine that intends to run "in the background" in parallel must know how to proactively release control back to the event loop. This is used particularly (maybe only?) when doing network I/O. This is why we could not use the "requests" library; it does not know about the event loop or how to return control to it. Each call would block the event loop and execute synchronously. The purpose-built "aiohttp" library knows how to unblock the event loop, so it can be used to do parallelism.

We have a pattern in Pulp 2 that has served us well during sync. A small number of threads each accomplishes the following work items as a "job". I might not have the ordering exactly right, but each thread has a loop that goes through this workflow.

- download a file
- validate the file
- move the file into place
- validate it in-place
- create and/or update database records as necessary

This work is largely complementary from a parallelism standpoint. While one thread is writing to the database, two others are still downloading, etc. We are able to utilize each constrained resource at the same time (remote network I/O, filesystem I/O, database I/O), which is a key value point for parallelism.

Of course as a reminder, python threads have parallelism limited by the GIL. But I/O operations of all kinds are exempt, so they can happen in parallel regardless of what type of I/O they are (http, database, filesystem, etc).

A main concern with asyncio is that we can only do parallelism for resources where an asyncio-aware library exists and can be used. Django does not

support an asyncio-aware database backend, so that would be a net loss. It also appears that filesystem I/O cannot be done with asyncio, so that would be a net loss. In the example above, every time you call "f_handle.write(chunk)", that blocks the event loop.

Please correct this if my understanding is off. My take-away at this point is that using asyncio would substantially degrade Pulp's ability to do download-related workflows with parallelism. It would also require us to change download libraries, which itself comes with substantial work required and risk, and would limit our options in the future for using different libraries.

I like asyncio a lot in concept. It's a much more elegant, and potentially efficient, way of scheduling concurrent work. After all, I'm a golang fan and love some goroutines! But for Pulp right now, this would seem to come with major downsides, and it's not clear to me that there is nearly enough upside to offset that.

#15 - 08/14/2017 06:26 PM - bmbouter

mhrivnak wrote:

> Two questions on this implementation:
>
> 1. Is it possible to limit the number of items being worked on at a time? I assume it is possible, but I don't see it here. It's a required feature for Pulp.

Yes, this can be done by limiting the connection pool size. Limiting the connection pool will limit the number of co-routines which have allowed to even start making a request from remote servers.

> 2. @jortel described above "Support synchronous coding patterns", which is an important part of how the existing implementation makes it easy to do concurrency. The plugin writer hands in a generator and receives back a generator. Is that pattern possible here?

This is a design goal of the prototype also and it does that. The plugin writer never registers an asynchornous callback so they only do synchronous programming. Instead they only deal with an iterator that yields either downloads one-by-one or content units as they are ready one-by-one depending on the level of abstraction we want to provide.

#16 - 08/14/2017 07:06 PM - bmbouter

mhrivnak wrote:

> A main concern with asyncio is that we can only do parallelism for resources where an asyncio-aware library exists and can be used. Django does not support an asyncio-aware database backend, so that would be a net loss. It also appears that filesystem I/O cannot be done with asyncio, so that would be a net loss. In the example above, every time you call "f_handle.write(chunk)", that blocks the event loop.

Note that data still continues to flow into the kernel sockets asynchronously even during these calls, so we're not preventing I/O even while we block in the few cases that we would.

> Please correct this if my understanding is off. My take-away at this point is that using asyncio would substantially degrade Pulp's ability to do download-related workflows with parallelism.

This design should be at least as good, if not better, performing than it's nectar-style equivalent design (a custom thread pool). Note that the critical path to complete a sync is WAN I/O. The aiohttp library is designed to yield control back better than the requests+thread because threading and requests don't know about each other while asyncio and aiohttp do. This was the reason asyncio was created to start with, to massively parallelize I/O based workloads. If thread pools for I/O workloads were significantly higher performing than asyncio wouldn't have a place in the Python.

> It would also require us to change download libraries, which itself comes with substantial work required and risk, and would limit our options in the future for using different libraries.

Pulp3 is a green-field implementation, and the aiohttp prototype was very easy to put together and works well. What substantial work are you describing? Also in terms of risk, aiohttp is suitable for production use and is recommended by the Python community. It is currently at version 2.2.0, which is well past the 1.0 version.

> I like asyncio a lot in concept. It's a much more elegant, and potentially efficient, way of scheduling concurrent work. After all, I'm a golang fan and love some goroutines! But for Pulp right now, this would seem to come with major downsides, and it's not clear to me that there is nearly enough upside to offset that.

I want to give you even more reasons to like asyncio! Let me retell the upside benefits some. By adopting asyncio for concurrent downloads, Pulp won't be receiving bugs that deal with download scheduling, download cancelling, download reporting, thread queues, or thread management because all of that is handled by asyncio. By adopting asyncio, Pulp is aligning with the Python community instead of carrying a custom implementation that provides similar functionality to the Python standard library. The Python community has already fixed many problems with their implementation, while ours is fresh and new and has had no production usage. asyncio has received lots of production usage. By using features of the Python standard library we will have significantly lower risk and maintenance burden. I also expect it to be higher performing too.

#17 - 08/14/2017 07:46 PM - jortel@redhat.com

@bmbouters, Thanks for the prototype!

As documented here: https://github.com/pulp/pulp/pull/3006, the download API and changeset API is layered to support a healthy separation of concerns. The Download and Batch classes are concerned with downloading and the ChangeSet is primarily concerned with affecting changes to repositories and delegates downloading.

When evaluating *asyncio*, we should begin with a common understand of the use cases for the download API. Let's start with this list. After we reach consensus, we can post to the final list to the description.

As an importer, I need to download single files.
As an importer, I need to download files concurrently.
As an importer, I want to validate downloaded files.
As an importer, I am not required to keep all content (units) and artifacts in memory to support concurrent downloading.
As an importer, I need a way to link a downloaded file to an artifact without keeping all content units and artifacts in memory.
As an importer, I can perform concurrent downloading using a synchronous pattern.
As an importer, concurrent downloads must share resources such as sessions,connection pools and auth tokens across individual downloads.
As an importer I can customize how downloading is performed. For example, to support mirror lists
As an importer, concurrent downloading must limit the number of simultaneous connections. Downloading 5k artifacts cannot open 5k connections.
As an importer, I can terminate concurrent downlading at any point and not leak resources.
As an importer, I can download using any protocol. Starting with HTTP/HTTPS and FTP.

As the streamer, I need to download files related to published metadata but delegate *the implementation* (protocol, settings, credentials) to the importer. The implementation must be a black-box.
As the streamer, I can download using any protocol supported by the importer.
As the streamer, I want to validate downloaded files.
As the streamer, concurrent downloads must share resources such as sessions,connection pools and auth tokens across individual downloads without having knowledge of such things.
As the streamer, I need to support complex downloading such as mirror lists. This complexity must be delegated to the importer.
As the streamer, I need to bridge the downloaded bit stream to the Twisted response. The file is not written to disk.
As the streamer, I need to forward HTTP headers from the download response to the twisted response.
As the streamer, I can download using (the same) custom logic as the importer such as supporting mirror lists

#18 - 08/14/2017 09:01 PM - bmbouter

@jortel, thank you for writing these use cases. They all look pretty good, and they all make sense. I can see how all use cases (importer and streamer) will be able to be satisfied by an asyncio design.

It's a lot to talk about though, so to move forward, I think we should focus on the importer use cases for now. It will allow us to not tackle too many issues at once. The motivation to focus on the importer first is to get the plugin API ready for plugin writers to start trying. We can fix the streamer up post-alpha, pre-beta. So with that in mind, I'm going to sort the importer's use cases you wrote out to connect them with what the prototype already does. If you want to move the importer use cases to the description I think that would be fine. If you want to put the streamer's use cases into a new issue for later discussion that would also be fine.

> As an importer, I need to download single files.

We can keep the existing downloaders and switch them to asyncio later (pre-beta). If we don't also want to depend on requests, we can trivially use aiohttp with a single coroutine to perform a single download.

> As an importer, I need to download files concurrently.

The prototype does this.

> As an importer, I want to validate downloaded files.

The prototype processes downloaded data in chunks so having it compute the digest/size in a stream based way should be very easy. You can see that todo here.

> As an importer, I am not required to keep all content (units) and artifacts in memory to support concurrent downloading.

The prototype allows you to add additional downloaders after downloading has begun so you can already limit the number of content units and artifacts in memory with the prototype. We could make it easier via a generator or iterator, but it's already possible without much effort.

> As an importer, I need a way to link a downloaded file to an artifact without keeping all content units and artifacts in memory.

The content unit downloader yields all files that are ready to be turned into artifacts and content units both which are immediately saved to the db so the in memory holding of those object types should be very low.

> As an importer, I can perform concurrent downloading using a synchronous pattern.

The co-routine model with asyncio is synchronous. The plugin writer never writes or registers any callbacks.

> As an importer, concurrent downloads must share resources such as sessions,connection pools and auth tokens across individual downloads.

aiohttp is built this way. By default it shares resources across all concurrent requests made by aiohttp. This is also configurable/customizable.

> As an importer I can customize how downloading is performed. For example, to support mirror lists

The prototype does not do this, but it could easily be added as an option.

> As an importer, concurrent downloading must limit the number of simultaneous connections. Downloading 5k artifacts cannot open 5k connections.

aiohttp constrains the number of concurrent connections by default. This is configurable and defaults to 100.

> As an importer, I can terminate concurrent downlading at any point and not leak resources.

asyncio does have explicit cancellation, but it's even easier if if a plugin writer were to return from sync() with a return statement. Once that occurs there would be no active frame references to the asyncio loop, which is the only thing referencing the coroutines, so all of it will be removed from memory with garbage collection. We could also do explicit cancellation, but why?

> As an importer, I can download using any protocol. Starting with HTTP/HTTPS and FTP.

The prototype support HTTP/HTTPS and is build on aiohttp. The aioftp library is like aiohttp library, only it's for FTP. So it would work roughly the same way. I don't think we need FTP even for the MVP release of Pulp3.

#19 - 08/14/2017 09:44 PM - mhrivnak

bmbouter wrote:

> mhrivnak wrote:
>
> > A main concern with asyncio is that we can only do parallelism for resources where an asyncio-aware library exists and can be used. Django does not support an asyncio-aware database backend, so that would be a net loss. It also appears that filesystem I/O cannot be done with asyncio, so that would be a net loss. In the example above, every time you call "f_handle.write(chunk)", that blocks the event loop.
>
> Note that data still continues to flow into the kernel sockets asynchronously even during these calls, so we're not preventing I/O even while we block in the few cases that we would.
>
> > Please correct this if my understanding is off. My take-away at this point is that using asyncio would substantially degrade Pulp's ability to do download-related workflows with parallelism.
>
> This design should be at least as good, if not better, performing than it's nectar-style equivalent design (a custom thread pool). Note that the critical path to complete a sync is WAN I/O. The aiohttp library is designed to yield control back better than the requests+thread because threading and requests don't know about each other while asyncio and aiohttp do. This was the reason asyncio was created to start with, to massively parallelize I/O based workloads. If thread pools for I/O workloads were significantly higher performing than asyncio wouldn't have a place in the Python.

I think we agree on most of this. The trade-off is that asyncio is more efficient at doing HTTP downloads but entirely lacks parallelism for the other kinds of I/O Pulp does. Maybe that's awash. But Pulp 2's thread pool downloading does not have a speed bottleneck we've managed to hit, so any gains in that type of I/O aren't all that valuable.

> It would also require us to change download libraries, which itself comes with substantial work required and risk, and would limit our options in the future for using different libraries.

Pulp3 is a green-field implementation, and the aiohttp prototype was very easy to put together and works well. What substantial work are you describing? Also in terms of risk, aiohttp is suitable for production use and is recommended by the Python community. It is currently at version 2.2.0, which is well past the 1.0 version.

I would not call Pulp 3 a complete green-field implementation. A lot of it, yes, largely out of necessity. The tasking system is an example where we are taking the X release opportunity to change some things, but fundamentally it's the same approach and same libraries being used. I also anticipate that the streamer won't require a lot of change. Likewise, I suspect many of the plugins will be able to re-use a lot of code.

We've been managing a thread pool for downloads for a long time successfully. That's very low-risk. We're benefiting from python 3's concurrent futures which make it a bit more convenient, but otherwise we're not doing anything all that new or different.

Even with the widely-used requests library, we've slowly fished out a number of edge case bugs over a long period of time and done our best to mitigate them. When we've attempted to use other download libraries in the past, it's been a similar experience. There are a **lot** of edge cases, we have a surprising amount of code to deal with them in Pulp 2, and it would take a lot of time to gain that stability again.

> I like asyncio a lot in concept. It's a much more elegant, and potentially efficient, way of scheduling concurrent work. After all, I'm a golang fan and love some goroutines! But for Pulp right now, this would seem to come with major downsides, and it's not clear to me that there is nearly enough upside to offset that.

> I want to give you even more reasons to like asyncio! Let me retell the upside benefits some. By adopting asyncio for concurrent downloads, Pulp won't be receiving bugs that deal with download scheduling, download cancelling, download reporting, thread queues, or thread management because all of that is handled by asyncio. By adopting asyncio, Pulp is aligning with the Python community instead of carrying a custom implementation that provides similar functionality to the Python standard library. The Python community has already fixed many problems with their implementation, while ours is fresh and new and has had no production usage. asyncio has received lots of production usage. By using features of the Python standard library we will have significantly lower risk and maintenance burden. I also expect it to be higher performing too.

I hear you on that. Fortunately, concurrent downloading is an area where we've been very stable and performant for a long time. While the behavior is moving from nectar into a new API, we are not re-inventing the wheel, and thankfully we don't need to.

One thing I do not see is that asyncio is meant to replace the use of threads in the python community. Clearly asyncio is amazing in some circumstances. But python 3 also invested in making it easier to manage thread pools, and they are still very useful for some applications. Thread pool management and event loop management are both in the standard library, and I think we're aligned with the community either way.

#20 - 08/15/2017 08:04 PM - bmbouter

- Subject changed from Replace Batch() with ConcurrentHttpDownloader and ConcurrentFtpDownloader that are asyncio backed to Replace Batch() with ConcurrentHttpDownloader and a ContentUnitDownloader that are asyncio+aiohttp backed

- Description updated

Based on some in-person discussion some of the deliverables are being updated.

#21 - 08/16/2017 03:02 PM - bmbouter

- Status changed from NEW to ASSIGNED

- Assignee set to bmbouter

- Sprint/Milestone set to 43

From a meeting yesterday, we wanted to put this onto the sprint to explore the idea with the expectation that it will undergo lots of review before deciding if we want to merge it.

#22 - 08/18/2017 04:18 PM - mhrivnak

Asyncio vs Thread Pool

I understand that asyncio provides a more efficient way to schedule concurrent network operations when compatible libraries can be used. Pulp 2's threaded downloading does not have performance problems, so for the case of "asyncio could perform better than threading", that appears to be a solution looking for a problem.

I'm not opposed to doing something a better way just for the sake of doing so. When weighing the benefits and costs though, in the performance case there is not much benefit to be had.

I also understand that the asyncio API may be easier to use. That could be worth exploring. My take is that concurrent futures makes thread pool management approximately as easy as asyncio makes coroutine management. There are small differences, but they are designed to be similar, and I think the overall developer experience would have similar complexity either way. For Pulp, either one will require some small amount of code to manage it, and so far that again looks like it would be of similar scale.

A direct comparison would be interesting. But for the reasons articulated below about the required http client library change, I fear that this direction will not be viable in the end, and thus investing in it would not be productive.

Download Library Switch

Because downloading is so fundamental to what Pulp does, there is a lot of risk in switching download libraries in general. Despite how prevalent HTTP is, "doing it correctly" is surprisingly hard. There are 25 years of evolving standards, all manner of good and bad implementations, and everyone's favorite: de facto standards. Any client library can only do its best to accommodate the latest of the standards, enough of the older ones, enough of the de facto ones, and enough of the wrong-but-common edge cases to maximize it usefulness. Judgement calls have to be made, and any two libraries are unlikely to behave exactly the same.

In Pulp's experience so far, hot spots for edge cases tend to be around proxy use and SSL. Correctly handling authentication to a proxy vs. an end server has been a problem. Pooling SSL connections correctly, considering client certificate use, has been a problem. Small details of how header values are returned (what to do when the same header appears twice, for example) and how they are handled (for example should you correct for the huge number of servers that misrepresent the content-encoding of compressed files?).

Looking at requests vs. aiohttp, a few things stand out as meaningful differences.

aiohttp does not support SSL to a proxy. So the proxy URL must use the scheme "http" only. "requests" does support this.
https://github.com/aio-libs/aiohttp/issues/845

Pulp 2 depends on the "requests" library to handle retries. Aiohttp does not have such a feature, and the maintainer thinks the feature is out of scope.

https://groups.google.com/forum/#!topic/aio-libs/p4sEZVUXdWo

Documentation about how to use client SSL certificates was only added recently. That doesn't necessarily present an incompatibility, but it does not inspire confidence that it has been used much. That is of course a critical feature for us.

https://github.com/aio-libs/aiohttp/pull/1849

These are differences that happened to be easy to find quickly, but I am confident there are more to discover.

To further illustrate the extent to which it is difficult to make a "correct" http client library, I quickly browsed aiohttp's recent bugs and pulled these as examples of complex issues. It can be difficult to determine what the "right" thing to do is for some of these issues, and others are just tricky edge cases that take time and use to discover.

https://github.com/aio-libs/aiohttp/issues/2146
https://github.com/aio-libs/aiohttp/issues/2076
https://github.com/aio-libs/aiohttp/issues/1821
https://github.com/aio-libs/aiohttp/issues/1843
https://github.com/aio-libs/aiohttp/issues/2143
https://github.com/aio-libs/aiohttp/pull/2173

It's worth noting that aiohttp was released as 1.0 only 11 months ago. It's off to a great start, but that is young for an http client library. We can see in their issue tracker that they are still discovering and sorting out a lot of tricky details.

If Pulp were to change download libraries at this point, we would need a very compelling reason to do so. As observed recently on a call, it has taken years to fish out all the quirks and bugs in "requests" that impact Pulp. Changing to a different library, especially a very young one, would reset that clock.

But just as importantly, Pulp has enough adoption and use at this point that we must be careful about introducing regressions, even in an X release. Our users have a wide variety of environments. Changing download libraries introduces the risk that a currently-working environment may become unsupportable due to library limitations. It's hard to know for sure that all of the same proxies would be compatible, for example. For this reason in particular, we need to have a very compelling reason to change http client libraries.

In the case of asyncio, using it does not appear to me to provide a big upside. I'm open to being convinced, but there's not much to improve with performance, and otherwise we seem to be talking about maybe a marginal improvement in plugin writer experience.

For all of those reasons, I do not think it makes sense to start using asyncio at this time.

#23 - 09/05/2017 06:04 PM - jortel@redhat.com

- Sprint/Milestone changed from 43 to 44

#24 - 09/19/2017 12:27 AM - bmbouter

- Status changed from ASSIGNED to MODIFIED

- % Done changed from 0 to 100

Applied in changeset pulp|1b36eeeedaa3bd349518e37992743fbbcb411d72.

#25 - 09/19/2017 03:25 PM - bmbouter

- Subject changed from Replace Batch() with ConcurrentHttpDownloader and a ContentUnitDownloader that are asyncio+aiohttp backed to Add the pulpcore.plugin.download.asyncio package with asyncio based downloaders

#26 - 12/19/2017 04:24 PM - bmbouter

- Tags deleted (Pulp 3 Plugin Writer Alpha)

Cleaning up Redmine tags

#27 - 03/09/2018 12:23 AM - bmbouter

- Sprint set to Sprint 25

#28 - 03/09/2018 12:24 AM - bmbouter

- Sprint/Milestone deleted (44)

#29 - 04/25/2019 06:46 PM - daviddavis

- Sprint/Milestone set to 3.0.0

#30 - 04/26/2019 10:38 PM - bmbouter

- Tags deleted (Pulp 3)

#31 - 12/13/2019 06:25 PM - bmbouter

- Status changed from MODIFIED to CLOSED - CURRENTRELEASE